Lossless Data Compression in VLSI

PROEFONTWERP

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. M. Rem, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen op
dinsdag 2 december 1997 om 16.00 uur.

door
Leonard Cornelis Benschop
geboren te Gorinchem

Dit proefontwerp is goedgekeurd door de promotoren:

prof.ir. M.P.J. Stevens
en
prof.Dr.-Ing. J.A.G. Jess

Copromotor: dr.ir. A.C. Verschueren

druk: Universiteitsdrukkerij Eindhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Benschop, Leonard Cornelis

Lossless data compression in VLSI / by Leonard Cornelis Benschop. -
Eindhoven : Technische Universiteit Eindhoven, 1997.
Proefschrift. - ISBN 90-386-0400-9
NUGI 832
Trefw.: datacompressie / toepassingsspecifieke geintegreerde
schakelingen / grote geintegreerde schakelingen / digitale systemen.
Subject headings: data compression / application specific integrated
circuits / VLSI / digital systems.

Summary

Data compression makes it possible to store more data on the same medium or to transmit more data on the same channel. When high-speed media or channels are used, high speed data compression is desired. Software implementations are often not fast enough.

The objective of this thesis is the design of a VLSI circuit that performs LZH (Lempel-Ziv and Huffman) compression at a speed of 100Mbit per second. The compressed data is compatible with the compressed data of existing compression software.

We examined all the requirements for such a circuit and created a specification document. The VLSI circuit can be used as a peripheral device within a normal computer system. It works independently of the main CPU. It reads the input data from RAM and stores the output data into RAM through DMA.

We created a detailed architecture consisting of several communicating modules: a bus interface, a sliding window coder, a buffering and statistics module, a Huffman coder and an internal processor. Some parts of the compression algorithm are implemented in software on the internal processor. The chip makes use of an external RAM and it contains several internal memories. The thesis contains a very detailed description of the design process and all design decisions that led to this architecture. We implemented and simulated several essential parts of the chip. We used several simulation methods at different levels. Many design decisions were based on simulation results.

To achieve the desired speed for the sliding window coder, we decided to implement it as an associative array. Associative arrays are large arrays of identical cells that can match their stored contents with a certain value (c.f. associative memories) and perform operations. They can give a dramatical speed up for some algorithms, especially sliding window coding. Several variations of this associative array were examined.

The sliding window coding array is the largest part of the chip and it dominates the cost. It is possible to make several versions of the chip, each with a different sliding window coder size, but with mutually compatible compressed data.

We can conclude that the chip can be fabricated and meets the stated requirements.

Samenvatting

Datacompressie maakt het mogelijk meer data op hetzelfde medium op te slaan of meer data over hetzelfde kanaal te versturen. Wanneer gebruik wordt gemaakt van media of kanalen met een hoge snelheid, wordt datacompressie met een hoge snelheid gewenst. Software-implementaties zijn vaak niet snel genoeg.

Doel van dit onderzoek is het ontwerp van een VLSI schakeling die LZH (Lempel-Ziv en Huffman) compressie uitvoert met een snelheid van 100Mbit per seconde. De gecomprimeerde data is compatibel met bestaande compressiesoftware.

We onderzochten alle eisen die aan zo'n schakeling gesteld worden en creëerden een specificatiedocument. De VLSI schakeling kan gebruikt worden als een randapparaat binnen een normaal computersysteem en werkt onafhankelijk van de centrale processor. Door middel van DMA worden de in- en uitvoerdata in het RAM gelezen en geschreven.

We creëerden een gedetailleerde architectuur bestaande uit verscheidene communicerende modules: een bus-interface, een 'sliding window coder', een buffer- en statistiekmodule, een Huffman-coder en een interne processor. Delen van het compressiealgoritme worden in software geïmplementeerd op de interne processor. De chip gebruikt een externe RAM en bevat verscheidene interne geheugens. Dit proefschrift bevat een zeer gedetailleerde beschrijving van het ontwerpproces en alle ontwerpbeslissingen die tot deze architectuur geleid hebben. We implementeerden en simuleerden verscheidene essentiële delen van de chip. We gebruikten verscheidene simulatiemethoden op verschillende niveaus. Veel beslissingen zijn gebaseerd op simulatieresultaten.

Om de gewenste snelheid voor de 'sliding window coder' te behalen, hebben we besloten deze als associatieve array te implementeren. Associatieve arrays zijn arrays van identieke cellen die hun inhoud kunnen vergelijken met een bepaalde waarde (vgl. associatieve geheugens) en bewerkingen kunnen uitvoeren. Zij kunnen een dramatische snelheidsverbetering geven voor sommige algoritmes, vooral 'sliding window coding'. Verscheidene variaties van deze array werden onderzocht.

De array voor 'sliding window coding' is het grootste onderdeel van de chip en overheerst de kosten. Men kan verschillende versies van de chip maken, met een sliding window coder van verschillende grootte, maar met onderling uitwisselbare gecomprimeerde data.

We kunnen concluderen dat de chip gefabriceerd kan worden en aan de gestelde eisen voldoet.

Acknowledgements

I would like to thank professor M.P.J. Stevens, who gave me the opportunity to work on this thesis.

I would like to thank professor J. Jess and professor J.P.M. Schalkwijk.

I express my special gratitude to Ad Verschueren, who helped me and my graduate students solve numerous problems with IDaSS.

René Wientjes also deserves my special credit for providing numerous hints towards the efficient implementation of the sliding window coder. The essential ideas of the associative cell array are developed by him, prof. Stevens and myself.

Ardo van Rangelrooij and Gert-Jan de Vos also provided some helpful hints during the design process.

Next I would like to thank Alex Wijffels and Johan Camps who designed two essential parts of the chip for their graduate work.

The ICS group at the Eindhoven University of Technology proved to be a marvellous work environment, which was caused by people like Lech Jóźwiak, Jeroen Voeten, Piet van der Putten, Marcel Kolsteren, Harald Vranken, Pieter Schoenmakers, Ad Chamboné and Rian van Gaalen.

Finally I would like to thank all my other colleagues at the ICS group, my friends and my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the goals of engineering is making optimal use of available resources. Data compression is one method of achieving this goal. More specifically, data compression achieves more efficient use of storage media (such as hard disks and tapes) and communication channels (such as telephone lines and LANs) for digital information. While some techniques simply aim at squeezing more bits into the storage medium or communication channel (channel coding), data compression aims at reducing the number of bits needed to represent the same information.

Information theory revolves around two closely related topics:

- Source coding is the technique to represent information in the smallest number of bits while still retaining the required quality. This includes compression of digital data but also quantization of non-digital information.

- Channel coding is the technique to convey as many bits as possible through a given channel and do so reliably. This includes modulation techniques for analog channels and magnetic media, error detection and error correction.

Data compression has become a very broad technological discipline. There is at least one annual conference entirely devoted to it (Data Compression Conference, Snowbird Utah). There are dedicated compression methods for text, various classes of still pictures, audio and video. Data compression methods vary greatly in compression factor, speed, required processing power, required memory and loss of quality of the digital information.

Text compression is a comparatively minor subtopic of data compression. The information to be compressed consists primarily of texts, such as word processor documents and program source files. The majority of files typically found on the hard disk of a computer, such as database files, spread sheets and executable programs are usually compressible quite well with the same methods, unless these files are already compressed.

To make use of compression one needs a compressor to compress the data before it is transmitted or stored and a decompressor to reconstruct the original data from the compressed data. The compressor and decompressor can be implemented in software, firmware or hardware.

This thesis describes the design of a VLSI hardware implementation of a compressor and decompressor for text compression.

1.1 Overview of the Problem Domain

The field of data compression can be divided into two subfields: lossless compression and lossy compression.

- With lossless compression the original data is exactly reconstructed after decompression. Examples of lossless compression include:
  - Generic text compression. This is the focus of the thesis. The source data consists of arbitrary sequences of characters (in most situations characters are 8-bit bytes). These methods are optimized for computerized text, but also usable for binary object code and bitmapped pictures. Not all possible character sequences can actually be compressed if compression is guaranteed to be lossless. This can be proved by a simple counting argument.
  - Application specific text compression. Some texts are known to contain frequent occurrences of certain words or are known to conform to a certain syntax. Program source code is a good example of this. Keyword compression (one byte for each keyword) was common in early BASIC interpreters. Syntax directed compression of program source files was discussed in [DW94].
  - Application specific compression of other data. For audio and for still images (fax, medical images) there exist specific lossless compression methods.
- With lossy compression the original data cannot be exactly reconstructed after decompression, but the observable differences from the original are usually very small. Lossy compression is always specific to one type of data because one needs to know what the bits signify and what their influence is on the observable quality of the data. There are specific lossy compression methods for digitized photographs (JPEG) [Wal91], video (MPEG) [LG91], speech (CELP) [SA85] and high quality audio (based on subband coding) [VBW89]. There are two reasons to use lossy compression over lossless compression:
  - A better compression ratio is generally achievable than with lossless compression, especially if the source data contains significant amounts of noise. Lossless compression cannot compress the noise (as it is random), lossy compression can eliminate it (as it is does not contribute to the quality of the information).

– A fixed compression ratio can be achieved, which is inherently impossible with lossless compression. This is needed to transfer video or audio over a fixed bandwidth channel.

There are also reasons *not* to use lossy compression:

– In an environment where the same information is repeatedly compressed and decompressed (video editing) the loss of quality of lossy compression will accumulate and may seriously affect the quality of the signal.

– Small details of an image may be insignificant to the overall visual impression, but they might convey critical information, for example in medical images. Even the remote possibility of losing such information will lead to legal problems.

1.1.1 Text Compression Theory

A text compression method has as its input a sequence of symbols of a source alphabet and converts it to a sequence of symbols in a destination alphabet. In most cases the source alphabet is the set of 8-bit bytes and the destination alphabet is the binary alphabet $\{0, 1\}$. The bits of the compressed data are usually grouped into 8-bit bytes. The decompression method reconstructs the same original source sequence from the compressed data.

Compression generally works by removing redundancy from the source data. The real information content of a text is usually much less than 8 bits per byte and the rest is redundancy, which can theoretically be removed.

There are several classes of text compression methods:

- Entropy coders encode each source symbol based on its probability. The entropy H of symbol x from an alphabet A is defined as:

  $$H(x) = \sum_{i \in A} -P_i \log_2 P_i$$

  where P_i is the probability of symbol i. An entropy coder encodes each symbol in such a way that the number of bits per symbol of the compressed data approaches the entropy on average.

  The encoding has to satisfy the requirement of unique decodability. Prefix codes always satisfy this requirement. Prefix codes are variable length codes in which no code word is the prefix of an other code word. This way a stream of bits can uniquely be parsed into code words.

  Huffman codes are the most well-known entropy codes and they are prefix codes. The algorithm invented by Huffman [Huf52] constructs an optimal prefix code from the probabilities of all symbols.

Huffman coding is optimal in the case that every single encoded symbol is represented by a whole number of bits. Arithmetic coding does not have this restriction and can achieve better results, especially if the probability of one of the input symbols is close to 1.

Arithmetic coding treats the entire message (consisting of several symbols) as a subinterval of the real interval $[0, 1)$. For each encoded symbol the original interval is reduced proportionally to its probability. The compressed data is a binary representation of a fractional number that uniquely identifies the message. The higher the overall probability of the message, the larger the eventual subinterval associated with it, the fewer bits are needed to represent the number adequately. Practical arithmetic coders use integer arithmetic and reduce accuracy to minimize the number of computations. Arithmetic coding methods are described by Neil, Cleary and Witten [WNC87], Langdon and Rissanen [LR82] and many others.

Before an entropy coder can compress (or decompress) the next symbol it needs to know the probability of each of the possible symbols. A *model* provides estimates for these probabilities. Several types of models exist with varying complexity and varying performance:

- The zero order model is the simplest model. The probability of a given symbol is considered independent of the context it appears in. The model is constructed by counting the frequencies of all symbols in the text to be compressed. A representation of the model must be transferred along with the compressed data. Zero order models are optimal for *memoryless* information sources only and text is not a memoryless source. This model combined with Huffman coding is called static Huffman coding.
- In an adaptive zero order model the probability of a given symbol is still considered independent of the context, but the probability estimates are adjusted while the data is compressed or decompressed. As the compressor and decompressor adapt their models in exactly the same way the model information need not be transferred separately. This model can be integrated with Huffman coding and this is called dynamic Huffman coding [Gal78].
- In Markov models the probability of a given symbol is modelled by a finite state machine. In most practical Markov models the state is entirely determined by a finite context consisting of 1, 2 or more preceding symbols. Sometimes the context consists of neighbouring symbols in more dimensions (as in 2-dimensional pictures) [Ris83].

  Most practical Markov models are adaptive. When compression is started, the model is simple (zero order) and it is developed while more statistics are available during compression.

  Examples of Markov modelling include PPM [BCW89] (pp. 152–153), dynamic Markov coding [BCW89] (pp. 167–205) and context tree weighting

[WST95]. Context tree weighting achieves the best compression ratio currently available [Vol96]. See [Wil91] for a few more adaptive modelling techniques.

These models are mostly used with arithmetic coding.

- Dictionary based methods replace strings of input symbols that are found in a dictionary with a reference to a dictionary entry. The dictionary is maintained adaptively.
  - Sliding Window Coding (or LZ77) methods are based on the original paper by Lempel and Ziv [ZL77]. They store the last N input symbols in a buffer. The dictionary consists of all substrings of that buffer. The coder output consists of references to such substrings (consisting of an address and a length) and literal symbols that were not encountered in the dictionary.

    The lengths, string addresses and literals output by the sliding window coder are often compressed with static or dynamic Huffman coding. Many widely used software programs use this combination, often called LZH. It yields better compression than either sliding window coding or Huffman coding alone.
  - LZ78 based methods are based on another paper by Lempel and Ziv [ZL78]. They use a dictionary that consists of a list of strings. Single-symbol strings are always part of the dictionary. The compressed data consists of indices of strings in the dictionary. Several variations exist. The Lempel Ziv Welch algorithm (LZW) is a popular variation of this method [Wel84].
  - Some methods combine some of the features of LZ77 and LZ78 methods [Yok92, GL96]. In [SS82] generalized substitution compression methods (comprising LZ77 en LZ78) are discussed.
- Several other methods exist that do not fall into the previous categories.

  Run length encoders convert runs of identical consecutive symbols to a representation that includes a repetition count. They are only applicable to data that contains long runs of identical symbols, such as bitmapped graphics.

  Move to front coders maintain all possible input symbols in a list and output the position of the input symbol in the list. Recently used symbols are moved to the front of the list. The position of symbols close to the front of the list is encoded in fewer bits than the position of other symbols. An entropy coder can be used for that.

  The Burrows Wheeler transform [BW94] rearranges a block of characters in such a way that it is better compressible by other methods such as move to front. It is not itself a compression method but it forms the basis for excellent compression methods. The compression ratio is comparable to PPM while it is faster and uses less memory.

The Burrows Wheeler transform was not available when this project was started so it was not considered as a candidate compression algorithm. It uses too much memory anyway.

1.1.2 Existing Hardware and Software

The existing text compression hardware and software can be divided into three categories:

- The first category consists of dedicated compression programs, often incorporated in an archiver. Their emphasis is on compression ratio and speed is not very critical. An archiver combines multiple (possibly compressed) files into a single compressed file that can be transferred as a whole. For an overview of archivers see [GN96] (pp. 23–25). The following types of products exist:
  - Standalone compressors. These are often combined with standalone archivers. Under CP/M files were 'squeezed' (static Huffman coding) and packed into LBR files. These tasks were performed by separate programs. Under Unix `tar` is the archiver and `compress` (LZW) or `gzip` (LZH) [Gai93] is the compressor.
  - Archivers with builtin compression. These include ARC, ZIP, ZOO, LHA [TO92], ARJ, UC2 and RAR. In older programs the LZW algorithm was used and LZH is used in most modern archivers.
  - High-end experimental compressors and archivers. These have a significantly better compression than programs from the other categories, but they are rarely used in practice due to their slow speed or their memory consumption, which ranges from hundreds of kilobytes to tens of megabytes. PPM with arithmetic coding is a very popular method. Examples include HA, ACB and X1. The program `bzip` [Sew96] is an experimental compressor employing the Burrows Wheeler transform. The Information Theory Group at the Eindhoven University of Technology performs experimental compression with context tree weighting [Vol96].
  - EXE packers. These compress an executable program and link a small decompressor with it so that the program can decompress itself on the fly every time it is executed. These programs include LZEXE and PKLITE. LZ77 algorithms are very suitable due to their fast decompression.
- The second category consists of on-the-fly compression products in software and firmware (which is software on an embedded processor). The emphasis is on speed rather than on compression ratio. The following products exist:
  - Disk compressors (Stacker, DoubleSpace). These compress all data that is read or written on the hard disk. The algorithms are LZ77 variants.

– V42bis modems [Tho92]. Firmware in the modem compresses data on the fly using the LZW algorithm.
– Backup devices. Tape devices sometimes contain compression in firmware and many backup programs implement compression as well.
– Libraries. Several libraries are available, both commercially (PKWARE) or free (`zlib`). These can be used by other programs to implement compression.

- The third category consists of hardware implementations of compression methods. They typically implement compression at much higher speeds than would be possible with software.

  – Massively parallel dictionary-based compressors. Many of these are only designed at the architecture level. Storer [Sto88, Sto92] designed many Lempel-Ziv architectures with systolic arrays. One of the architectures with a dynamic dictionary was implemented as a prototype in VLSI, using an array of 30 processors, each containing 128 cells. Improvements are described in [SRM90].

    Zito-Wolf [ZW90] described a VLSI implementation of a systolic array for sliding window coding, based on one of Storer's architectures. They put 128 processors (good for 256 bytes) on one chip.

    Hirschberg et al.[HS94] describe parallel implementations of dictionary-based compression methods on parallel processors with shared RAM.

    IBM's ALDC compressors [Cra95] have made it into a commercial product. They employ an LZ77 algorithm with a window size of 512–2048 bytes. Improved architectures with parallel dictionaries are described in [FRT96].

    Groot [Gro91] designed an LZ78-based coder with content addressable memories.

    Further there are many recent patents on LZ77-based compressors using content addressable memories [HCPAG94, Par94, WOF^{+}96].

  – Other compressors. Mukkerjee [MNM91] designed a Huffman coder. Jones [Jon92] implemented move to front compression at 100Mbit/s.

1.1.3 Future Applications

In the world of microprocessing there is a trend towards higher speed. Disks, networks and tape devices tend to have higher and higher speeds each year.

The following speeds are available at the moment or in the near future:

- The fastest hard disks currently available have speeds in the order of 10MByte/s.
- Fast local area networks have a data rate of 100Mbit/s (12.5MByte/s).

- The fastest tape devices commonly used for backups on workstations achieve a rate of 1MByte/s. In [PML93] the IBM 3490 tape backup system is described, having a throughput of 10MByte/s with compression.

In all those situations compression is desirable as it increases the effective capacity. On-the-fly compression is preferred over stand alone compression as it eliminates the need for separate storage of uncompressed files.

Compression should not degrade the effective speed of disks, networks and tape devices. If the compression device is fast enough with respect to the (storage or communications) media it increases the effective speed as well as the effective bandwidth.

If compression devices achieve a good compression factor as well as a satisfactory speed, if they are reliable and have a reasonable cost, then there will be a market for such devices.

Hardware devices offer the following advantages over software on the main CPU of a computer system:

- They are much faster if they must perform compression with a comparable compression ratio.
- They do their work without tying up the main CPU of a computer system, which is an advantage in multitasking environments.

Most standard embedded processors have a much lower processing power than the main CPU so firmware implementations on embedded processors are even slower than software on the main CPU, unless a fast or specialized embedded processor is used.

1.1.4 Savings and Limitations

The goal of data compression is saving storage media or communications bandwidth. For typical texts data compression can achieve a savings of a factor 2 to 3. Software can be distributed on fewer diskettes (or CD-ROMs), computer users may need smaller hard disks, files are downloaded in less time, etc. Savings in the amount of storage or bandwidth used will not usually translate into a proportional savings in cost. Higher capacity storage media usually cost less per megabyte. On the other hand if even a tiny bit of compression makes upgrading of the existing hard disk unnecessary it can save a considerable amount. As modem connections are billed by connect time and not by the amount of transferred data, compression will save on the cost of downloading large files, but not on the cost of on-line mail reading.

Depending on the speed of the compression and decompression process with respect to the speed of the data storage or communication medium, data compression also reduces the total time needed to store, retrieve or transfer the data.

As compression removes redundancy from the data the consequences of errors in the compressed data are magnified. The loss of a few bytes of a compressed file makes recovery of the remaining data practically impossible. Data compression should not be used over data links without an error correcting protocol. If compression is used on backup tapes, it is not advisable to compress all files in a single compressed data stream. In case of error one will lose all the remaining contents of the backup tape and not just a single file. Generally reliability is increased by splitting the data into small independently compressed blocks (the consequences of an error are confined to one block), but this reduces the compression ratio.

For some files even the loss of a single bit must be considered fatal, e.g. binary object code. In this case compression reduces the probability that a bit error occurs, so it effectively *increases* the overall reliability of the transfer.

It is often desirable to read archived data at some time in the future. At the moment it is very difficult to find hardware that is able to read tapes or disks that were written twenty or thirty years ago. Twenty or thirty years from now it might be as difficult to find hardware capable of reading today's storage media. Obsolete data formats will add to the difficulty of recovering useful data. It is essential that the compression format is well documented and that portable implementations (source code in a widely supported standard language) are available.

Text compression can achieve a savings of a factor 2 to 3, but this savings cannot be achieved again on data that is already compressed. This has the following consequences:

- It is possible to build data compression devices into a hard disk. This way a hard disk that physically stores 1GB of data can hold up to 2.5GB of typical computer files. Can this disk be sold as a 2.5GB disk? No. A bulletin board owner purchases the disk and finds out that it holds only 1GB of compressed files. How to specify the capacity of such a disk is a marketing problem rather than a technical one. A good way would be specifying two figures (raw capacity and typical capacity with compression). For tapes this is already being done.

- Suppose that a communications company exploits a communication link that carries digital data. The company rents bandwidth to customers who pay per megabyte. The company finds out that it can carry twice as much traffic on the same line if compressors are installed. Should the company make the investment? Customers might use compression as well and if the user data is already compressed the company can no longer carry 2Mbit/s of user data over a 1Mbit/s link.

1.2 Problem Statement

The objective of this thesis is the design of a VLSI circuit that meets the following requirements:

- It must have a compression ratio comparable to that of frequently used archivers (2.7–3 bits per byte [1]).
- It must have a speed of 100Mbit per second of uncompressed data, which makes it suitable for use in future high-speed applications.
- It must be flexible enough to be used in many applications. The design should be adaptable to different compression methods.
- It should be compatible with existing compression methods. At the very least there must be software to decompress any data compressed by the device.
- It must be possible to interface the device to many different computer systems.
- It must work reliably.
- It must have an acceptable cost.

These properties should make the device suitable for the applications mentioned in section 1.1.3. It should provide an alternative to existing products that are either not fast enough or either provide a lesser compression ratio.

Due to previous work it was decided to use an LZH-based compression method.

As opposed to many design tasks the requirements were not fully specified and the specification could be changed in the course of the project. While the original intention was to create working silicon, this was no absolute requirement.

1.3 Overview of this Thesis

This thesis consists of a comparatively small main part and a comparatively large set of appendices. The main part describes the design process and the result of it. The emphasis is on the design decisions and the reasons why they were made that way. The appendices provide a reference. They describe in great detail the algorithms, the compression formats, several tests and simulations and a description of the designed product. These descriptions are detailed to the bit level.

The main part is built up as follows:

- Chapter 1 is this introduction.
- Chapters 2 through 4 describe what led to the design process. They cover requirements, the design context and specification respectively.

[1] In the literature about text compression it is common practice to specify the compression factor in bits of compressed data per character of input text. For compressors that can operate on arbitrary binary data, an input character is specified as an (8-bit) byte.

- Chapter 5 describes the design process itself (excluding requirements analysis and specification). This chapter concentrates on behavioural modelling, the architecture and the design of individual parts of the circuit.
- Chapter 6 describes the result of the design process. It provides a description of the designed product and alternative solutions.
- Chapter 7 gives the conclusions and directions for future research.

The five appendices cover the following topics:

- Appendix A: Algorithms.
- Appendix B: Compression formats.
- Appendix C: Compression benchmarks (compression ratio for many variations of the algorithms).
- Appendix D: Simulations at different levels.
- Appendix E: Detailed description of product, such as data sheets, I/O map, instruction set etc.

Chapter 2

Requirements

When a complex system is designed it has to meet a large number of requirements. These requirements often conflict with each other and one has to find a workable compromise.

For the compression device multiple objectives had to be taken care of:

- Speed. The target speed is 100Mbit/s or 12.5MByte/s for uncompressed data. This speed is obtainable in modern high-speed local area networks. It is about 1000 times as fast as the speed of `gzip` on a 25MHz 80386, still a very common computer in early 1993. Even on a 90MHz Pentium processor `gzip` would take 20–100 times as much time as the hardware device.

- Compression ratio. The achievable compression ratio must be significantly better than that of simpler hardware compressors. It should be comparable to that of archiver programs such as LHA or PKZIP. This means a compression ratio of about 3 bits per character.

- Interfacing. The device should be usable as a peripheral in a computer system. It should not require significant participation of the main CPU in the computer system. DMA is the only data transfer method that does not tie up the CPU and that is fast enough in many systems.

  It should be possible to send the compressed data directly to the destination peripheral, saving bandwidth on the system bus.

- Compatibility. The compression method should be compatible with one or more widely used compression formats. At the very least there must exist easily obtainable software to decompress data compressed with the device.

  At the hardware level the device should be compatible with many existing bus architectures. It must compress and decompress the data for many different computer peripherals.

- Flexibility. Ideally the device should be usable for all text compression methods, but that is a rather unrealistic goal unless you design a general purpose computer. It is desirable to make the device usable for many related compression methods.
- Cost. Ideally the device should cost as little as possible. It is clear that the compressor may not cost more than any possible savings it may provide. Further its cost must not be unfavorable with respect to alternative compression products. Cost is mainly determined by the following factors:
  - Chip area which is proportional to the number of gates. External memories are less expensive (per kilobyte) than internal ones, but they increase the number of pins and external components.
  - Number of pins.
  - Design cost (and uncertainty) divided by production volume. Some techniques may reduce the chip area needed, but make the design process harder (increasing the design time and cost) and increase the possibility that the first silicon does not work as simulated. On the other hand the use of standard cells slightly increases the chip area but reduces design cost and uncertainty.
  - External components needed.
- Power dissipation. A high power dissipation contributes to the cost if extra cooling measures are required. The cost of the electricity itself might be of concern in portable equipment. High power dissipation may reduce reliability.
- Functional correctness. The compression algorithm must produce valid compressed data for all possible input data sets. Compression must be lossless in all cases.
- Reliability. The device must be testable and it should contain an internal self-test facility.
- Ease of use. There must be a well-documented interface so programmers know how to access the device. It is not important that this interface is easy to use as this interface can be hidden in a device driver program that is accessible to all programs that use the device. A device driver can make compression completely transparent to the application. It must however be possible to write an efficient device driver, so the interface complexity must not be too high.

2.1 Compression Method

The compression method used is of direct consequence to the available compression ratio and to the complexity of the design. Table 2.1 gives an indication of the compression ratio, the complexity and the memory usage of several compression methods.

Method	Ratio bits/byte	Complexity instructions/byte	Memory kbyte
Huffman	5.0	30	8
Sliding Window	4.0	100	32
LZW	3.6	30	56
LZH	**2.9**	120	**48**
Markov+Arithmetic	2.3	1000	500

Table 2.1: Compression Ratio and Complexity of Methods.

The instruction count and memory usage are based on a typical implementation on a general-purpose computer. Instruction count is a rough estimate based on relative timing of programs. Memory usage is based on a representative example of an implementation. For the sliding window methods an 8k window is assumed (as in LHA), for the LZW algorithm a 13-bit code size is assumed (as in PKZIP 0.91). The program HA was taken as an example of a high-end processor with Markov modelling and arithmetic coding.

Based on table 2.1 we chose the LZH method. Markov modelling and arithmetic coding use about an order of magnitude more memory and computing time than LZH or the other methods. The other methods do not achieve the desired compression ratio.

On a general purpose computer the LZH method would use 12.5MByte/s $\times$ 120 instructions per byte, or 1.5×10^9 instructions per second. It is also clear that the LZH method at the required speed is beyond the capabilities of even the most powerful microprocessors, hence an all-software implementation is impossible. A profiler shows that sliding window coding takes the largest part of the work of a software implementation of LZH and from [Ben93] we know that fast parallel implementations of sliding window coding are possible in the from of associative cell arrays.

Note that the estimated complexity of an algorithm on a general purpose computer does not necessarily give an accurate figure of the complexity of the same algorithm in hardware. Some algorithms can be parallelized better than others depending on the amount of data dependency between iterations of loops. Software algorithms often use extra memory for hashing to speed up searching while associative memories can be used in hardware to do a fully parallel search.

Since the sliding window is in principle a very simple FIFO data structure, sliding window compression seems to profit at least as much from parallelization as all the other methods. Furthermore, the sliding window coding and the Huffman compression are independent processing steps, making them easy to pipeline. The speed of LZH in hardware should be no less than the speed of sliding window coding alone.

Having chosen the LZH compression method we arrive at a design that contains the following parts:

- A fully parallelized sliding window coder that is able to compress one byte per clock cycle. The clock frequency should therefore be at least 12.5MHz.

- A Huffman coder that postprocesses the output of the sliding window coder. The speed of the Huffman coder should be adequate to let the sliding window coder run at its full speed, but not much faster. Its clock frequency can be higher than that of the sliding window coder, preferably a multiple thereof.

Of course the device needs to decompress as well. Both coders should therefore also be capable of decompression. As far as the sliding window coder is concerned the already present FIFO data structure should be a suitable part of a decompressor as well. Decompression capability can be added at little extra cost.

Within the framework of LZH based compressors it is possible to trade compression ratio for complexity, which in turn translates into speed and cost. See Appendix C (comparison of methods) for more details. The size of the sliding window is the main parameter in this respect. Enlarging the sliding window size will lead to an increased chip area, hence an increased cost. Further it will lead to longer path delays through the parallel array, longer delays through the more complex selection logic and hence to a reduced speed. Finally it leads to a higher power dissipation.

In CMOS technology power dissipation is nearly proportional to clock frequency. The power dissipation can be dramatically reduced when the circuit is not active. Given the same circuit the power dissipation is proportional to the clock frequency as is compression speed. If the circuit is made inactive while it does not actually process data, the actual energy consumption for a given amount of data is roughly the same.

2.1.1 Cost of Compatibility

One of the objectives for the design is making it compatible with one or more existing compression methods. Compatibility should work in both directions. Data generated by the hardware device can be decompressed by existing implementations of the method and data compressed with existing implementations can be decompressed by the device. The consequence of all this is that the compression method is determined by whatever standard you choose to be compatible with instead of cost, speed and compression ratio considerations.

Unfortunately there were no *standard* compression methods based on LZH at the time the design was started. The 'deflate' algorithm was in use by both PKZIP and `gzip` and this was considered the de facto standard. Later it became even clearer that this method was a de facto standard. It even made it into an (informational) RFC of the Network Working Group.

Unfortunately 'deflate' uses a sliding window of 32kB. We assumed that 8kB was the maximum sliding window size that we could afford to implement in hardware. To achieve compatibility in both directions we would absolutely need to have a sliding window of 32kB in the decompressor. For the compressor it would be no problem if it had a sliding window of only 8kB as long as the data could be decompressed by existing 'deflate' implementations.

Therefore we decided to choose a compression method that used a sliding window not larger than 8kB. The (obsolete) PKZIP1.1 'implode' method and the method used by both LHA and ZOO2.1 were candidates. As the latter method was used in more than one program we decided to go for that.

At a later stage in the design process it became clear that the device needed a large buffer memory for static Huffman coding, which was not really needed for decompression. As sliding window decompression needs no (parallel) searching it is possible to build a sliding window decompressor around the existing memory at little extra cost. This way the device could use a considerably larger sliding window for decompression than for compression. Therefore the size of the sliding window is no longer dictated by the standard you must be compatible with. This made the implementation of `gzip` possible. Furthermore it becomes possible to define a range of compression devices, each with a larger sliding window for compression, that are all two-way compatible with each other.

When the device is to be compatible with two or more compression methods, this will add to the cost. Things that can be hard-wired in a single-standard compressor should now be programmable. There are many different ways of Huffman encoding addresses, lengths and literals from a sliding window coder. Bits and bytes can be stored in different order in different methods and addresses or lengths are often split in two parts, one of which is Huffman encoded. Those parts of the Huffman process that are not implemented in software are especially costly to make programmable. In chapter 6 we will revisit this topic.

2.2 Interfacing

The device must be usable within a microcomputer system. It is essential that the data can be transported to and from the device fast enough. This means a rate of 12.5MByte/s of uncompressed data plus about 5MByte/s of compressed data. The data transfer to and from the device should demand as little as possible from the CPU in the computer system so DMA is the only realistic method.

In a typical setting the uncompressed data are prepared in memory and are sent to the compressor through DMA. The compressed data are stored in a memory buffer through a different DMA channel. The contents of this memory buffer are later sent to a peripheral device such as a hard disk or network card. During decompression the data are transferred in reverse order. First the data from a peripheral device are stored in a memory buffer. The contents of this memory buffer are sent through DMA to the device for decompression. The decompressed data are stored into memory using a different DMA channel. The decompressed data in memory can be further used by the computer.

Transferring both the uncompressed and the compressed data through DMA simultaneously means a considerable load on the system bus. This is even a bigger problem as the compressed data also has to be transferred to a device, again using the same system bus. It is clear that the ISA bus used in most older PCs is completely unsuitable for the task (DMA throughput maximum 10MByte/s on 16-bit channels). The PCI bus is much better in this respect (maximum throughput 133MByte/s at 32bits). In some system architectures it may be possible to use DMA directly between the compressor and an I/O device thereby saving the intermediate memory storage. Bus mastering would also increase the efficiency of the DMA transfer, but it would add to the cost of the chip (extra pins, 32 address lines needed).

It would be desirable to send the compressed data directly to the device for which it is destined without using the system bus. Likewise it would be desirable to receive compressed data directly from the device when decompressing. Microprocessor systems do not normally have dedicated I/O channels for this purpose (the SCSI bus comes closest). It would be possible to add a dedicated data channel for compressed data that can directly be connected to the controller of a peripheral device. The device could then be included on the controller card of a peripheral. No standards were available for connecting the compressor to a peripheral controller in this way.

It would be desirable to make the device suitable for use in many different computer systems. 8, 16 and 32 bit data transfer are desirable options. Standard DMA handshaking (through DREQ and DACK) and standard communication with the CPU (through interrupts and I/O ports) are also desirable. The desired speed cannot be achieved in all systems that the device can work with.

The clock frequency of the device should be derived from whatever clock is available on the bus of the host computer system.

Chapter 3

Design Context

A design project does not exist in a vacuum. The project exists in an environment and the course of events is determined by many factors like requirements, plans, existing work, people and available means. The requirements have been discussed in the previous chapter. This chapter describes the other elements of the design context.

Every project has a goal to achieve. For this design project the goal is the design of a device that performs data compression and decompression and that meets certain requirements. To achieve that goal one has to make an adequate planning. Plans already exist at the start of the project (based on the existing work, available tools and people) and they are gradually adjusted and refined. Plans have to take into account the available means.

This particular design project is a continuation of previous work. Several design decisions had already been made and we already had an idea of what the solution looked like. Associative arrays were an essential part of that idea.

Besides myself there were many people who contributed to the design project. They implemented certain parts, they advised me or they contributed some useful ideas.

To successfully realize a project one has to make use of tools. These tools may already be available or they have to be obtained or designed. One of the many tools is IDaSS [Ver90], an interactive design and simulation system for digital systems.

3.1 Previous Work

During my study as a post-graduate designer I examined a design method for complex information processing systems. This method revolved around dividing the design into a few independent submodules with well-defined hardware interfaces. One of the design projects examined was a data compression and decompression device using sliding window coding and possibly Huffman coding [Ben93].

We had already obtained the following results:

- Data flow diagrams for sliding window compression and decompression with and without static Huffman coding. Huffman coding was only performed on lengths and literals, not on addresses. These data flow diagrams were based on the method of Ward and Mellor [WM85].

- Comparisons of compression ratios of several variations of sliding window coding and other algorithms. These comparisons ware based on an ad hoc set of files, both text and binary. They showed the superiority of LZH (and LZ77 with fixed codes) over LZW.

- An IDaSS design of a hardware sliding window coder (using an associative cell array) and a 128-cell layout generated by the ASA silicon compiler. The total area of this chip was 31.7mm^2. The clock frequency of a 32-cell version would be 36.9MHz.

- Descriptions of several possible architectures with and without Huffman coding.

  The device could be designed in two environments:

  - As a part of another peripheral. The data would be transferred on dedicated channels.
  - As a computer peripheral using DMA to transfer data to and from the memory of the host system.

  The architecture for a device without Huffman coding specified the following modules:

  - A bus interface to communicate with the outside world.
  - A string matcher and substituter (sliding window coder and decoder).
  - A byte coder and decoder (for fixed prefix coders),
  - A controller.

  The architecture for a device with Huffman coding specified the following modules:

  - A bus interface.
  - A string matcher and substituter.
  - A code buffer to store blocks of sliding window coded data before they could be Huffman coded.
  - A statistics module to gather frequency statistics for the data to be Huffman coded.
  - A programmed prefix coder and decoder (for Huffman codes).
  - A CPU as a controller and for the Huffman tree computations.

The architectures specified several data and event channels, but no (internal) bus. It was already argued that the main data flow (uncompressed bytes, sliding window coded data and compressed bytes) should not be handled by the CPU as this would reduce throughput.

In the case of static Huffman coding it was planned to implement the design in several chips. It was thought that a standard microcontroller would be adequate to perform all Huffman tree computations.

The principle of the parallelized array for sliding window coding was already invented by René Wientjes, prof. Stevens and myself. This was in turn based on string matching techniques described by Lee in [Lee62, LP63].

3.2 Associative Arrays

Associative cell arrays are defined as follows:

- The hardware consists of a large array of identical cells plus a small central controller.
- Each element of a data structure is stored in one cell of the array. Each cell can have a state and can store one or more data fields.
- Each cell can perform elementary operations of an algorithm related to one element of the data structure. In particular the cells can compare their data contents to a global match value that is distributed to all cells.
- Several global signals may exist. Some signals (like the match value) are generated by the central controller and are distributed to all cells in the array. Other signals are generated by the cells and are combined in a certain way (logical and, logical or, number of first cell with 'true' signal) and the result is fed to the controller.

Associative cell arrays differ from systolic arrays because of the existence of global signals. In a true systolic array the cells communicate with neighbouring cells only.

Prior to the start of this project and during its early phases associative cell arrays were proposed as the solution for many problems:

- The sliding window coder was already mentioned.
- Huffman coding was planned to be performed by table lookup and decoding by reverse table lookup in specially constructed CAMs in which each word contained a bit mask. We know these CAMs as functional memories [Hed95].
- Sorting for static Huffman coding would be another candidate for associative arrays.

- Later the principles of functional memories were extended to a parallel algorithm for dynamic Huffman coding.

In his thesis René Wientjes examined the implementation of data structures in hardware using arrays of parallel cells that could do associative matching [Wie95]. Associative string matching was first described by Lee [Lee62, LP63].

Associative cell arrays have the following advantages over other ways to implement certain algorithms:

- They can be much faster than sequential implementations. In theory operation with $O(n)$ time complexity can be reduced to $O(1)$ time complexity.
- They often need less complex per-cell hardware than true systolic array implementations.

They have the following disadvantages and restrictions:

- As all cells perform their operations simultaneously they dissipate much power, much more than a combination of a RAM and a sequential implementation of the algorithm. Systolic arrays also dissipate much power. The cells in an associative cell array probably dissipate less power than systolic arrays, but the global signals take much power to drive.
- Those cells also take much chip area, but probably less than systolic arrays.
- Because global signals exist, the delay of these signals becomes an important factor in the maximum clock frequency. In systolic arrays there is only one global signal (the clock) whose skew increases with increasing array size.
- Adding extra operations to the array often necessitates a complete redesign of the cells.

3.3 Tooling

When we started we had the following design and simulation tools in mind:

- IDaSS is an interactive design and simulation system for complex digital systems such as microprocessors, written by Ad Verschueren [Ver90]. It is based on Smalltalk and lets you interactively edit and simulate diagrams of synchronous digital circuits that consist of registers, memories, combinatorial operators and state machines. This level of description is commonly referred to as RTL (register transfer level).

  IDaSS is the only known tool that lets you interactively modify and simulate a digital circuit at the register transfer level. IDaSS offers the following features:

- It works at the register transfer level. In VHDL one can also work at a comparable level of abstraction, but it takes much discipline. VHDL has so many features that let you work at either much lower or much higher levels of abstraction, while IDaSS forces you to work at one level of abstraction.
- It allows interactive and graphical editing. Other graphical editors work at much lower levels, typically the gate level.
- Because IDaSS works graphically, it avoids the very verbose interconnection specifications that VHDL designs have. IDaSS syntax is much simpler than that of VHDL.
- It allows interactive simulation even while modifying the circuit. No other tools have this capability.

These features make it possible to design a complex digital system much faster than with other tools.

Of course IDaSS is not perfect and I found it had the following problems:

- The underlying version of Smalltalk is showing its age. Its user interface is often awkward to use (no overlapping windows) and it does not take advantage of modern personal computer systems.
- Other tools are not well integrated with it.
- Some IDaSS features are very costly to implement in silicon if at all. A user who designs circuits with IDaSS does not always have a clear idea of the complexity of the hardware. Close interaction with Ad Verschueren proved to be necessary to avoid unnecessarily costly implementations. Later IDaSS versions give a rough indication of the speed of a circuit.
- The then existing version of IDaSS allowed only a register value in the same (sub)schematic as the source of an input signal to a state machine. If one wants to use subschematics that communicate with each other using handshaking, this restriction proves very awkward. The restriction was lifted in later versions of IDaSS, but the design process was still influenced by it. The use of state machines (as an IDaSS primitive) was avoided in several places where they would normally be the natural choice. State machines were simulated with registers and combinatorial operators instead.

The version of IDaSS that was used for the development of the device, works at the register transfer level (RTL). Later versions also have support for objects at the algorithmic level. No automatic translation is available from the algorithmic level to either RTL IDaSS or VHDL.

IDaSS should ultimately include tools for the design at a very high behavioural level, as described in [Ver92].

- The IDaSS to VHDL translator was being worked on during the project. It was planned that all IDaSS designs could be automatically translated to synthesizable VHDL designs. A similar translator already existed for SID, the input language of the ASA silicon compiler.

- The Compass tools are a large collection of IC design tools, consisting of lay-out editors, simulators and a silicon compiler that accepts VHDL as its input language.

  The Compass tools support manual lay-out construction using the 'sticks' lay-out tool and manual lay-out editing.

  Further they support input and editing at the standard cell level, the gate level and the VHDL level.

These tools were considered adequate to create a chip-layout that could be turned into working silicon.

Every working IDaSS design would be translated to VHDL and then be synthesized by the Compass tools. Most parts of the compressor could be implemented that way. The associative arrays could be created by hand using the 'sticks' layout editor from the Compass tools.

Without the use of high-level tools like IDaSS and silicon compilers, a small team cannot design a non-trivial IC. It would be impossible for the following reasons:

- The low-level design of all parts would cost too much effort.

- There would be no automatic verification of low-level descriptions of the system (gate or transistor level) against high-level descriptions (RTL).

- At the lower levels an IC would be impossible to simulate as a whole, due to the enormous number of gates that must be simulated. Errors at the highest levels of the system would certainly go unnoticed.

- If non-standard cells are used, a simulation (a Spice simulation of the extracted circuit) might fail to correctly simulate the real behaviour of even a single gate.

On the other hand. creating an IC is definitely not as simple as writing a well working IDaSS model. Some IDaSS constructs do not translate well to VHDL and the silicon compiler does not always give efficient (or even working) silicon for all VHDL constructs. One has to write the IDaSS design with VHDL in mind and one has to manually edit the VHDL and layout to get an acceptable result. For a design as large as the data compressor, this becomes a very non-trivial task, requiring the participation of specialists.

Apart from the inherent problems of hardware design using automated tools that generate a lay-out from a high-level description, there were a few problems specific to this design trajectory:

- VHDL is not a very suitable language for synthesis (it is a *description* language). Most silicon compilers accepting VHDL accept only a subset of VHDL with arbitrary restrictions. This subset is not standardized and it varies among different versions of the same package. For more information on VHDL see [Bha95].
- Memories (the compressor needs a few) pose specific problems. As memories cannot be generated from (behavioural) VHDL descriptions, each silicon compiler includes some memories in its library, each with its own restrictions. These memories are then available as VHDL entities for simulation. Therefore the available memories in a specific silicon compiler dictate the behaviour of memories at higher levels.
- Both IDaSS and the IDaSS to VHDL translator were moving targets. Versions with some desired features were often not available when the the design of some parts of the device was started. The use of certain features was avoided because they were not available and this avoidance led to less elegant designs. The avoidance of state machines because of input signal restrictions is an example of this.

Although manual layout design of the cells of associative arrays, and especially the use of special tricks like dynamic registers and precharged logic, can save a large percentage of chip area, this requires even more specialized skills and increases the risk that the silicon will not work, while it does work correctly under simulation. Using non-standard cell logic in a complex design is considered infeasible for this university group.

IDaSS and the Compass tools provide simulation of the system under design. The intermediate VHDL can be simulated using dedicated VHDL simulators.

One simulation language not mentioned thus far is the C programming language, because it is not meant to be a simulation language. Yet it became the primary language in which simulations were carried out during this project. My experience with C and the availability of compilers were important factors in the choice of C. C offers easy access to data files and the speed of compiled C is hard to beat. This way large amounts of data could be processed by the simulated algorithm. A distinct disadvantage of C is the lack of parallelism, which was not important for some simulations and could be worked around in others. C came in handy at several stages of the design process and for several levels of simulation. These simulations are described in more detail in appendix D (simulations). C was used for the following simulation tasks:

- The behavioural model of the processing task of the device. As the primary task of the device is a processing task, rather than a communication task, it was possible to simulate the processing without regard to communication. C is a very suitable language to implement compression programs and the C program was fast enough to process megabytes of data.

  Many useful statistics could be deduced from the C program to help optimize the architecture.

- Algorithms of associative cell arrays. IDaSS was not very suitable for large arrays of cells. In VHDL these could have been modelled using the FOR...GENERATE construct. As VHDL models parallelism, this would be a more straightforward modelling.

  It is possible to model the the arrays in C (see appendix D, simulations). State variables and signals can be mapped to C variables and a sequential algorithm can update these variables in the right order.

- Huffman algorithms that were to be carried out on the internal processor.

- Instruction level simulation of the internal processor.

3.4 Design Team

While no design team was officially formed for the project, I did not perform the entire design task alone. Several formal and informal relationships existed between me and other people. Several of these people can be considered members of the 'design team'.

The following people made significant contributions:

- Prof.ir. M.P.J Stevens was my promotor and supervisor for this project.

- Ad Verschueren was the IDaSS expert and helped me make many architectural decisions. His help was invaluable for the design of the internal processor.

- René Wientjes helped invent the first usable associative array for sliding window coding. He also helped me design a lay-out for the cells.

- Two graduate students, Johan Camps and Alex Wijffels, implemented two parts of the design in IDaSS (the bus interface and the internal processor) as their final work. These students were supervised by me.

- My colleague research assistants, while working on different projects, also gave me valuable information. Most notable are Jeroen Voeten (overall design methodology), Ardo van Rangelrooij (CMOS implementation) and Gert-Jan de Vos (internal processor)

- I got some advice regarding Huffman coding from the Information Theory Group at this university: Tjalling Tjalkens, Frans Willems and prof.dr.ir. J.P.M. Schalkwijk.

- Finally some helpful hints are due to Egied Bormans and Rieks Joosten from Pijnenburg, a silicon design company in Vught. They helped me make the decision to change the design to the 'deflate' format.

3.5 Planned Design Trajectory

The initial planning was based on previous work. This planning was included in the research proposal for this project. Some of the assumptions made at early stages of the project proved to be false, which lead to adjustments of plans. At various stages of the project new ideas emerged which lead to additional plans.

The original research proposal for this design project already contained a design planning.

> The project must be finished in 2–3 years. Several design steps must be performed:
>
> - Analysis of the problem and high-level architectural decisions. Part of this has already been performed in [Ben93]. Completion will take 3 months.
> - Precise analysis of several existing compression methods, prefix coding characteristics and required window size. Decision which methods are within reach (3 months).
> - Writing a state of the art report (2 months).
> - Detailed architectural model, chip packaging, internal channel protocols, external interfaces, general hardware testability (3 months).
> - Design of the prefix encoder and decoder in hardware (4 months).
> - Design of the string match/substitute array (4 months).
> - Design of other hardware modules (2 months).
> - Design of CPU hardware configuration (existing CPU architecture, 1 month).
> - Software engineering for one or more existing compression methods (4 months).
> - System integration, simulation or fabrication, testing (4.5 months).
> - Evaluation and writing a report (2 months).

It was definitely planned to present working silicon as the result of this design or at least a lay-out that could go right to the silicon foundry. The planning was very optimistic to say the least.

Further it was planned to use the same design methodology as described in [Ben93]. This method would be usable to create the detailed architecture (to be finished in 11 months, in reality 19 months). From there is was planned to design most modules using IDaSS, VHDL and the Compass tools.

It was planned to make a simulatable VHDL model of the entire device including a simulation of its environment. Parts could be translated from the IDaSS models and other parts would be written in non-synthesizable VHDL. As said before this was never realized. This was primarily caused by the fact that not all parts that should be implemented in IDaSS were actually implemented in IDaSS and some parts that were, could not be translated into VHDL.

The original research proposal did not mention the formation of a design team and it was not planned to delegate parts of the design to other persons. During the course of the project it was definitely planned to:

- seek expertise from industry to aid in the design of those parts of the chip that could not be implemented with a silicon compiler.
- delegate the design of some modules to graduate students.

The design of two of the modules (bus interface and embedded processor) was delegated to two graduate students, who produced a working IDaSS implementation. Help from the industry was also sought but not obtained in time.

At a very early stage in the design process it was planned to include dynamic Huffman coding as well as static Huffman coding into the design. This was dropped later, but a heavily parallelized dynamic Huffman algorithm based on associative arrays had been devised. See appendix A.3 (dynamic Huffman coding).

Chapter 4

Specification

The specification of any artifact is based on its requirements. Requirements often contradict each other. Usually the requirement that the artifact can be fabricated at reasonable cost conflicts with the requirement for high performance.

For many non-trivial design projects the specification changes during the course of the design. Problems arise that preclude the implementation of a specified feature or a changed design decision opens up new possibilities so that previously unspecified but highly desired features can be implemented.

This chapter describes how and why the specifications were made the way they are. Detailed specifications can be found in the appendices.

4.1 Compression Formats

The device compresses and decompresses data using the LZH method. Other compression methods than LZH (or either sliding window or Huffman alone) cannot be supported. There are many different LZH-based compression formats. There are a large number of different encodings for the lengths, literals and addresses from the sliding window coder. As the work load for the Huffman coding is too high for a general-purpose processor, this task must be performed by hardware. It seems impractical to make a hardware device that accommodates all existing and future encodings. Therefore only a limited number of basic encoding formats can be supported.

4.1.1 The Initial Format

No existing LZH based formats are formally standardized. The de facto standard `gzip` needs a sliding window of 32kB and initially it could not be implemented. The formats that have a sliding window of no more than 8kB of which specifications or source code are available are `PKZIP` implode and LHA. Accommodating two totally different encoding schemes would add considerably to the hardware complexity, so only one basic encoding format was specified. As the LHA format was used in more than one program (and offers a better compression ratio), this was chosen. Later architectural changes made it possible to implement the `gzip` format and the specification was changed to that format. The original design however, was based on LHA.

The requirement that data compressed with the device can be decompressed with readily available and portable software can be satisfied by the program `dcompr`, which is written in ANSI C. It is not really necessary to make the format compatible with an existing program, as long as `dcompr` can be made available.

One of the goals of the design project is an optimal compression ratio. If a slightly different format would yield a considerably better compression ratio than LHA, we would go for it instead. Therefore we decided to design a research format with many different parameters, which is a superset of LHA. Some settings of the parameters yield a format that is compatible with LHA. This format is implemented by the program `dcompr`. For details of the format see appendix B.1. The research is described in appendix C (comparison of compression methods). The main conclusion is that the sliding window size and the (non-)greedy matching method are of dominant importance. Huffman coding gives a considerably better compression ratio than no Huffman coding. Other parameters, such as the code buffer size, the maximum string length and the exact coding method are of minor importance. Formats with similar parameters yield similar results.

The following parameters were selected for inclusion in the hardware device:

- A sliding window size from 1 to 8192 bytes.
- Self matching possible, non-greedy matching and a minimum string length of 3.
- A maximum string length of 1 to 256.
- Sliding window coder output is divided into blocks of up to 8192 literals of length, address pairs.
- LHA compatible Huffman codes for literals, lengths and addresses. Each block has its own set of Huffman trees.
- Alternative method that uses one set of Huffman trees for the entire file.
- Fixed (non-Huffman) alternative codes for literals, lengths and addresses. End-of-section codes (see section 4.2.3) are possible with this method.

The program `dcompr` supported additional format options that were excluded from the hardware device:

- A sliding window size up to 32768. This is too large for hardware.
- A minimum string size of 2, no self matching and greedy matching. These options yield a worse compression ratio and add to complexity.
- Dynamic Huffman coding (end of section codes possible).

- An alternative Huffman code for addresses. This yields no better compression, is not standard and would add to the complexity.

Dynamic Huffman coding would yield a compression ratio comparable to static Huffman coding. The main advantage is that the latency between input and output is considerably less than for static Huffman coding. For static Huffman coding a large block is compressed with the sliding window coding and stored in a buffer. Next a Huffman code is derived from the frequency statistics. Next a description of that code is transmitted with the compressed data. Finally the buffered block of data can be Huffman encoded. These activities can be carried out in a pipelined fashion, so the throughput is not harmed, but the latency (the time interval between the reading of the uncompressed data and the writing of the compressed data) is very long. Dynamic Huffman coding would eliminate this latency. As opposed to static Huffman coding the concept of end-of-section commands can be meaningfully used with dynamic Huffman coding.

The complexity of a dynamic Huffman coder that satisfied the speed requirements would be prohibitive. Therefore dynamic Huffman coding was not implemented.

Especially for cases when low latency and the use of end-of-section commands is desired, a non-Huffman format is implemented.

The format with one set of Huffman trees per file needs to buffer the first block of data, after which it compresses all the rest of the file without buffering. At the start of the compression session it has a high latency, which drops considerably during the course of a compression session. If Huffman tree computations were a bottle neck this format would largely eliminate this bottle neck. This was one of the reasons to include this format.

Another variation was proposed in which there were two separate block sizes. Static Huffman coding was used with one block size B_1, but the statistics for the Huffman coding were based on only the first part of the block with a smaller size B_2. This would reduce the latency without increasing the work load for Huffman tree computations. It would be a mix between 'blocking' and 'streaming' behaviour.

However, it was impossible to implement this in a way compatible with the LHA format. In the LHA format each block starts with a count of the lengths and literals contained in it. At the time the block would be written, this count would be unavailable (it is not simply the block size B_1 as the end of file can come first). Therefore this option was dropped from the specification.

A workaround would be possible if the block size was set to the number of lengths and literals guaranteed to be available at the time the count had to be written, but not higher than B_1. This would make the functional behaviour of the compressor dependent on the relative timing of its parts and on the response time of the host system.

4.1.2 The `gzip` Format.

An architectural change made it possible to use a much larger sliding window for decompression than for compression. Therefore we could stay completely compatible with the 'deflate' format used by `gzip`, even though the compressor would be unable to use a 32kB sliding window. As 'deflate' would be a much more desirable standard format it was decided to implement that instead of LHA.

Extensions to GZIP format were defined, such as a window size of 64kB, dynamic Huffman coding and end-of-section codes. The program `dcompr` was changed into `gdcompr` to test all options of the extended 'deflate' format. This way we could repeat the experiments already performed with the LHA format.

Most of the 'deflate' extensions are not implemented in the device. The device could have the ability to decompress 64kB deflate and it could use 'end-of-section' codes in the non-Huffman format, for the rest it would just compress or decompress standard deflate. When compressing it would not utilize all features of `gzip`.

- Due to the general architecture it would be very hard to generate 'stored' (uncompressed) blocks. These are usually generated if `gzip` is used to compress uncompressible data. If stored blocks cannot be generated, the data is expanded by 1% or less.
- The sliding window size for compression is restricted to 8192 bytes.

4.1.3 Desirable Simple Enhancements.

As specified the device compresses and decompresses just the raw 'deflate' data as specified in rfc1951 [Deu96a]. In practice this format is either used as part of `gzip` or as part of the `zlib` library format. It is also used within the ZIP archive format.

To implement `gzip` (as specified in rfc1952 [Deu96b], it would be desirable to add the following features to the device:

- Each gzip file contains a variable length header at the start, hence the compressed data is not necessarily word-aligned. It would be desirable if the device could be told to skip a certain number of bytes at the start. The host CPU should generate or analyze the header. When compressing the header area would be overwritten with zero bytes and the host system should place the header there after the device wrote the first block.
- Each `gzip` file contains a 32-bit CRC of the original data and the original size at the end. The device should be able to generate and check the CRC. The size could as well be written by the host CPU. A CRC generator that computes 2 or 4 bits per clock cycle is comparatively easy to implement.

To facilitate 'zlib' compression (as specified in rfc1950 [DG96]), it would be desirable to add Adler-32 checksum capability to the device. For each byte this would take 2 16-bit additions modulo 65521. This could be implemented using 2 registers, one 16-bit adder and a bit of control logic.

4.2 Interfacing

This section describes the specifications of the interfacing of the compression device to the environment in which it operates. For details see section E.1 (data sheet). The device acts as an intelligent coprocessor within a computer system (the host system). It performs the complete compression or decompression task with minimal attention of the CPU. The CPU must be able to run another task while compression or decompression is in progress.

Figure 4.1 shows the compression device in the host system. It uses interrupts and I/O ports to communicate with the CPU and DMA to transfer its data.

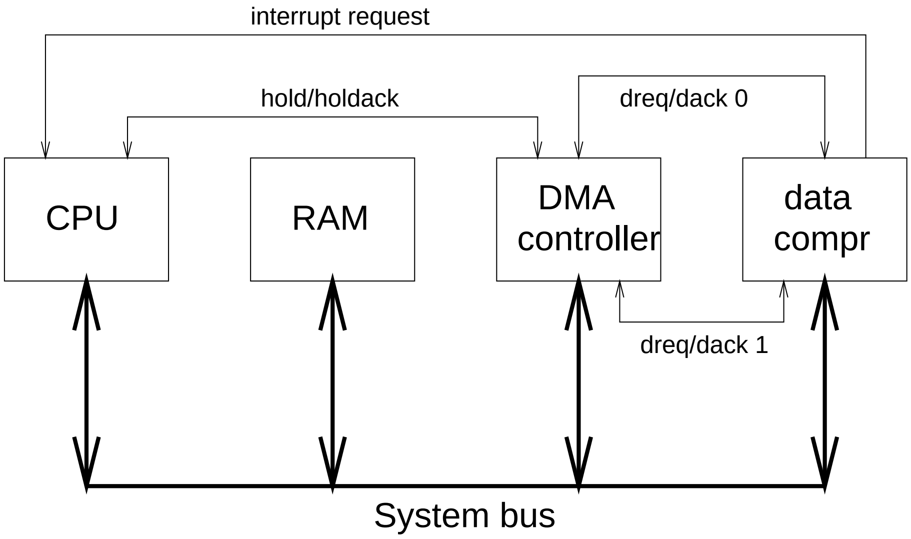

Figure 4.1: The Compressor in the Host System.

4.2.1 The Host System

The host system contains a CPU, RAM and I/O devices, all connected by a bus. The bus consists of a data bus with a width of 8, 16 or 32 bits, an address bus with a width of 16–32 bits and several control signals. I/O devices are accessed through I/O ports. I/O ports have addresses and are accessed in a way similar to memory, either using dedicated I/O control signals or using the same control signals as the RAM. I/O devices can send interrupts to the CPU. When the CPU receives an interrupt it interrupts its running program and then it executes a special program, the interrupt handler, to communicate with the I/O device. After finishing the interrupt handler the CPU resumes its original program.

Under normal circumstances the CPU is the only device that controls the bus. It drives the address bus and the read/write control signals. The DMA (direct memory access) controller can temporarily take control of the bus after requesting it from the CPU. Typically the DMA controller requests control for one read or write cycle. The

DMA controller can transfer large amounts of data between an I/O device and memory without CPU interaction.

A device that wants to write data into memory or read data from memory, activates the DREQ (DMA request) line of the DMA controller. Next the DMA controller requests control of the CPU (HOLD/HOLDACK handshake) and activates the DACK (DMA acknowledge) line of the requesting device. If a memory read request was made, the DMA controller addresses a memory location for reading and when the data appears on the data bus it activates the IO WRITE line. At that time the I/O device can read the data from the bus. If a memory write request was made, the DMA controller activates IO READ, so that the I/O device can place data on the bus. Next it addresses a memory location for writing so that the data from the bus are written to memory. The memory address is stored in a register of the DMA controller and is automatically incremented for each transfer. As described above the DMA controller uses one bus cycle per DMA transfer. Some DMA controllers use two bus cycles per transfer, one to access memory and one to access the I/O device. Most DMA controllers have more than one DMA channel, each with its own pair of DREQ/DACK lines and its own address registers and counters.

4.2.2 Data Transfer

The compression device processes large amounts of data. Both the input and the output data are stored in memory. As the CPU must be able to do something else while the device processes the data, DMA is the only way to transfer the data to and from the device. Input data and output data are at independent memory locations and therefore two DMA channels are needed.

Both the input data and the output data for the device are arbitrary sequences of bytes. As the sequences can be very long they must be divided into smaller blocks. Each block is transferred by a single DMA session. Blocks can have arbitrary lengths from 1 to a certain maximum.

As the device must be usable in a wide range of different computer systems the DMA transfer width has to be configurable for 8, 16 and 32 bit wide transfers. The device works with bytes internally, so a single DMA transfer can carry 1, 2 or 4 bytes. As a data block can contain an arbitrary number of bytes, the final DMA transfer of a data block may carry bytes that are not really part of the block. E.g. if the DMA transfer width is 32 bits and the block size is 17 bytes, 5 DMA transfers are needed, the first 4 carry 4 data bytes each and the last one contains 1 data byte and 3 bytes that have to be ignored.

The desire to specify the block size in bytes, even in systems with wider DMA channels, requires the following provisions in the compression device:

- There must be a means to specify the exact number of bytes in a data block, especially an input data block. The number of DMA transfers can be specified using the counter registers in the DMA controller, but this does not specify the exact number of bytes. A 2-bit register in the compressor is needed to make this possible. The DMA controller outputs an EOP (end of process) signal to signal the last transmission in a block, which can be made available to the compressor, so in principle the 2-bit register in the compressor and the counter register in the DMA controller are sufficient to specify the exact number of bytes in a block.

- When a job (or part of it) is finished the input data block need not be all read and the output data block need not be all written. The application that uses the compressor usually needs to know the exact number of bytes that the compressor has read or written. In most cases the counters in the DMA controller can be read, but they do not specify the number of bytes. Hence additional 2-bit counters are needed in the compressor.

To allow for DMA controllers without readable counters or a usable EOP signal we decided to add registers for the full block size to the compressor. 16-bit wide registers were chosen to allow for a maximum block size of 64kB. The following four registers are made accessible from the CPU:

- Input block size in bytes (writable).
- Output block size in bytes (writable).
- Number of bytes read from input block (readable).
- Number of bytes written to output block (readable).

4.2.3 Communication with Host CPU

The compression device is controlled by the host CPU. Commands from the CPU initialize the device and once compression or decompression is started, the device communicates with the host CPU by means of interrupts and I/O registers.

The compressor contains I/O registers that are accessible as I/O ports from the CPU in the host system. These registers are 8 bits wide to make the device usable in 8-bit systems. For ease of use we decided not to define write-only registers at the same I/O address as read-only registers. The following functions must be available through the I/O registers:

- Commands from the host CPU to the device.
- The status of the device.
- Setting of the interface mode: the DMA transfer width, enabling of the IRQ and the output block behaviour with end-of-section commands.
- Error codes that the host CPU can read.

- The four block size registers, each split in two 8-bit parts, making 8 I/O registers.
- Several selectable parameters of the compression method.

Registers that deal specifically with compression are kept separate from the registers that deal with interfacing and data transfer. This makes it easier to use the same bus interface unit in a different intelligent coprocessor with different core functionality. Therefore there will be two different error code registers.

The selectable parameters belong to the core processor. They are made indirectly accessible though a 'parameter register number' and a 'parameter register content' register. In this way it is possible to define 256 parameter registers without the need for many address lines.

All I/O register functions can be implemented in 15 registers.

The following commands can be given to the compressor:

- Abort the job currently running.
- Commands related to input data stream (continue with next block, end of section, end of data).
- Commands related to output data stream (continue with next block, end of section acknowledge).
- General commands. The following commands are defined:
  - Reset the compression device to a known state.
  - Run a self test.
  - Reprogram the parameter registers. We defined a separate command for setting the parameter registers to make it possible to check their validity prior to compression or decompression.
  - Start compression.
  - Start decompression.

These can be combined into a single 8-bit command register.

The following status information must be available:

- The current job is aborted.
- Input block status (normal, waiting for next block, end of section, end of data).
- Output block status (normal, waiting for next block, end of section, end of data).
- General status (ready for new command, initializing, testing, reprogramming, compressing, decompressing).

This status information can be combined into a single 8-bit register.

The device uses a single interrupt. An interrupt may be caused by more than one condition. A single interrupt may be caused by more conditions at once. The conditions that cause the interrupt can be found out by querying one status register. There is a single IRQ line that is deactivated as soon as the status register is read by the CPU. The device sends interrupts in the following cases:

- The input data block is all read. The device expects a new data block or a signal indicating the end of the data.
- The output data block is all written and more output data is available. The device expects a new output data block.
- The job is finished.
- An error occurred.

After the device is reset, tested and reprogrammed, any number of compression and decompression commands can be given. The device can only be active with either the compression or the decompression of a single data stream. The device cannot simultaneously compress and decompress neither can it simultaneously be active with two independent data streams.

A compression or decompression session is started with the 'compress' or 'decompress' command. It can end for the following reasons:

- The CPU sends an abort command to the device.
- The device encounters an error and aborts the operation.
- The CPU sends an end-of-input command to the device. After all data is processed the device terminates the operation normally.

In all three cases the device notifies the CPU with an interrupt.

In some situations the data stream for the compressor consists of short blocks with a long time between them (e.g. commands issued from a terminal). It is important that each command is processed in its entirety as soon as it is received by the compressor. If the current input data block is completely read, the compressor sends an interrupt. If the CPU gives no command for the input data, the compressor will just wait. The CPU cannot give a continue command because no new data are available. The CPU can give an end-of-section command on the input block. Next the compressor compresses any data that it has read and outputs the compressed data. A special end-of-section code will be embedded in the compressed data. The compressor will send an IRQ to notify the CPU that the compressed data are ready. Next the CPU acknowledges the end-of-section condition on the output block and processes the output data. If new input data becomes available the CPU can send a 'continue with next input block' command to the device

When decompressing the device detects the special end-of-section code in the compressed data and as soon as all preceding data is decompressed and output the compressor sends an IRQ to notify the CPU that a section of data is available.

There are two ways to treat the output block across an end-of-section condition:

- Continue to write more data into the same output block. If 16 or 32 bit DMA is used, the subsequent output data must be aligned to a multiple of 2 or 4 bytes. For compression the format of the special end-of-section code can be chosen to accommodate this, for decompression this is not generally possible, unless the uncompressed data itself is always aligned across section boundaries.
- Start a new output block. This is the preferred solution in most cases.

Special care must be taken to account for zero-sized input or output blocks or zero-sized sections. This means in particular that during decompression of possibly zero-sized files a 'continue with next output block' command shall never be given unless the devices explicitly requests this. This is true because in the current specification to 'output bytes written' register indicates a range from 1 to 65536, not including zero. Hence the case of zero output bytes must be indicated by the absence of a request for the output block.

4.2.4 Hardware Connections

For its communication with the rest of the system the device needs the following connections:

- 32 data bus lines, tristate bidirectional. Only 8 are used for I/O port data. 8, 16 or 32 are used for DMA.
- 4 address lines plus one chip select line to address 16 I/O ports.
- IOread, IOwrite and Ready lines for the normal bus cycles. The Ready line is an 'open collector' output from the device to the CPU.
- Two sets of DREQ, DACK lines, one for each DMA channel.
- An IRQ (interrupt request) output.

The semantics of these signals were made to be compatible with the popular ISA bus. With a bit of external logic the device can be made compatible with many other bus specifications. Alternatively one can use strappable inputs to change the polarity of certain bus signals such as `IOREAD` and `DREQ`. These inputs are directly soldered to either logical 0 or logical 1 and make the bus signals on the IC compatible with a particular bus standard. They cost extra pins, but eliminate need for external components such as inverters. They are not specified but if enough pins are available they can be added in a very late stage of the design process.

Further the device needs the usual power, ground, clock and reset inputs. If the device consists of more than one chip or if it needs additional memory chips then it needs extra connections to interface them.

The device requires a single clock of 50MHz to process 12.5 million uncompressed bytes per second. It can operate at lower clock frequencies, but it will have a proportionally lower throughput. This and the selectable width for DMA transfers makes the device usable in a very large number of computer systems. The device can only work at its full speed if the computer system offers 32-bit DMA transfers and if a clock of 50MHz is available. If a lower clock frequency is available on the bus of the host system a frequency multiplier can be used to provide a reasonable clock input that is synchronized with respect to the bus,

We did not specify a dedicated port for compressed data that would be directly connected to a peripheral. This would save bandwidth on the system bus. Such a port would be useless without a device to connect to it. Furthermore the compression chip would mostly be tied to a single peripheral.

4.2.5 Changes to the Interfacing

Because of the long latency with static Huffman coding the throughput would be low if small data sequences (files) were compressed. It would be desirable to process several independent files in a single compression session. The end-of-section mechanism already defined, would be usable to implement this.

Further it was decided to add an option to fill the compressed data to the nearest multiple of a specified power of two. This would be very useful if the compressed output was directly written to or read from disk sectors.

Chapter 5

Design Process

Most existing software engineering methods specify a number of stages in the design process:

- Requirements analysis.
- Specification.
- Behavioural (or conceptual) model.
- Architecture.
- Coding.
- Testing.
- Maintenance.

Complex digital information systems are often designed using methods that are derived from software engineering methods. Some software engineering methods already emphasize real time systems or mixed hardware/software systems, such as the method by Ward and Mellor[WM85] and the similar method by Hatley and Pirbhai [HP87]. In a previous thesis [Ben93] I described a methodology that specifies the architectural modelling for a system designed with the method designed by Ward and Mellor and that will consist largely of hardware.

Whatever design method is chosen, any systematic design process utilizes at least the following types of models:

- A behavioural model describes the behaviour of the system: what information goes into it, what information comes out of it, what type of information must be stored and what processes act on the information.
- An architectural model describes the physical hardware or software parts the system consists of. It specifies interfaces between the parts and the representation of the information exchanged by those parts.

Associated with each design stage there is usually a document or set of documents that describes the result of the design stage. There is one documents that describes the requirements, one that describes the specification, one that describes the behavioural model and one that describes the architectural model. Of course there are also documents describing the implementation (coding), the tests and all bug fixes and revisions of the design. Prior to implementation (coding) the work of each stage is based on the result of the previous stage. Implementation is based on architecture, architecture is based on the behavioural model, which is in turn based on the specification, which is based on the requirements.

In most practical design situations the result of each of the design stages does not remain constant but is influenced by the following design stages. In other words there is feedback between the design stages. Difficulties encountered during implementation may necessitate a change in the architecture. When the architecture is defined it may be necessary to change the behavioural model, the specification or even the requirements.

Complex systems often consist of several subsystems. Once the architecture of the whole system is defined, the specification of each of the subsystems becomes clear. Once the interfacing between the subsystems is established, it becomes possible to treat the design of each of the subsystems as a completely new design task. Subsystems may in turn contain other subsystems, thus a hierarchy of design problems exists. The design methodology described in [Ben93] especially promotes hierarchical design. However, the specifications of subsystems are seldom so independent and definitive that they will never be changed during the further course of the design process. Hence a change in the specification of one subsystem may necessitate a change in another subsystem.

During the design process a designer encounters numerous situations where more than one solution exists for a given problem. One task can be implemented by several algorithms or one algorithm can be implemented in several ways. One has to select one of the alternatives. This selection has to be based on certain objectives such as speed, cost or generality. One must evaluate each of the alternatives.

Once a design is made, it should be possible to use it as the basis for the design of similar systems. It must be possible to reuse parts of the design in different systems. It must also be possible to scale the design. If a data compressor for sliding window coding is designed with a buffer size of 8192 it should be possible to use the same basic design for other buffer sizes as well. The extent to which scalability can be applied should be indicated as well (e.g. buffer sizes can range from 1024 to 65536). It should be clear what has to be changed when a different compression format is needed.

5.1 The Behavioural Model

During the early phases of the design the functions of the device must be analyzed. The primary tasks of the device are text compression and decompression. The functions related to interfacing are only needed to serve the main processing tasks.

Hence the system to be designed can be characterized as a processing oriented system rather than a real-time system. According to the specification there are no hard real-time constraints. A data throughput of 12.5MB/s is specified and the device has to satisfy the low-level bus-timing requirements of the bus system it is connected to, but otherwise there are no hard limits on response time. Even excessive delays in the compression algorithm cause the device to perform below its specified rate, but not to fail or produce incorrect results.

The emphasis of the design process was on the compression algorithms and their efficient implementation. Therefore the initial behavioural modelling concentrated on the functional behaviour rather than temporal behaviour. Compression methods were primarily viewed as (mathematical) functions from byte sequences to byte sequences (see appendix A, compression algorithms).

5.1.1 Interfacing and Processing Tasks

One of the first divisions of the functionality was between bus interfacing and core processing.

- Bus interfacing comprises of the following tasks:
  - getting the input data from memory to the device through DMA.
  - getting the output data from the device to memory through DMA.
  - communicating with the host CPU through I/O registers and interrupts.
  - Starting and stopping compression and decompression sessions.
  - Managing input blocks, output blocks and end-of-section conditions.
- Core processing comprises the tasks of data compression and decompression.

The bus interfacing and core processing tasks exchange the following information:

- A finite-length sequential stream of input bytes from the bus interface to the core processing task. It may be divided into sections.
- A finite-length sequential stream of output bytes from the core processing task to the bus interface. It may be divided into sections.
- A finite set of compression parameters, written by the bus interface (on behalf of the host CPU) and read by the core processing task.
- A command from the bus interface (usually on behalf of the host CPU) to the core processing task. Each compression or decompression command is logically associated with an input stream, an output stream and a set of compression parameters. A session can be aborted both by the core processing task and by the bus interface (usually on behalf of the host CPU).
- A status from the core processing task to the bus interface.

Even the earliest preliminary architecture diagrams in [Ben93] pages 171 and 173 had the bus interface as a separate unit that supplied input bytes to the relevant data processing module and accepted output bytes from the relevant data processing module. The byte channels were however drawn as bidirectional channels for uncompressed and compressed bytes respectively.

5.1.2 Behavioural Modelling of the Processing Task

The behavioural modelling of the compression and decompression tasks concentrated primarily on the functional behaviour. At the highest level this functional behaviour consists entirely of the implementation of the specified compression method.

Compression methods were first judged on the following criteria:

- Their being lossless for all possible uncompressed byte sequences. Methods that would not losslessly transfer certain classes of byte sequences to compressed sequences and back to the original, were not considered. It was not required (in fact logically impossible) that the methods would actually *compress* all byte sequences without exception.
- Their compression ratio for a certain set of test files.
- Their compatibility with existing compression methods.
- Their estimated complexity to implement.

At a very early stage in the design stage we decided to implement an LZH based method. Later we decided to analyze the LHA format and define some extensions to it. In appendix C (comparison of compression methods) we compare the compression ratios of several variations of the LZH compression method.

A compression method uses several algorithms and these algorithms can be implemented in several ways. Algorithms are described in appendix A. Algorithms can be either executed by software on processors or by hardware that is specially designed to execute the algorithm.

The function of the compression process can be decomposed as follows:

- The sliding window coding process performs the sliding window compressing methods on a stream of bytes and converts it into a stream of literal bytes and substring descriptors, each consisting of a length and an address.
- The Huffman coding process takes the stream of literals, lengths and addresses from the sliding window process and converts it to a stream of compressed bytes.

For decompression the Huffman decoding process and the sliding window decoding process are applied to the data in reverse order.

Both static and dynamic Huffman coding were considered.

Static Huffman compression comprises the following subtasks:

- Buffering of a blocks of symbols.
- Collecting of frequency statistics of the symbols.
- Computing the Huffman trees from the frequency statistics.
- Transmitting a description of the trees with the compressed data.
- Encoding the symbols with the Huffman trees.
- Converting the encoded symbols to 8-bit bytes.

In LZH coding the symbols to be encoded are literal bytes, lengths and addresses. Two Huffman codes are usually employed, one for the lengths and literals and the other one for the addresses. A functional diagram of the LZH compressor with static Huffman coding is shown in figure 5.1. Apart from the tasks of Huffman coding and sliding window coding there is the bus interfacing task and a control task to coordinate all other tasks.

The following data items exist in the LZH compression process with static Huffman coding:

- A stream of uncompressed bytes that is processed by the sliding window coding process.
- A stream of lengths, literals and addresses from the sliding window coding process to the buffering process. The buffering process divides the stream into blocks. The same stream also goes to the statistics collecting process.
- The frequency statistics for lengths, literals and addresses. For each block of lengths, literals and addresses one set of statistics is produced.
- The Huffman trees produced by the tree computation process. For each set of statistics one set of trees is produced. The trees go to the Huffman coding process and to the tree transmission process and they may have different representations.
- A stream of lengths, literals and addresses from the buffering process to the Huffman coding process. The stream is divided into blocks.
- A stream of variable length bit fields from the Huffman coding process to the byte conversion process. The tree transmission process produces a similar stream.
- A stream of compressed bytes from the byte conversion process.

Static Huffman decompression comprises the following tasks:

- Extracting variable-length bit fields out of the compressed byte stream.
- Decoding the Huffman tree descriptions received along with the compressed data.

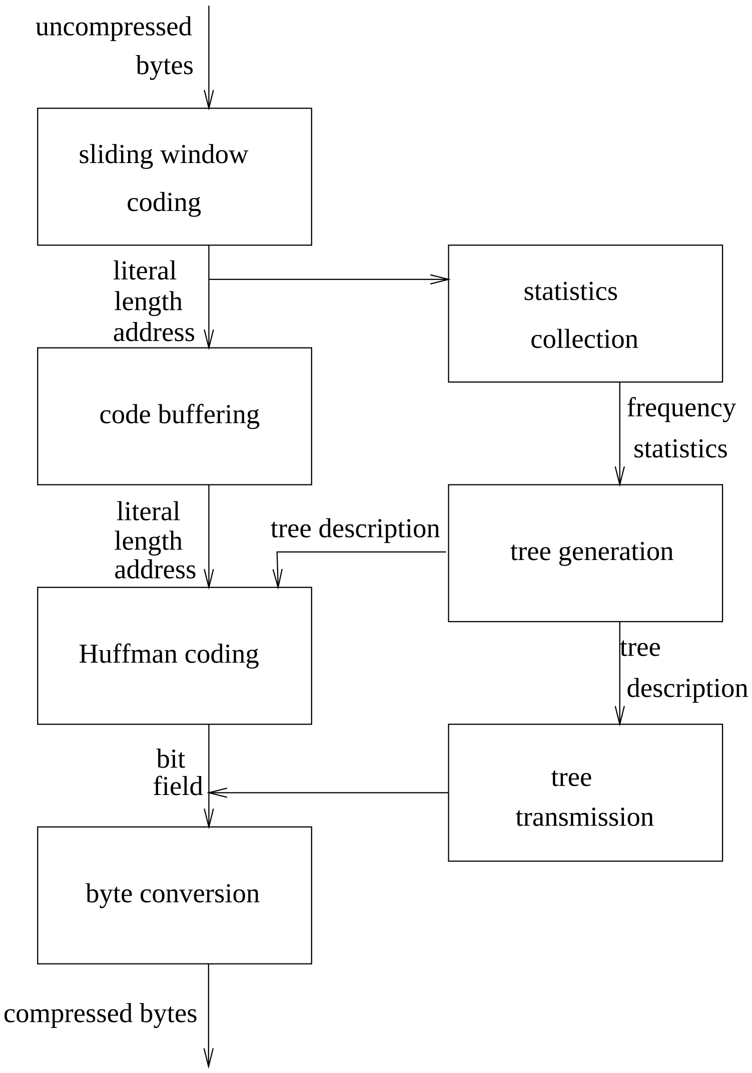

Figure 5.1: Functional Diagram of Compressor with Static Huffman Coding.

- Generating the Huffman trees for the Huffman decoding process.
- Decoding the compressed data with the Huffman trees.

Buffering, statistics collection and computing the tree from statistics are not needed when decompressing. A functional diagram of the LZH decompressor with static Huffman coding is shown in figure 5.2. The decompression data flows are shown as dashed lines so they are distinguishable when we combine the two diagrams.

The following data items exist in the LZH decompression process with static Huffman coding:

- A stream of compressed bytes to the byte conversion process.

- A stream of variable length bit fields from the byte conversion process to either the Huffman decoding process or the tree receiving process. The flow of information is not strictly unidirectional as the receiving process determines the number of bits in each field.
- The set of trees as decoded by the tree receiving process.
- A stream of length, literals and addresses from the Huffman decoding process to the sliding window decoding process.
- A stream of uncompressed bytes from the sliding window decoding process.

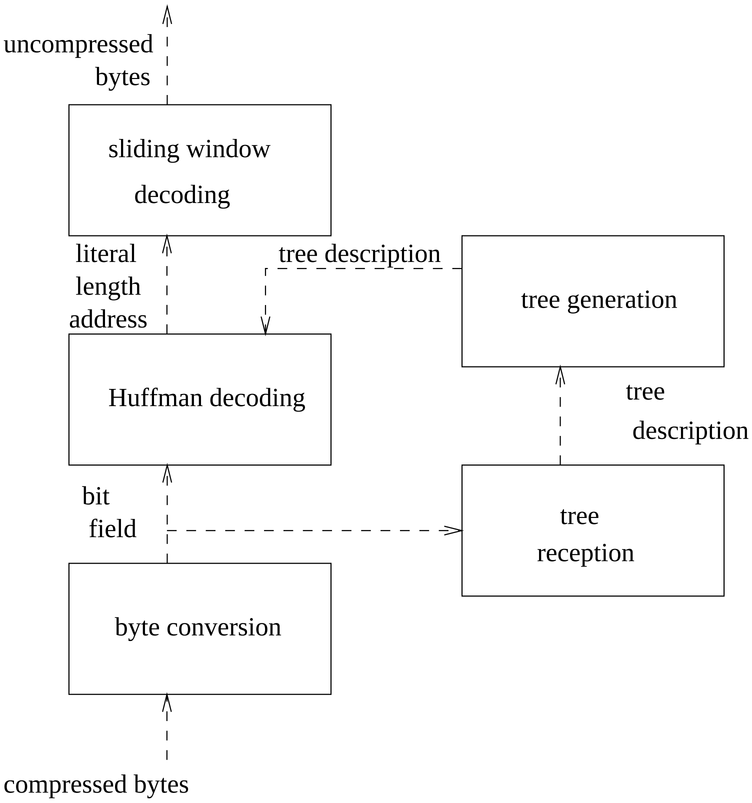

Figure 5.2: Functional Diagram of Decompressor with Static Huffman Coding.

For dynamic Huffman coding the static Huffman coding or decoding process in the diagram would be replaced by a dynamic Huffman coding process. No code buffering or tree transmission is needed. Statistics collection and tree updating are integrated into the dynamic Huffman coding process. In case of count overflow a static tree computation may be needed as well. Dynamic Huffman decompression has to maintain the same frequency statistics and perform the same tree computations as the compression task.

The diagrams in figures 5.1 and 5.2 are very similar to data flow diagrams as used by the method of Ward and Mellor [WM85]. Together with a detailed description of the tasks and a description of all data they would form a behavioural model. In [Ben93] there are actual Ward and Mellor data flow diagrams of an LZH process in which only lengths and literals are Huffman coded.

The program `dcompr` (see appendix C, comparison of methods) served as an executable behavioural model for the processing part of the design. It would generate the exact same (not merely compatible [1]) compressed data as the hardware device would. Like the core processing task defined in the previous section it exchanges the same information with its surroundings.

- The input and output byte stream are read from or written to files. Input section boundaries can be indicated on the command line. Output section boundaries are shown on the screen.
- Commands and parameters are passed on the command line.
- Status information is shown on the screen.

The algorithms to implement the same compression method in `dcompr` are different from those likely to be used in the hardware. the functions like sliding window (de)-coding, Huffman (de)coding, buffering and statistics collection are recognizable in the C program and the same information is exchanged between them. Of course the C program is executed sequentially whereas the functions of the hardware device would be executed in parallel.

5.1.3 Temporal Behaviour

The externally observable temporal behaviour of the device mostly comprises the communication with the host system as specified in appendix E.1 (data sheet).

Some of the internal temporal behaviour of the device will also be externally visible, such as the latency between input and output, the rate at which input data is consumed or output data is produced and any bursts in these data rates.

The tasks run in parallel as much as possible. The bus interface performs DMA accesses while input bytes are read by the core processing task and output bytes are produced.

The processes of sliding window coding, Huffman tree computation, tree transmission and Huffman coding work together as a high-level pipeline at the level of static Huffman blocks. This is shown in figure 5.3. Arrows show the dependencies between the processes. While one block is Huffman encoded, the Huffman tree for the next one is being computed and the block after the next one is sliding window coded. The block

[1] One can compress the same uncompressed data stream to different compressed data streams, each of which conforms to the compression format and can be decompressed to the same original

being Huffman coded is read out of the buffer, the block whose Huffman tree is computed is stored in the buffer (but not accessed) and the block that is sliding window coded is written into the buffer. Tree transmission cannot run concurrently with Huffman coding as the tree description may not be mixed with the Huffman compressed data in the compressed byte stream. It could run in parallel with tree computation if the pipeline had yet another stage. It was decided not to add that stage.

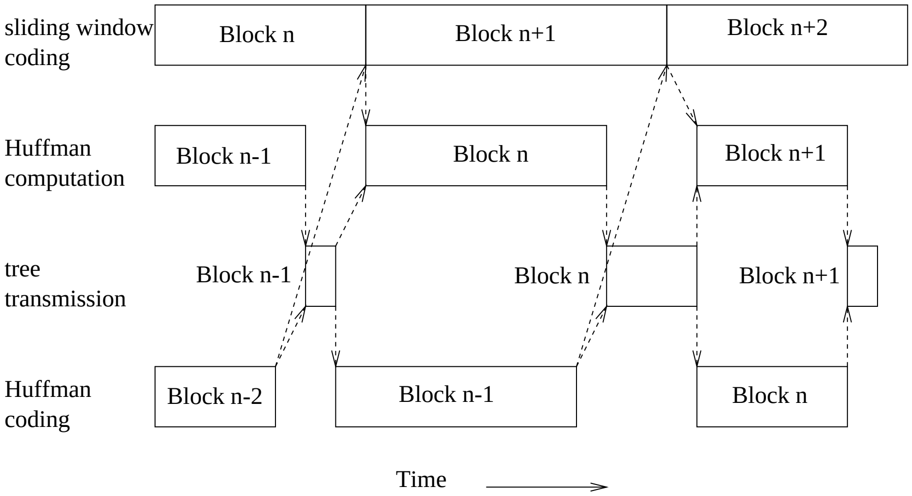

Figure 5.3: High-Level Pipeline of LZH Compression.

The buffer must be capable of holding at least three blocks. The first and the last block need not reside completely in memory, but space for at least two blocks is necessary.

As shown in the figure the processes in the pipeline have variable durations (which are data dependent). Of course all tasks must be synchronized with respect to each other. The following types of synchronization are required:

- Synchronization between the device and the host system. This exists at a very low level in the form of DMA cycles and bus cycles to access I/O ports. Whenever an input block is all read or an output block is all written, the device sends an interrupt to the host CPU. It expects a command in return. This is synchronization at the level of DMA blocks. Only the bus interfacing task takes part in synchronization with the host system. It is completely invisible to the core processing task.

- Synchronization at the level of individual data items. Individual data items are bytes and lengths, literals or addresses. No process should behave erratically because another process produces data items too slow or too fast. A form of flow control or handshaking is needed.

- Synchronization at the level of static Huffman blocks if static Huffman coding is used. This is the synchronization that keeps the pipeline working. Only when a block is completely sliding window coded, its Huffman tree can be computed.

Only when that has been finished the tree can be transmitted and only when that is finished the block may be Huffman coded.

- Synchronization at the level of sections and sessions. When the device receives an end-of-section command, it has to take special actions. Special codes may have to be inserted in the compressed data and in the case of multi-file compression the sliding window coder has to be reset. When all data before the end-of-section is compressed, the bus interface must be notified. This type of synchronization extends to the external interface.

The synchronization is such that the correct functional behaviour is guaranteed regardless of the time taken by all tasks.

It is assumed that the sliding window coder is capable of compressing 12.5 million uncompressed bytes per second. As this would be the costliest part of the device its optimal utilization is emphasized. It is assumed that under normal circumstances no other processes pose a bottleneck. The speed of the other tasks must be matched as closely as possible to that of the sliding window coder to prevent underutilization of parts of the device or bottle necks.

If the size of static Huffman blocks is increased a larger buffer is needed. On the other hand the tasks of Huffman tree generation and tree computation have to be performed less frequently so their work load is reduced. Compression ratio is very slightly enhanced while the latency between input and output is increased.

When the device is decompressing there are only two stages in the 'high-level pipeline' In one stage the tree descriptions are received and decoded and the data is Huffman decoded. These tasks are fully dependent on each other so no overlap can occur. In the other stage the output of the Huffman decoder is decompressed by the sliding window decoder. If there were no buffering between the Huffman decoding and sliding window decoding processes the sliding window coding process would be idle during the time that tree descriptions are received and decoded. If the sliding window decoding process should be the performance limiting factor as the sliding window coding process is when compressing, then this is not desirable. Therefore buffering is specified.

5.2 Initial Architecture

In reality the behavioural modelling process in the previous chapter was not purely behavioural[2]. At the same time we had an architecture in mind. For some parts of the behaviour the architecture was already very clear, for other parts it was not.

The bus interface was viewed more as a hardware module than as an abstract process in a functional decomposition. Likewise the sliding window coder was viewed as a concrete module. The tree generation and transmission processes were not viewed as

[2]The behavioural model contains a structure that parallels a likely architecture. Elements of this structure are the bus interface and the sliding window coder.

(hardware) modules as it was not yet clear whether they would be implemented in hardware or software.

Interfaces between modules also varied in their level of abstractness. At one end of the spectrum the external bus interface and its protocol were fully established at the time of behavioural modelling. Similarly the streams for uncompressed and compressed bytes and the streams of lengths, literals and addresses were pictured as high-speed data channels. At the other end the interfaces to the tree generation and tree transmission processes were not at all concrete. The mechanism to implement the control signals was also undecided.

At a very early stage of the design process the bus interfacing process was viewed separately from the core processing task and similarly in the earliest architectures the bus interface was a separate module. The implementation of the bus interface module was started long before the final architecture was known.

It was also decided very early to use a half-duplex architecture. The device can either compress or decompress, but not both at the same time. Decompression is viewed as the reverse process of compression and this is reflected in the architecture. For the decompression process the hardware modules of the compression process are reused as much as possible.

Most modules have a task both when compressing and when decompressing. When compressing the main data flow runs from the sliding window coder through the code buffer through the Huffman coder to the byte converter. When decompressing the main data flow runs in the reverse direction. The channels for bytes and for lengths, literals and addresses are bidirectional.

Table 5.1 shows how the modules from the initial architecture are used in the compression and decompression processes. Figure 5.4 shows the initial architecture diagram. Solid lines show the data flows for compression, dashed lines show the data flows for decompression. Dotted lines show the control flow.

Module	Used during compression	used during decompression
Bus Interface	Yes	Yes
Sliding Window Coder	Yes	Yes
Statistics Collector	Yes	No
Code Buffer	Yes (essential)	Yes (performance)
Tree Generator	Yes	Yes (process received tree)
Tree Transmitter	Yes (transmit tree)	Yes (receive tree)
Huffman Coder	Yes	Yes
Byte Converter	Yes	Yes
Controller	Yes	Yes

Table 5.1: Module Usage in Initial Architecture

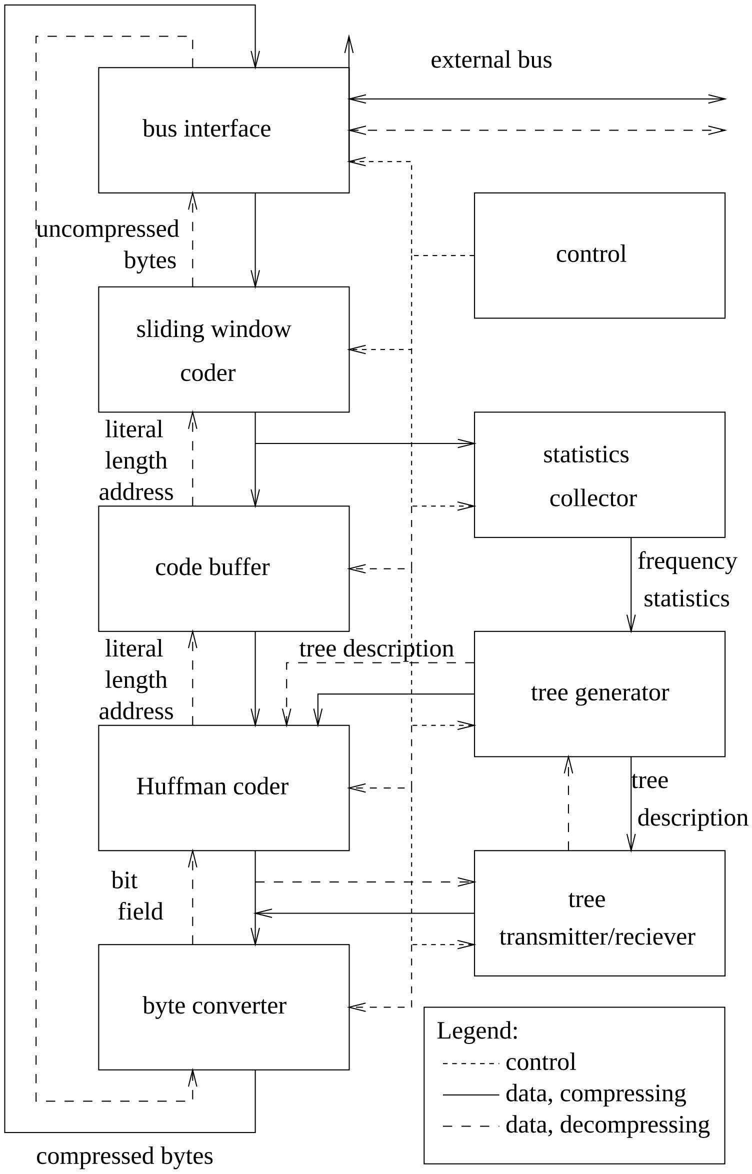

Figure 5.4: Initial Architecture Diagram

5.3 Selection of Algorithms and Implementation Methods

Selection of algorithms and implementation methods is an important step between behavioural modelling and the architecture.

We have to determine what the requirements of all tasks are, what algorithms are available and what implementation methods are suitable. Next we have to select the optimum algorithms and implementation methods. An important part of this task is the trade-off between hardware and software.

5.3.1 Requirements of Individual Tasks

The rate statistics for the various data streams are obtained by simulations described in appendix D. R_u is the uncompressed byte rate and it was defined as 12.5Mbyte/s. All other rates depend on R_u. R_c is the compressed byte rate and it is 6.25Mbyte/s (based on average compression ratio) . R_s is the rate at which substrings are produced. A substring is either a literal or a length/address pair. This rate is 4.2 million substrings per second (based on average substring length). R_b is the rate at which byte conversions are performed, which is 8.3 million per second. R_t is the rate at which static Huffman blocks are produced, which depends on the code buffer size. If the code buffer holds 8192 substrings, R_t is 500 per second. The LHA compression format is assumed. These rates are average rates.

Function	Processing	Communication	Storage
	(MIPS)	(Mbyte/s)	kilobyte
Bus Interface	20	20+12+6	0.04
Sliding Window	1000	12+4	8-10
Statistics	8	4	1
Buffering	8	4+4	32-64
Tree computation	10	2	6
Tree transmission	2	1	1
Huffman coding	8	4+8	2
Byte conversion	8	8+6	0.005
Control	0.01	0.001	0.05
Dynamic Huffman	200	4+8	8

Table 5.2: Requirements of Functions

Table 5.2 gives an overview of the requirements for communications bandwidth, processing power and storage of the processes. For processing power estimation a sequential algorithm is assumed (not necessarily executed on a general-purpose computer). Even though the figure is specified in MIPS, it does not mean that the task will be executed on a processor. The specified rates for communication bandwidth and processing power are average. Due to synchronization requirements at the level of static

Huffman blocks, the processes are not continuously active (see figure 5.3) and therefore the peak rates can be higher (estimated 30% higher for the Huffman coding and byte conversion processes). If any of the processes does not meet the specified rate, it is likely to become a bottle neck. The process requirements are as follows:

- The bus interface has to cope with an uncompressed byte rate of R_u (12.5Mbyte/s) and a compressed byte rate of R_c (6.25Mbyte/s). Both byte streams have to be transferred between the core processor and the host system memory. The required memory bandwidth totals to about 20Mbyte/s.

  Processing requirements consist of incrementing counters and converting bytes from or to DMA words. This is estimated at 20MIPS.

  Storage requirements are minimal: 15 I/O registers (including counters), 16 parameter registers plus some registers to convert memory words (16 or 32 bits) from and to 8-bit bytes.

- The sliding window coder has to cope with an uncompressed byte rate of R_u (12.5Mbyte/s) and a substring rate of R_s (4.2 million strings per second).

  If the task were performed in software, it would take about 1000MIPS.

  Decompression would take about 50MIPS.

  Storage requirements are 8kB for the sliding window itself plus a few kB for flag bits if a hardware algorithm is chosen. For a software implementation the memory requirement would be several times the sliding window size (32kB).

  For sliding window decompression 8kB would suffice.

- The buffering process reads one stream of substrings from the sliding window coder and sends one stream to the Huffman coder. Both streams are at a rate of R_s (4.2 million strings per second).

  Processing consists of storing and retrieving data in a buffer and incrementing of counters. This is estimated at 8MIPS.

  Storage requirements are 32-64kB if two or three blocks of literals, lengths and addresses, 8192 substrings each, must be stored.

- The statistics collecting process takes one stream of substrings at a rate of R_s (4.2 million strings per second).

  Processing consists of reading, incrementing and writing a counter for each length, literal or address. This is estimated at 8MIPS.

  Storage requirements amount to 526 words of 13 bits each, 512 to count lengths and literals, 14 to count address prefixes.

- The tree computation process would need to read the statistics and write the trees at every static Huffman block. This is estimated at 2Mbyte/s.

Processing is estimated at 10MIPS for a linear Huffman algorithm.

Storage is estimated at 6 kilobytes.

Considerably less would be needed for the task when decompressing.

- The tree transmission and receiving processes take a few hundred bit fields for each tree and a tree description of a few kilobyte. Total communication requirements are estimated at 1Mbyte/s.

  Processing requirements are estimated at 1MIPS.

  Storage requirements are estimated at 1kB.

- The Huffman coding process takes a stream of substrings at a rate of R_s (4.2 million strings per second) and a stream of bit fields at a rate of R_b (8.3 million fields per second).

  Processing requirements are estimated at 8MIPS (primarily table lookup).

  Storage requirements are estimated at 2kB.

- The byte conversion process takes a stream of bit fields at a rate of R_b (8.3 million fields per second) and a stream of compressed bytes at a rate of R_c (6.25Mbyte/s).

  Processing is estimated at 8MIPS and storage is negligible.

- The control process would take very few resources. It would only be activated a few times for every static Huffman block, for every section and for every session.

- Dynamic Huffman coding would replace static Huffman coding. It would take the same substring and bit field streams.

  The algorithm would require 200MIPS.

  Storage amounts to 8kB for a tree with 512 leaf nodes.

5.3.2 Software Hardware Tradeoff

Tasks that process the full data stream must be implemented in hardware. The full data stream constitutes uncompressed bytes (at 12.5 million per second), compressed bytes (at 6.25 million per second) and lengths, literals and addresses (at 4.2 million literals or length, address pairs per second). These rates are such that even trivial processing (including fetching and storing) of these data streams would cost a significant fraction of the processing power of a fast processor (25–50MIPS). The involved tasks are bus interfacing, sliding window coding, statistics collecting, buffering, Huffman coding and byte conversion as well as the corresponding tasks for decompression.

Huffman tree computation, tree transmission and tree reception are performed in software for the following reasons:

- These tasks have a high complexity with heterogeneous operations.

- These tasks combined have a reasonable work load for a fast processor.
- Very little can be performed in parallel.

It was decided to employ one programmable processor in the device both for the tasks mentioned above and for the overall control of the device. The overall control comprises initialization and synchronization of the tasks performed by other modules.

The internal processor interacts with the other hardware modules through I/O ports on an internal bus.

Sorting of symbol frequencies is an essential part of the chosen static Huffman tree computation algorithm. It would comprise most of the work load for the internal processor. Alternatively it could be performed in hardware:

- Special hardware to perform sorting. This would require complex arrays of associative cells, which are very costly.
- On-the-fly sorting as part of statistics collection. It could barely be implemented with standard RAMs. It would take several times the size of the statistics RAM.

5.3.3 Interfacing and Synchronization

The main data streams are transferred over dedicated data channels with handshaking. These channels are bidirectional.

The statistics must be transferred from the hardware module that performs the statistics collection to the processor that runs the tree generation process. The tree description must be transferred from the processor to the Huffman coding hardware. It is decided to use sharable RAMs for this transfer. Such a sharable RAM consists of two RAM memories (banks) that can be connected to two different memory buses. Each memory bank is connected to only one memory bus at a time, but the connections can be switched (under control of the internal processor). One bank can be accessed by the hardware module while the other one can be accessed by the processor. When the connections are switched, the bank that was accessible to the hardware module, becomes accessible to the internal processor and vice versa.

The internal processor communicates with all other modules using an internal processor bus that is connected to various RAMs and I/O ports on the hardware modules. The hardware modules can demand the attention of the internal processor by setting an interrupt. When the interrupt is set, the processor interrupts its normal program and executes a special subroutine (the interrupt handler). The interrupt handler checks the status of all the hardware modules and takes care of any special conditions

Synchronization at the byte level or at the length, literal and address level is performed in hardware by handshaking on the data channels. Synchronization at the level of (static Huffman) blocks, sections and sessions is performed in software. These two types of synchronization interact with one another. Blocks of lengths, literals and addresses are managed by the buffering task. End of section and end of data markings travel along with the data on the data channels. Whenever a special action is needed due to a block, an end-of-section or end-of-data condition, the hardware module involved sends an interrupt to the internal processor.

Synchronization at the level of DMA blocks (input data and output data blocks) is performed between the bus interface and the host system and is completely invisible to all other modules including the internal processor.

5.3.4 Algorithm and Implementation Selection

The bus interface must perform simple operations at a very high rate. Therefore it is implemented in hardware. A standard-cell design can be generated by a silicon compiler.

For sliding window coding the following algorithms and implementations would be possible:

- A software program would take 1000MIPS if a hashing or tree based algorithm is used. The required amount of RAM would be several times the window size.
- The same algorithm could be implemented with a state machine in hardware. This would reduce some of the instruction overhead.
- An associative array of parallel cells would implement the algorithm at one byte per clock cycle. Each cell would store one byte plus one flag bit. A non-greedy matching algorithm can be implemented with one additional flag bit per cell. The compression improvement would be comparable to doubling the window size.

Due to the high speed requirements we choose the associative cell array. We implement the version with the non-greedy matching algorithm with a size of 8kB. The same cell array can be used for decompression at little extra cost. Sliding window decompression could alternatively be implemented with a state machine and a RAM that is used as a ring buffer.

The buffering process performs simple operations at a high rate. Therefore it is implemented in hardware with standard cell logic. The RAM is used as a ring buffer. Some registers would manage the blocks. When the device is decompressing the buffering process would act as a ring buffer without block management. Due to the size the RAM is external to the chip. This adds about 35 pins to the package. we add 2 static RAM chips of 32k$\times$8bits each.

Statistics collection is also a simple operation at a high rate. Like the buffering process this should be implemented in standard cell hardware. The RAM is internal to the chip.

Huffman coding could be implemented by the following algorithms:

- A bit-serial algorithm using a tree. This would be too slow. The total compressed bit rate is 50 Mbit/s. As the clock frequency is 50MHz this would be the absolute peak rate for a bit serial algorithm. As one has to allow some idle time during the tree transmission or reception process the speed is not adequate.

- Table lookup. This would work at one clock cycle per encoded symbol. This is adequate.

For Huffman decoding the following algorithms exist:

- A bit serial algorithm. This would be inadequate.
- Table lookup using a reverse mapping from the next 16 code bits to symbol. This would require a too large memory.
- A special associative memory with a mask in every cell. This would require associative memories which are awkward to implement in standard cell logic.
- A mixed table lookup/bit serial approach. The next 9 code bits are mapped to either a symbol or an index in another table. From the other table one can work bit-serially. This would be the optimal choice.

The Huffman coder and decoder can be implemented in standard cell logic. The code RAM is internal to the chip.

Byte conversion is also performed by standard cell hardware. The hardware would contain a barrel shifter or a similar device.

Huffman tree computation is performed in software. The standard $I(n \log n)$ heap algorithm would be too slow. A very fast radix sort algorithm combined with an $O(n)$ Huffman algorithm by Lu an Chen[LC90] is fast enough.

Tree transmission and reception are also performed in software using straightforward algorithms. This software has to interface to the byte converter through I/O ports.

Dynamic Huffman coding is not implemented due to the high complexity of the parallel algorithm that uses a very complex parallel cell array. A bit-serial algorithm would not be fast enough (see section A.3, dynamic Huffman coding).

Table 5.3 shows how the implementation selection is translated into the module decomposition of the architecture.

5.4 Making the Architecture

Making the architecture revolves around subdividing the design into a set of communicating units, also called modules. The behaviour of each of the modules and the interfacing between them are specified in great detail. Figure 5.5 shows the hardware architecture diagram. The architecture is described in detail in appendix E (detailed description of device).

The architecture was based on the behavioural model (both functional and temporal behaviour), on the initial architecture and on the algorithm and implementation selections. The software/hardware trade-off was a very important factor for the architecture.

If software is used within the device, a processor is needed. One can use an existing processor or define a new processor architecture. We did not use an existing processor architecture for the following reasons:

Module	Function and implementation
Bus interface module	bus interface Hardware
Sliding window coder	sliding window coding Hardware + associative cell array
Buffering and statistics module	buffering, statistics (later also sliding window decoding) hardware + internal RAM (statistics) external RAM (buffering, sliding window decoding)
Huffman coder	static Huffman coding + byte conversion
Internal processor	control + tree computation + tree transmission processor + software
Not implemented	dynamic Huffman coding

Table 5.3: Module Decomposition and Implementation Selection.

- Simple embedded controllers (like the 8051 family) were not fast enough for the computational tasks. 16-bit operations and array indexing would require several instructions.

- RISC processors were fast enough but they would incorporate features that are expensive to implement and completely unnecessary, such as a large 32-bit register set and multiplication instructions.

No existing processors seemed to be fast enough for the computational tasks without being unnecessarily complex.

Within the architecture the following types of interfacing must be defined with respect to the processor:

- The hardware interfacing of the processor to the rest of the system. A processor usually communicates with the rest of the system through a processor bus, consisting of data, address and control buses. The signals on the bus must be specified and also the memory and I/O map. The memory and I/O map defines which memories and I/O ports are accessible at which addresses.

- The interfacing of the software to the processor, i.e. the instruction set architecture.

Apart from the processor, the device contains three main hardware modules to implement parts of the compression and decompression algorithm and it contains the bus interface to communicate with the host system. Various modules work on the data in parallel, in a pipelined fashion. Several synchronization mechanisms guarantee that the functional behaviour is correct regardless of the relative speeds of the modules.

As the module behaviour of all modules and the interfacing between them is specified to great detail, it is possible to assign the design of each of the several modules to a different designer or design team. Specification documents were written for each of the five hardware modules. The bus interface and the embedded processor were actually implemented by others.

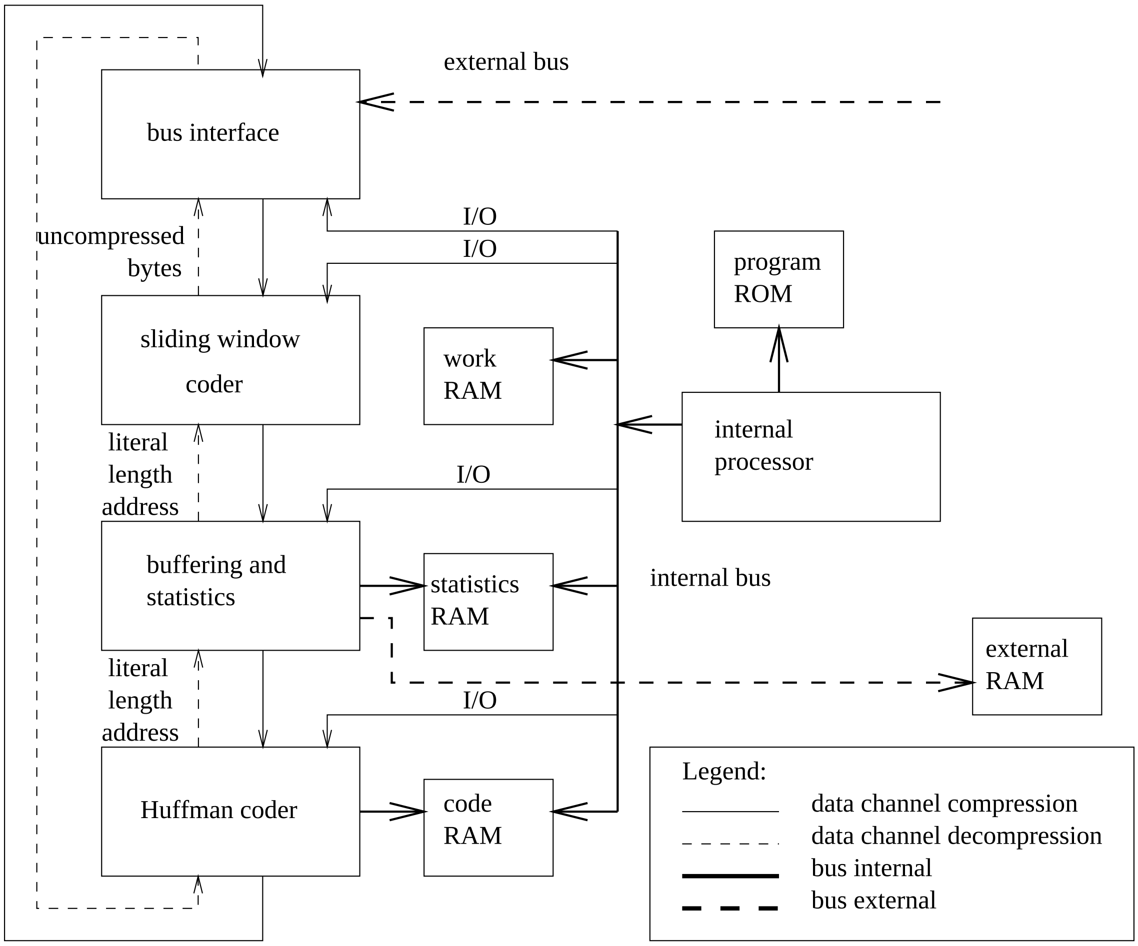

Figure 5.5: Hardware Architecture Diagram.

When the architecture was fixed, a simulatable VHDL model was planned, but not implemented. An IDaSS model was created that incorporated all modules as subschematics with all interfacing between them. The model contained all the on-chip RAMs. The bus interface and the buffering and statistics module were incorporated into that model. It was hoped to convert the whole model to VHDL using automated tools, but attempts failed. This was mainly caused by the fact that the translation tool from IDaSS to VHDL was not completed at that time.

5.4.1 Module Decomposition

The device consists of the following hardware modules (see table 5.3:

- The bus interface communicates with the host system. It was originally specified before the architecture was defined. Later some additions were made to fit the module into the architecture.

- The sliding window coder performs sliding window coding and decoding.
- The buffering and statistics module performs buffering and statistics collection. The buffer RAM is an off-chip memory, only accessible by this module. The statistics RAM is on-chip. Statistics collection was made a logical part of this module as it consumes the same stream of literals, lengths and addresses. The architecture would be less clean if the same channel had two receiving modules.

  Later the task of sliding window decoding was added to this module.
- The Huffman coder converts lengths, literals and addresses to and from compressed bytes. Descriptions of the Huffman code to use are in an on-chip code RAM. The byte converter is a part of this module. During decompression the interface between the byte converter and the Huffman decoder would have to be bidirectional as the receiving module (the Huffman decoder) is going to determine how many bits it will receive per bit field. As both modules have to work so close together at such a high speed, we decided to combine them into a single module.
- The internal processor serves to run the tasks that are implemented in software.

Apart from the hardware modules the software on the processor can be seen as a module in its own right. The software was not further divided into modules. See section E.3 (module specifications) for a more detailed description of the modules.

The three main data stream processing modules (sliding window coder, buffering and statistics module and Huffman coder) can be in three working modes:

- Idle.
- Compressing.
- Decompressing.

Depending on the working mode they use the data channels in one of two directions. The processor can set the module in one of these modes through a command on an I/O port. In some cases the modules can put themselves back in idle mode.

In each mode the sliding window and Huffman modules have two submodes: processing and standby. In processing mode these modules work without processor interaction. In standby mode the processor can communicate with the module. The processor can send or receive variable length bit fields through the byte converter in the Huffman coder when it is standby. The module puts itself to standby mode when a certain event occurs, in which case it generates an interrupt.

The buffering and statistics module can execute two concurrent processes: writing a block into the buffer and reading a block from the buffer. Both processes can be active or inactive independently. Alternatively the module can be active as a FIFO buffer without regard to blocks.

5.4.2 Hardware Interfacing

The following types of hardware interfacing exist between the modules of the device:

- The data channels were designed to transfer data at a high speed between two modules. Some data channels carry bytes, others carry lengths, literals and addresses. All data channels can carry end-of-section and end-of-data marks. The byte channels have two extra signals to indicate these marks, on the length, literal and address channels these marks are signified by special data values. In the former case all 256 byte values were already in use for normal data, so extra lines had to be used. In the latter case this was not necessary. In a previous version of the specification the length, literal and address channels did have dedicated signals for these marks and they had more data lines so an address/length pair could be transferred as one item. This was changed to save some data lines and because it was easier to process lengths and addresses sequentially. Consequentially the length, literal and address channel has a 'state'. Whenever the current item denotes a length, the next item will be an address (and cannot be an end-of-section or end-of-data) mark. At several places in the design there is logic to keep track of the channel state.

  The protocol is a synchronous handshaking protocol. The sending module sets the 'available' signal when it has data available and the receiving module sets the 'ready' signal when it is ready to accept data. If the two signals are both set, the data transfer takes place at the next active clock edge (see appendix E.2 (hardware interfacing)).

  Although this protocol permits a throughput of one data item per clock cycle the actual modules limit the throughput to one item per two cycles.

  At first the channels were defined as bidirectional with all lines shared by both directions. All lines were driven by tristate buffers. The receiving module would float all lines except the 'ready' line. The transmitting module would drive all lines except the 'ready' line. As the proper timing of the initialization of the modules could be a problem, the handshaking signals were not shared and driven to zero when not in use. Even later not even the data lines were shared, so there were in fact two unidirectional channels. According to Verschueren [Ver96] tristate buffers for on-chip signals should be avoided when possible.

- Internal RAMs hold the statistics and the Huffman code. In the high-level compression pipeline one block is being Huffman encoded by the Huffman coder that needs access to the code RAM. The second next block is being sliding window coded while the statistics module updates the statistics RAM. The intermediate block resides inside the code buffer and its statistics are being processed by the software. The software needs to read the statistics RAM and it must fill the code RAM. Both RAMs have to be accessible by the processor, but access is needed for a different data set from the one accessed by the other module (statistics module or Huffman coder).

This situation calls for a solution in which two banks exist of both the code RAM and the statistics RAM. If there were only one bank of the statistics RAM the internal processor would have to copy the contents to its own RAM and clear the bank before the statistics module could continue with the next block. During this time the statistics module would be idle. Similar considerations apply to the code RAM.

The memories do not have to be dual ported. An appropriate switching mechanism (controlled by the internal processor) can switch the interface (data, address and control lines) of each of the memories to either the internal processor bus or the dedicated memory interface of the other module. Such a switching mechanism would consist of a set of multiplexers.

- The internal processor bus connects the internal processor to its own RAM, the banks of the statistics and code RAMs and to I/O ports of other modules. Each module would have several I/O ports. The memory and I/O map is shown in appendix E.4.

  A synchronous protocol is used in which one data transfer is possible per clock cycle. Initially the data bus was bidirectional, but this was changed to separate data buses for input and output.

  A single level-sensitive internal interrupt was defined that is shared by all modules. When the processor reads the status register of a module, the module deactivates its interrupt signal.

The interfaces are described in more detail in section E.2 (hardware interfacing).

5.4.3 Processor Instruction Set Architecture

This section describes the instruction set architecture of the internal processor and the design decisions that form the basis of the instruction set architecture. The processor is designed to perform the Huffman tree processing tasks fast. Further it must perform low-speed control tasks, but is is not really optimized to perform those efficiently. Speed is not very important for those operations.

We left out features that we really did not need, such as signed arithmetic with associated flags, multiplication and division and byte addressing. Further there is no point in having a 32 bit word length if none of the data are more than 16 bits wide.

We decided to implement a Harvard architecture. This means that the program and data memories are separate. They are on separate buses, each with its own data and address lines. We chose this for the following reasons:

- High-speed processing is desired. Every instruction needs an instruction fetch and many need a data read or write. As the data read and write use an entirely different memory bus from the instruction fetch, both can occur in parallel.
- There is no reason to load or modify programs.
- The buses are on-chip, so there is no high cost associated with an extra bus.

There was not enough demand for ROM tables to justify a read operation in the program ROM. The data address space includes both memory and I/O ports (memory mapped I/O). There would be no speed advantage if there were dedicated I/O addresses and address decoding is only slightly more complicated with memory mapped I/O. On the other hand, we would have needed dedicated I/O instructions without memory mapped I/O.

5.4.3.1 Supported Operations.

The processor must be fast at the Huffman algorithm (Lu/Chen) and the associated sorting. Those algorithms use only very limited operations like addition and subtraction. Further the processor must perform simple control tasks that necessitate bitwise logical operations. Shifts are needed at some places, but the shift operations normally associated with the concatenation and extraction of bit fields in the compressed data stream is already performed by the Huffman coding unit, so no variable-distance shift operations are needed.

To implement the algorithms there will be a need for the following operations:

- Arithmetic operations like add, subtract and compare.
- Logical operations like 'and', 'or' and exclusive or.
- Simple shift operations.
- Move operations from memory to registers, from registers to memory and between registers.
- Conditional and unconditional jumps.

The largest quantities we will ever need to handle are the 16-bit code words. Frequency counts are limited to 13 bits. Therefore we decided to use 16-bit data operations and registers. There is no need for multi-precision arithmetic and therefore we do not support a carry flag and operations that depend on it. Neither did we support signed arithmetic or overflow detection.

As a single accumulator architecture would necessitate many moves from and to the accumulator, we decided to make an architecture with general purpose registers. As determined from some of the time critical algorithms, four general purpose registers would be too few and 8 would be adequate. Therefore we decided to implement 8 general purpose registers.

We decided to have a few flag bits in a status register and have conditional branches depending on those flag bits. No signed arithmetic is performed, so we decided to omit the traditional Overflow and Negative flags. No carry flag in the traditional sense is needed, but we need to discriminate between 'less than', 'greater than' and 'equal' in the compare instruction. Two flags were reserved for the result of the compare instruction. There are conditional branches for all six traditional compare conditions.

As we need a special value (no next item) in the sorting algorithm, and 0 is unsuitable for that purpose (it occurs as a regular value), we decided to use the word with all bits set as another special value and made a flag specially to test against this value. We ended up having two flags (Zero and Set) that are set at moves and other operations. These flags are separate from the comparison flags . There are four conditional branch instructions to branch if these flags are set or clear. As the word with all bits set is the result of subtracting one from zero, the 'Set' flag turned out to be quite useful in loops with decrementing counters.

To optimize the radix sorting algorithm even further, a few operations were added to speed up the operation. These are the 'sel2' and 'sel3' instructions to extract a specific bit field from a register.

5.4.3.2 Addressing Modes.

As the Huffman related algorithms need about 2 kilowords of working storage, not counting the Huffman coder RAM, we decided to use a 12-bit data address size, which could address 4 kilowords, including working RAM, statistics RAM, coder RAM and I/O ports.

Some information that is rapidly changing and has a limited life time, is stored in registers. Examples are counters, index variables and computed totals. All arithmetic and logical operations are supported between two registers. The unary operations (like 'set' and 'shr') only work on registers. Of the binary operations (like add and 'or') the destination is always a register. The 'move' operation is the only operation whose destination can be in memory.

Some data cannot be stored in registers, but consist of a single word stored at a fixed address. These are normally accessed with direct addressing. With direct addressing one would access the following information:

- Single (scalar) variables that are accessed less frequently or are to be kept longer than registers, mostly state information and parameter settings.
- I/O ports.

As the address size is 12 bits and we would like a uniform instruction length, it is not practical to include the full address in the instruction. As the I/O addresses are not necessarily adjacent to the area where the scalar variables are stored, we decided to implement *two* areas for direct addresses, both containing 64 addresses. In the basic architecture the base addresses of these areas are stored in two registers, the direct base registers. In the real implementation these addresses are most likely hardwired. If direct base registers are used, one would hardly call it direct addressing, but these registers are likely to contain a fixed address during the execution of the program.

The processor is optimized for array accesses and therefore there is a form of indexed addressing. In indexed addressing the effective address is the sum of an index register and a base address. The index register is one of the 8 general purpose registers. The base address is stored in one of a set of 'index base registers'. Four bits are reserved to encode the index base register, hence there can be up to 16 of them, but 8 would suffice. As there is much non-sequential array access, the base-index approach of array indexing (as opposed to pointer registers) is really preferable. We chose a separate set of array base registers, because we would otherwise need more general purpose registers, necessitating a larger instruction size. Furthermore, some operations would necessitate three simultaneous general purpose register accesses instead of two. Finally the index base registers are 12 bits whereas general purpose registers are 16 bits.

In every computer program there are many instructions that have a small constant as one of the operands. Therefore most processors have immediate addressing. As we would like a uniform instruction size, we cannot put a full 16-bit immediate constant in an instruction. We chose an 8-bit immediate constant instead, ranging from 0 to 255. With a specialized load instruction for the high half of a register, we can load an arbitrary value into a register in two instructions. The binary operations (move to a register, 'add', 'or') can use immediate addressing.

As all instructions have the same size, we did not separately implement instructions that can be replaced by one different instruction. The increment and decrement operations can be replaced by an add or subtract operation with an immediate value of 1. Likewise, the shift left operation is an addition of a register to itself.

5.4.3.3 Instruction Length and Program Size.

To simplify instruction decoding and pipelined implementation, we decided to make all instructions the same size, like most RISC processors do. As we have a Harvard architecture, this size could be different from the data size.

Consider the class of immediate operations. We need 8 bits for the immediate constant, 3 bits for the operation and 3 bits to indicate the register. This makes 14 bits, plus one or more bits to indicate that we have an immediate operation. For an adequate encoding of all instructions (memory load/store, conditional and unconditional jumps, register to register, immediate and others) we would need 16 bits. The encoding shown in appendix E.5 (instruction set) shows that it can be done in 16 bits.

We estimated we would need about 4000 instructions, therefore we chose an instruction address size of 12 bits.

5.4.3.4 Other Operations.

As the processor is used in control applications and must respond to some signals that occur at an unpredictable time, it needs an interrupt facility. Interrupts occur infrequently so the interrupt response time is not of prime importance. Only one interrupt input is needed and no nested interrupts are supported.

Apart from an interrupt facility a subroutine call is highly desired. There are two different ways to save the program counter (and status flags) in case of an interrupt or subroutine call:

- A stack.
- A dedicated register, often called link register.

For interrupts, a link register would be adequate, but for subroutine calls we would need special instructions to save and restore the link register if we needed to nest subroutines. In this case the hardware would not be much simpler than with a stack in memory. As the program counter is 12 bits and the status flags are 4 bits, they would fit together into one data memory word.

An interrupt is implemented by inserting a subroutine call instruction into the instruction stream. This would save the status flags and the program counter on stack. A return from interrupt would restore the status bits and re-enable the interrupt response, a return from subroutine would not do so. There are also instructions to push and pop registers on the stack. The stack pointer is a dedicated register.

There are special move instructions from a general purpose register to the stack pointer, a direct base register or an index base register. Those registers can be set, but they cannot be read back. Therefore an interrupt handler cannot save them, but it is not really needed.

The buffer and statistics module has hardware to increment locations in the statistics RAM, but it does not clear them. This task is reserved for the internal processor. In the radix sort algorithm some smaller memory areas must also be filled. Filling a memory region using a loop would cost three instructions per word: a store instruction, a register decrement instruction and a conditional branch. In a pipelined architecture the branch would cost many cycles. With partially unrolled loops the branch overhead could be reduced. To make filling as efficient as possible, a special fill instruction is introduced. It takes two general purpose registers and one index base register. One general purpose register is used as the source register and the other one is used as an index register. The source register is stored at the location whose address is the sum of the index register and the base address. Then the index register is decremented. The instruction repeats itself until the index register is decremented through zero. This instruction satisfies the register file constraints by reading two registers and writing one.

5.4.3.5 Evaluation of the Architecture.

Initially all binary operations could be used with one of the operands in memory. The data path would thus go through memory and through the ALU. This would necessitate a pipelined implementation with many stages. To reduce the number of pipeline stages it was decided to change to a load store architecture. Only move operations can have an operand in memory. It is either move from memory to a register (load) or from a register to memory (store). Other binary operations could no longer take memory operands, but they could take register or immediate operands. This modification increased the number of instructions in the Huffman tree computation program only very slightly and no instructions were added to the critical loops of the radix sort.

The 8 general purpose registers form one register file. In each instruction at most two registers are read and at most one of them is written. This fact was observed during the instruction set design and care was taken to keep it that way.

Adding autoincrementing or autodecrementing would necessitate the modification of two general purpose registers in one instruction, so it was not adopted, even though the move to a load store architecture would free up the instruction bits to encode them.

The direct base registers will not likely be implemented as real registers, but they will be hardwired to fixed addresses. The same could be done with the 16 potential index base registers, but this approach has some disadvantages:

- The same subroutine cannot be called with different arrays as parameters.
- 16 Arrays would probably not be enough for the entire program.

5.5 Design of Individual Modules

The architecture defines the detailed behaviour of all modules and the interfacing between them. Hence it is possible to design each of the modules independently. The design of each module can be viewed as a separate design problem. We will review the design of each module in turn.

5.5.1 Bus Interface

The specification of the bus interface was made before the architecture. Therefore the module has the following properties:

- The bus interface performs all its tasks autonomously. It does not rely on the presence of an embedded processor to perform high level control tasks.
- Its interfacing is not adapted to the internal processor bus.

These properties have the advantage that the module can better be reused in other designs.

The original interface between the bus interface and the rest of the device was directly derived from the interface between the bus interfacing and core processing in the behavioural model.

- The core input channel carries a byte stream from the bus interface to the core processor. This stream includes end-of-section and end-of-data marks.
- The core output channel carries a byte stream from the core processor to the bus interface. This stream includes end-of-section and end-of-data marks.
- The core status channel carries an 8-byte error code from the core processor to the bus interface (one per session).
- The core command channel delivers commands to the core. It also carries two abort signals, one from the core and one from the bus interface.
- The core parameter channel is an interface to a parameter memory that resides inside the core processor.

All channels are synchronous channels with handshaking.

The module was designed by Camps[Cam95]. He made an architecture with the following main submodules:

- The data path that carries the data between the external data bus and the byte channels.
- The data path control module that controls the handshake signals and the byte counters.
- The general control and self-test module.

This module was implemented and simulated in IDaSS. Some changes were later implemented by me.

The following hardware was added to make the bus interface fit into the rest of the architecture:

- A parameter RAM of 16 bytes, accessible either by the parameter port of the bus interface or by the internal processor.
- Switching logic to route the input byte channel either to the sliding window coder (uncompressed bytes) or to the Huffman coder (compressed bytes). The output byte channel is routed similarly.
- An interface to provide I/O ports on the internal processor bus. The command channel, the abort signals and the core error code are made accessible as I/O ports. The arrival of a command or the abort signal from the bus interface causes an interrupt. Further an I/O port controls the accessibility of the parameter RAM and the switching direction of the byte channels.

5.5.2 Sliding Window Coder

The heart of the sliding window coder is the associative cell array and most of the work was concentrated on that. Details can be found in appendix D (simulations).

The basic algorithm uses a large cell array and a priority encoder that finds the first matching cell in the whole array. This was fist described in [Ben93]. An 8 cell version of the sliding window coder was created in IDaSS and later translated to SID, the language of the ASA silicon compiler. An 128-cell version was created with ASA.

When a read buffer is added to each cell and an address decoder is added to the whole array, one can use the same array for decompression as well.

The array can be laid out in a matrix in which there are two priority encoders, one for the rows and one for the columns. Similarly there are two address decoders. It is conceivable to make an array of 32 rows and 256 columns making a total of 8192 cells.

Cells for such an array were designed at the transistor level and implemented with the sticks layout tool of the Compass tools. Simulation results of a single cell are available. Many dirty tricks like precharge logic were used. We felt uncomfortable with the use of such tricks so we would go for a standard cell lay-out even though it has a higher cost.

Later a non-greedy matching (lookahead) algorithm (see appendix A, algorithms) was found that could be implemented in a cell array. Each cell would contain two match flags instead of one. Such a cell was designed at the gate level. The behaviour of the array at the register level was simulated with a C program.

5.5.3 Buffering and Statistics

The original implementation of the buffering and statistics module was implemented and simulated by me in IDaSS. Due to some restrictions on signals usable by state machines in the then current version of IDaSS I avoided the use of state machines as much as possible and implemented them as registers and operators instead.

The original implementation contained the following parts:

- A module to write incoming data to the external RAM at increasing addresses.
- A module to read outgoing data from the external RAM at increasing addresses.
- A module to arbitrate the RAM access between the two modules described above.
- A module to manage the limit registers that contain the start and end addresses of all static Huffman blocks.
- A module to implement the I/O registers on the internal bus.
- Multiplexers to route the incoming and outgoing data to the appropriate channels.
- The statistics module.

- A central controller.

The statistics module is an autonomous submodule that acts on the lengths, literals and addresses from the sliding window coder. It just reads the data and handshake signals from the channel and does not itself have a way to control the flow on the channel. As the buffering module limits the flow to one item per two clock ticks, the statistics module has to cope with that rate. It needs exactly two clock ticks to update the RAM for each item. The statistics module must determine the class number of an address. The same logic is also needed by the Huffman coder. It was not thought worthwhile to share the logic of this operation between the two processes. For `gzip` the class determining logic is much more complex.

Later the task of sliding window decompression was added to the buffering and statistics module. This would necessitate a closer integration of the memory reading and memory writing logic. As many registers and adders as possible must be reused between sliding window decompression and buffering. The statistics module remains an autonomous submodule. To save memory bandwidth during the sliding window decompression process the bytes would have to be read or written to the external RAM in 16-bit pairs.

As there is no longer a buffer between the Huffman decoder and the sliding window decoder, the sliding window decoder has to work in bursts at a rate higher than 12.5 million uncompressed bytes per second. This burst rate must be handled by the bus interface as well. Alternative buffering solutions are proposed in chapter 7.

A sustained memory bandwidth of 25 million read or write operations per second is considered feasible. The sliding window decoding process would need to write 12.5 million bytes per second on average (which is 6.25 million 16-bit writes). The worst case reading rate occurs with unaligned substrings of length 4, which need 3 read operations for 4 bytes. This would amount to 9.5 million read operations per second. Typically it is closer to 6.25 million accesses per second. Combined it means an average memory access rate of 12.5–16 million operations per second. The peak rate can be made 50–100% higher than the average rate to accommodate bursts in the sliding window coder decoding operation.

5.5.4 Huffman Coder

The static Huffman coder itself has not been implemented or simulated in IDaSS.

Huffman compression would be implemented by table lookup in the code RAM. The table contains both the code word and the code word length.

Huffman decompression would be implemented by reverse table lookup on the next 9 bits of the compressed data. The table lookup yields the decoded symbol and the code word length. For code words longer than 9 bits additional table lookups are needed, but these occur comparatively rarely. The algorithm to decode code words longer than 9 bits uses the fact that the Huffman tree is in canonical form and has very low storage requirements.

The Huffman coder would need to keep track of literals, lengths and addresses in the data stream. Depending on the state it must perform the correct Huffman coding or decoding operation. Addresses have to be split in a class number and an offset within the class. Both have to be encoded separately. When an address is decoded its class number is decoded first and the offset next. Class number and offset have to be combined into an address. For `gzip` this operation is more complex. In `gzip` lengths use a class offset system as well and there are more address classes than in LHA.

The byte conversion process was implemented and simulated in two IDaSS models, one for conversion of bit fields to bytes and one for conversion of bytes to variable length bit fields.

5.5.5 Processor

The internal processor was implemented and simulated in IDaSS by Wijffels[Wij96], using many of the guidelines described by Patterson and Hennessy [PH90].

To achieve a high speed a pipelined architecture with five stages was necessary. Critical path analysis was performed. It is expected that the processor can be implemented at a clock rate of 50MHz. In programs with tight loops nearly 60% of the cycles are wasted on various types of stalls, so an average rate of 20MIPS is achieved.

Small changes were made to the original instruction set. The architecture was changed to a load store architecture to use either a path through RAM or through the main ALU, but not both. This requires fewer pipeline stages. The load store architecture contains five pipeline stages and consists of the following parts:

- Fetch stage. This stage contains the program counter and fetches the next instruction from ROM. It implements the 'pre-fetch' and 'fetch' stages of the pipeline.
- Decode stage. This stage decodes the instruction and its operands. It also contains the general purpose registers and the base registers.
- Execute stage. This stage contains the main ALU.
- Write back stage.

The specification of the internal bus was changed to have 'data in' and 'data out' lines. This better matches the behaviour of the available on-chip RAMs. These need their write data concurrent with the write address, while they output their read data the cycle after the read address. With one data bus there could be no write cycle directly after a read cycle. Separation of the read and write data lines also avoids tristate buses.

Later Verschueren converted the IDaSS design to VHDL and synthesized a layout for it using the Compass Tools. The implementation contains about 37000 transistors or about 9500 gate equivalents. It has an area of 34.7mm^2 in 1μm technology.

5.5.6 Software

The software that runs on the internal processor has not yet been written.

The main program would wait for a command and depending on the command it starts a compression or decompression session, runs a self test (RAM test) or checks the compression parameters.

When the main program starts a compression session it initializes the other modules and next it enters a loop in which it performs Huffman computations on static Huffman blocks. It communicates with the interrupt handler by means of some flags in memory and also directly with the hardware. The interrupt handler communicates with the hardware. The whole process must be free of deadlocks or critical races.

When compressing the Huffman loop contains the following steps:

- Compute the code word lengths for the length/literal code and the address codes from the statistics with the Lu/Chen algorithm.
- Compute a Huffman code (tree code) to compress the code word lengths of the length/literal code.
- Send the length and literal count of the block with the compressed data.
- Send a description of the tree code with the compressed data.
- Encode the code word lengths of the length/literal code with the tree code and send them with the compressed data.
- Send the code word lengths of the address code with the compressed data.
- Construct the code words from the code word lengths (canonical Huffman code) and program the code RAM.

When decompressing the Huffman loop contains the following steps:

- Read the length and literal count from the compressed data.
- Read the description of the tree code from the compressed data.
- Read the code word lengths of the length literal code from the compressed data and decode them with the tree code.
- Read the code word lengths of the address code from the compressed data.
- Construct the code words from the code word lengths and program the code RAM.

The above steps apply to static Huffman coding for the LHA algorithm. For `gzip` the steps are very similar.

An assembler, an instruction-level simulator and a disassembler for the internal processor were written in C to assist software development.

The tree computation algorithm (computing code word lengths from frequencies and code words from code word lengths) was implemented in assembler (see appendix D, simulations). These are the first and the last steps of the compression loop.

5.6 Adaptations to the Design

For a non-trivial design the initial specification often needs some changes. Parts of the specified behaviour may turn out to be impossible to implement or at least extremely costly, Not only the specification, also the behavioural model and the architecture can be prone to changes after they have been defined.

The first specification described many desirable compression options. Based on cost and benefit, some features were removed from the specification. Therefore dynamic Huffman coding and a few options of sliding window coding were eliminated.

Multi-file sessions were not originally part of the specification. Because the use of end-of-section commands was not meaningful for static coding and because the same mechanism could be used for multi-file sessions it was decided to implement multi-file sessions with static Huffman coding. As static Huffman coding has a very high latency, multi-file sessions are highly desirable.

While the architecture was already defined, the exact specification of the hardware interfacing between modules underwent some changes. Splitting the data bus lines for the internal bus into separate 'data in' and 'data out' lines is the most important example of this.

A major specification change was the change of the compression format to `gzip` instead of LHA. The most visible architectural change was the relocation of the sliding window decompression task to the buffering and statistics module (see figure 5.6). As a consequence the decompression data stream no longer passes through the sliding window coder. The bidirectional data channels that were already implemented as pairs of unidirectional channels had to be made unidirectional. Demand from the industry was the main reason to investigate this change.

The necessary architectural changes as a consequence of the format change gave us a good insight in the way the design depends on the parameters of the compression format. This enables us to generalize the design so it can be readily adapted to many compression formats.

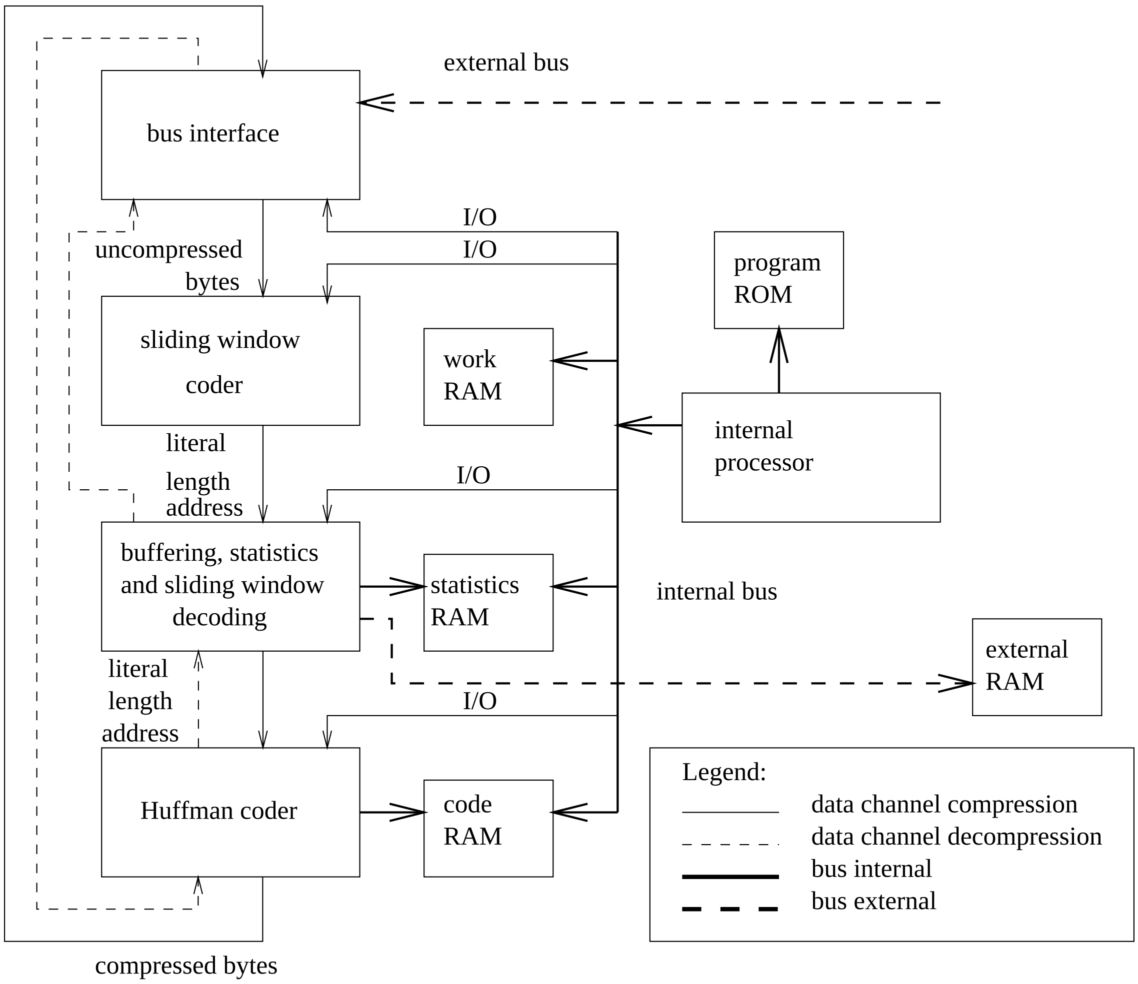

Figure 5.6: Hardware Architecture Diagram for GZIP

Chapter 6

Description of Solution and Alternatives

The result of the design process is a detailed description of a VLSI device that can be manufactured. This chapter gives a summary of the architecture and the properties of this device. For exact details see appendix E.

The specifications are shown in appendix E.1 (data sheet). The device compresses and decompresses data using the 'deflate' method that is used by the `gzip` program. This method is specified in [Deu96a] and some remarks on the method are made in appendix B.1 (compression format).

The device is used inside a host computer system that contains a CPU, memory, a DMA controller and I/O devices. The input data for the device is read from memory through DMA and the output data of the device is written to memory through DMA. The CPU and the device communicate through I/O ports and interrupts. The device can compress and decompress, but only one of these tasks can be in progress at any given time.

The device consists of five main hardware modules that are interfaced to each other. The development of this hardware architecture is discussed in chapter 5. For a detailed description see appendix E.

For every system level design the design process does not yield the design of one single artifact, but the design of a class of artifacts. Apart from the device described in this chapter and specified in appendix E.1 (data sheet), which represents an optimum for one particular set of requirements, there are many alternative possible devices. This chapter categorizes several alternative directions.

On one hand the device as specified is complex and hence costly to manufacture. Some design alternatives reduce the complexity of the device at the cost of speed, compatibility with existing compression methods or compression ratio.

On the other hand there is always a desire for more features, better performance, compatibility with more existing formats or a better compression ratio. With the progress of VLSI technology some of the extra options will become affordable in the near future.

Finally there are parts of the device that can be reused in different devices.

6.1 Algorithms

The following algorithms are used to implement the compression and decompression tasks:

- Sliding window coding with a hardware array using non-greedy matching.
- Sliding window decoding with the external RAM used as a ring buffer, 2 bytes are read or written at once.
- Buffering with a ring buffer using the external RAM.
- Statistics with a RAM (read-increment-write at address determined by data value).
- Huffman encoding with table lookup.
- Huffman decoding with reverse table lookup with extra lookups for long code words.
- Byte conversion with barrel shift hardware.
- Huffman tree computations in software with the Lu/Chen algorithm, preceded by three-phase 4,4,5 bit radix sorting.

6.2 Testability

Most of the hardware of the device consists of standard cell logic, output by a silicon compiler. Many silicon compilers support scan-path testing for the standard cell logic.

All internal RAMs are accessible to the internal processor and it should be straightforward to run a simple test program on the internal processor to test all internal RAMs.

The sliding window coder, if not implemented with standard cell logic, will pose the most severe test problem. Fortunately the array already acts as a FIFO with respect to the data bytes. A special test mode could add FIFO behaviour for the flag bits. This makes it possible to store any combination of values into the array and read them back at the end of the FIFO to verify they have not changed.

The external RAM interface provides a convenient test point to verify the data as it comes from the sliding window coder.

6.3 Estimated Properties

The complexity of the circuit is dominated by the sliding window coder. When it is implemented in standard cell technology the number of gates per sliding window coder cell is 80. The rest of the circuit will take about 20000 gates, excluding memories. Size is determined by the number of gates, the bonding pads and the size of the memories. The total amount of RAM is 80kbits and the amount of ROM is 64kbits. For a 0.5μm process the per-gate area is estimated at 0.00025mm^2 per gate. The area for RAM is estimated at 0.125mm^2 per kilobit. The area for ROM is estimated at 0.05mm^2 per kilobit. To the total circuit area (gates plus memories) a 10% routing margin must be added. Around the whole circuit area (plus routing area) there must be a border of 0.5mm wide to accommodate the bonding pads.

Cost is a nonlinear function of the size and the number of pins. Power dissipation is also dominated by the sliding window coder. The per-cell dissipation is estimated at 0.4mW. Table 6.1 shows the estimated number of gates, size, cost and power dissipation versus the window size as well as the estimated compression ratio.

The number of pins is 86. The compression speed is 12.5Mbyte/s at a clock frequency of 50MHz.

Window Size	Compr Ratio bits/byte	No. Gates x1000	Area mm^2	Cost US$	Dissipation W
0.5k	3.6	60	44	50	1.0
1k	3.4	100	55	75	1.2
2k	3.2	180	80	150	1.6
4k	3.0	340	130	300	2.4
8k	2.9	660	230	600	4.0
16k	2.8	1300	420	1200	7.2

Table 6.1: Complexity versus Sliding Window Coder Size.

6.4 Alternative Architectures

There are several ways to implement the same behaviour with a different architecture.

The proposed architecture implements the whole device (except an external RAM) in a single chip. When the size of a chip exceeds a certain limit the yield drops dramatically and the per-chip cost increases. If this is the case it can be more cost-effective to implement the design in several chips. One of the options is a two chip design in which one chip implements the sliding window coder and the other chip implements all the other functions.

The two chips would need the following interconnections:

- Uncompressed bytes channel from the main chip to the sliding window coder: 12 pins.

- Length, literal and address channel from the sliding window coder to the main chip: 15 pins.
- Communication of the sliding window coder with the internal processor : 4 pins.

The communication with the internal processor should accommodate commands to the sliding window coder, status from the sliding window coder and parameter settings. Because of the low bandwidth a serial protocol is used. If sliding window decompression is also performed by the sliding window coder then the data channels need to be bidirectional with the associated cost of extra pins and/or tristate buffers.

Splitting the device into two chips would add 31 pins to the main chip. The sliding window coder chip has its own packaging and around 35 pins. Further there need to be controllers on both ends of the serial communication link between the two chips.

The proposed architecture has a considerable amount of RAM (nearly 10 kilobytes) on chip. On-chip RAM is more expensive per kilobyte than standard RAM chips. One could reduce the cost of memory by moving the RAM to external memory chips, but this would add to the number of external components and pins. Therefore it is only cost-effective for large memories. For the data compressor it means that it can only be cost-effective if the on-chip RAMs are lumped together into one larger memory, in which case bandwidth of the external memory port could be a bottle-neck. In the current architecture each on-chip RAM has its own connection to the module that uses it. With external RAM all memory accesses have to go through one or two memory buses. This means that there must be arbitration logic to share the bus among several competing modules and that wait-state logic must be added to modules that would not need it when there were no shared RAMs.

Let us compare the standard design and a design without on-chip RAMs.

- The standard design has only the buffer memory as external RAM (size 32k $\times$ 16 bits). The number of pins is 86, 35 of which are for the external RAM. The estimated access bandwidths are as follows:

  buffer RAM (external RAM) 6 million reads plus 6 million writes per second.

  Statistics bank 1 6 million reads plus 6 million writes per second.

  Statistics bank 2 0.25 million reads+0.25 million writes per second.

  Work RAM 6 million accesses per second (reading and writing).

  Code bank 1 8 million read per second.

  Code bank 2 0.5 million writes per second.

  These rates apply to compression. For decompression the access bandwidth of the external RAM must be allowed to have a burst rate of 25 million accesses per second. One of the code banks has a burst access rate of up to 16 million reads per second (average rate is close to 8 million per second). The other code bank and the statistics banks are unused.

It is assumed that an external RAM can have a maximum access bandwidth of 25 million per second and an internal RAM a bandwidth of 50 million accesses per second. None of the internal RAMs is used close to its maximum bandwidth.

- An alternative design uses two external RAMs, one of size 32k×16 bits and one of size 4k×16 bits. The total number of pins is 118, 35 of which are for one RAM and 32 are for the other RAM.

  The functions of the RAMS are as follows:

  - The 32 kiloword RAM is used for buffering and for both code banks when compressing. This yields a total access bandwidth of 20.5 million per second. When decompressing it is only used for sliding window decoding and has a burst access rate of 25 million per second.
  - The 4 kiloword RAM is used as the processor work RAM and for both statistics banks when compressing. This means a total bandwidth of 18.5 million accesses per second. When decompressing the RAM is used as the processor work RAM and for the code RAM. The total access bandwidth is 20 million accesses per second. During the static block decompression cycle the activity on the RAM alternates between work RAM activity (tree reception and decoding) and code RAM activity (Huffman decoding of the data).

  While the RAM access bandwidths are clearly adequate for typical data, they might pose a bottle neck for some data.

The alternative design needs 32 extra pins and complex arbitration and wait-state logic to share the RAMs among modules. Further it may suffer from RAM bandwidth bottle necks when a-typical data is processed. Further the memory access rates do not permit to use some RAM as an extra buffer between the Huffman coder and the sliding window decompression to smooth out the bursts.

In the current design sliding window decoding is implemented in the buffering and statistics module and not in the sliding window coder. This allows us to decompress a format with a larger sliding window than is available for compression. Further it has the advantage that the sliding window coder can be less complex. The read buffer can be removed from each cell.

The only significant problem with this approach is that the buffering and statistics module does no longer provide buffering between the Huffman decoder and the sliding window decoder, so that the bursts can be smoothed out. The sliding window decompressor with the external RAM is designed to handle the data at a higher peak rate than the original sliding window coder. Therefore an average rate of 12.5Mbyte/second can still be achieved. The peak load on the bus interface (which contains no buffering of its own) and the DMA subsystem of the host system is however increased, which could be undesirable. The following solutions are conceivable:

- Use the unused internal RAMs as a buffer between the Huffman coder and the sliding window decompressor. During decompression the statistics RAM is not used and only one bank of coder RAM is used. Part of the working memory of the internal processor is also unused. The total amount of internal memory that could be reserved for this buffer is 2 to 3k words of 16 bits, which is barely adequate. The disadvantages of such a solution are:
  - Low buffer capacity.
  - Complex bank switching of internal memories, hence increasing access delay.
  - The statistics RAM has a data width of only 13, which is not enough. Either it cannot be used or its width must be increased to 15 or 16.
- Use the sliding window coder as a buffer for the decompressed bytes between the buffering and statistics module and the bus interface. This has the following disadvantages:
  - The sliding window coder should again be equipped with a random access read facility, which we had just hoped to eliminate.
  - The peak shift rate should be increased compared to what is required for compression. The expected delay with this is however low compared to other delays in the coder. Peak dissipation is increased though.
  - If the sliding window size in the coder is small (4kbytes) the buffer capacity is still not great.
- The sliding window decoder already buffers the uncompressed bytes in the external RAM, by the very nature of the algorithm. We only need an additional read facility to utilize this for buffering of the uncompressed data stream. The disadvantage is that the bandwidth of the external RAM is probably not adequate to support the extra read activities.
- Instead of buffering the uncompressed bytes, buffer the lengths, literals and addresses into the external RAM. This seems to have only disadvantages compared to the previous solution:
  - An area of the external RAM has to be reserved for this buffer, reducing the maximum window size of the decompressor. As the standard deflate format specifies a window size of 32kB, this poses no real problems.
  - The additional external RAM activity is bidirectional, probably requiring a larger bandwidth. In the previous solution writing to the buffer came for free.

A combination of the above techniques can also be used.

6.5 Alternative Compression Formats

It is possible to choose a different compression format for the device. The current design is compatible with one existing LZH compression format. One can simplify the format to simplify the design. On the other hand one can make several extensions.

As demonstrated by the move from LHA to `gzip` it is comparatively easy to change the design to compress a different existing LZH-based compression format. The two formats are of comparable complexity. While `gzip` has slightly fewer symbols in the Huffman code for lengths and literals, the length class and address class extraction logic is more complex.

In order to make the design simpler to implement we can adapt the compression format. We can do the following things:

- Shrink the sliding window coder to an implementable size.
- Abandon compression options and remove the parameter RAM. This saves several registers inside modules (sliding window coder, buffering and statistics).
- Abandon second-level Huffman coding of code description to save processor cycles.
- Abandon Huffman coding for addresses, to save hardware in the statistics module and Huffman coder.

If we omit Huffman coding altogether we can dramatically simplify the architecture. There will be no need for any of the internal or external RAMs, the processor can be replaced by a simple controller (it does not need to be fast) and the Huffman coder can be replaced by a simple (hard wired) prefix encoder and decoder. The buffering and statistics module can be completely eliminated. Sliding window decompression can be implemented by the sliding window coder or alternatively by an external RAM.

Instead of adapting the compression format to simplify the design, we can make the design more complex in order to accommodate several formats. The following changes would have to be made in order to implement nearly all existing static LZH-based formats. These include static LHA, PKZIP implode format the 'deflate' format and probably ARJ.

- The sliding window coder must have options to set the minimum match length (2 or 3) and to eliminate self-matching. It does not need to accommodate large sliding windows or very long substrings.
- The sliding window decoder must take care of large substring lengths (512 at least). It already takes care of sliding windows of 64kB, which is adequate.
- The internal processor needs a loadable program memory.

- The statistics module must be prepared to categorize lengths and addresses into several classes. The following provisions must be made with regard to lengths:
  - No classes, each length has its own counter (256 different lengths)
  - Up to 32 non-uniform classes.
  - No classes up to certain length plus one class for greater lengths.

  The following provisions must be made with regard to addresses:
  - Up to 128 uniformly sized classes.
  - Up to 32 non-uniform classes.
- The Huffman coder and decoder needs the following provisions:
  - Provision for all bit/byte order conventions.
  - Class provisions as in the statistics module.
  - Encoding provisions as described below. The compressed data contains several of the following items:
    * Bit to distinguish literal from length/address pair.
    * Huffman (or fixed prefix code) encoded literal.
    * Huffman (or fixed prefix code) encoded length or length class.
    * Huffman (or fixed prefix code) encoded symbol that is either literal or length class.
    * Extra length-class dependent bits.
    * Huffman (or fixed prefix code) encoded address class.
    * Extra address-class dependent bits.

    The encoder and decoder must be able to process these items in virtually any order. Non Huffman encoded items can be catered for by filling the tables with fixed values.

Instead of static Huffman coding it would be desirable to use dynamic Huffman coding as it combines a good compression ratio with a low latency. The use of arithmetic coding instead of static Huffman coding must be considered. For non-adaptive multi-symbol arithmetic coding there exist extremely fast implementations [PS93].

6.6 Alternative Interfacing

The bus interface is another area in which alternatives are possible.

It is possibly to simplify the bus interface by omitting some features. One can omit the bus width option and allow only 32-bit DMA. One can omit end-of-section commands. Instead of the 16-bit counters one can rely on the counters of the DMA controller and the EOP signal. In this case one still needs 2-bit counters to determine the exact number of bytes in a 32-bit DMA session.

It is also possible to add more features to the interface. It would be desirable to have a dedicated port to transfer the compressed data without loading the main system bus. This port could be directly connected to a specially designed peripheral. Instead of a non-standard port one could also add a SCSI bus interface to the device so it could directly transfer the compressed data to or from a disk or tape device.

6.7 Reusable Parts

The following parts of the device could be reused in other designs:

- The bus interface can be used in a large class of intelligent coprocessors, such as data compression and encryption devices and protocol processors. If the device is to be used as a peripheral in a computer system and if its task is processing of a high-volume data stream, then the bus interface may be usable.
- The internal processor is optimized for high-speed processing tasks involving simple 16-bit integer operations and arrays. These tasks are mainly sorting and lookup table construction for other hardware. These tasks could be useful in Huffman coding and decoding and in protocol processing.
- Associative cell arrays were designed for sliding window coding and dynamic Huffman coding. Similar arrays may be useful in other speed-critical implementations of highly parallelizable algorithms.

Chapter 7

Conclusions and Future Research

First we can conclude that the compression device can be made, that it will work at the specified speed and with a reasonable compression ratio as specified by the requirements. The architecture described in this thesis is sufficient for persons skilled in the art of chip design to design the actual lay-out for this device.

The per chip cost is estimated at $500 to $600 for a window size of 8kB. The power dissipation is estimated at 4 Watts at a sliding window size of 8kB. Both cost and power dissipation are dominated by the sliding window coder, so it is possible to reduce both by decreasing the size of the sliding window coder, in exchange for a somewhat lower compression ratio. A smaller window size is believed to be more cost-effective in most situations. Cost and power dissipation are acceptable for many applications.

As is the case with all chips the fabrication process yields a certain percentage of defective chips. It is possible to test the chip for any defects. If the device is not defective it executes a deterministic algorithm that compresses data losslessly. The algorithms can be proved to be correct and the compression method has been used in software implementations for a long time without any problems. Data that is compressed with the device can be decompressed by existing software so that it remains accessible when the chip fails.

The device can be used within existing computer systems. It is possible to mount it on a PC card. If a software driver is integrated into the operating system, compression can be used transparently by the applications. Its use of 2 DMA channels can be problematic in an Intel-based PC architecture. The chip can be used with an ISA bus, but then it cannot work at the specified speed. Therefore it should be used with a PCI bus instead.

The compression method is two-way compatible with the 'deflate' method as used by the existing `gzip` program. This compatibility is not affected by the size of the sliding window coder. Versions of the chip with different sliding window coder sizes are mutually compatible. When the available number of transistors per chip increases, a new version with a larger sliding window can be made without affecting compatibility in any way.

From the above statements we conclude that the device meets its requirements as stated in the initial problem statement and the requirements chapter.

The following parts of the design are new inventions:

- The cell array for the sliding window coder, especially the use of non-greedy matching in it.
- The Huffman decoding algorithm optimized for canonical codes.
- A fast hardware implementation of dynamic Huffman coding (is not part of the current design).
- The integration of all algorithms for the LZH compression method in one device, all matched in speed.

The architectural framework leaves room for many alternative solutions as described in the previous chapter. It can be adapted to many LZH based compression methods. No real research is needed to do this. Improvements in chip technology will make larger sliding windows possible.

Future research should go into the following directions:

- Implement the dynamic Huffman coding algorithm in hardware so that it can be used instead of static Huffman coding. The use of arithmetic coding should also be considered.
- Design a chip that is programmable in such a way that many LZH-based compression methods can be handled by the same device.

Bibliography

[BCW89] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, New Jersey., 1989.

[Ben93] L.C. Benschop. A Design Methodology for Information Processing Systems. Technical report, Instituut Vervolgopleidingen, 1993.

[Ben96] L.C. Benschop. 100MBit per Second VLSI Implementation of Sliding Window Coding. In *Data Compression Conference*, page 424. IEEE Computer Society Press, Los Alamitos, California, 1996.

[Bha95] J. Bhasker. *A VHDL Primer (revised edition)*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[BW94] Michael Burrows and D.J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Digital SRC Research Report 124, electronic distribution, ftp://ftp.digital.com/DEC/SRC/research-reports/SRC-124.ps.gz, 1994.

[Cam95] J.M.A. Camps. Design of a Universal Bus Interface for DMA-using Peripherals. Master's thesis, Eindhoven University of Technology, 1995.

[Cra95] David J. Craft. ALDC and a pre-processor extension, BLDC, provide ultra fast compression for general-purpose and bit-mapped image data. In *Data Compression Conference*, page 440. IEEE Computer Society Press, Los Alamitos, California, 1995.

[Deu96a] L. Peter Deutsch. DEFLATE Compressed Data Compression Format Specification version 1.3. Electronic distribution, various ftp sites, document RFC-1951., 1996.

[Deu96b] L. Peter Deutsch. GZIP file format specification version 4.3. Electronic distribution, various ftp sites, document RFC-1952., 1996.

[DG96] L. Peter Deutsch and Jean-loup Gailly. ZLIB Compressed Data Format Specification version 3.3. Electronic distribution, various ftp sites, document RFC-1950., 1996.

[DV92] Dinant Dondertman and Jeroen Vermeeren. Ontwerp van een Contents Addressable Memory in SID en op transistor-niveau (in Dutch). Master's thesis, Eindhoven University of Technology, 1992.

[DW94] R.M. Davies and I.H. Witten. Compressing Computer Programs. In *Data Compression Conference*, page 469. IEEE Computer Society Press, Los Alamitos, California, 1994.

[Fen93] Peter M. Fenwick. Ziv-Lempel Encoding with Muliti-Bit Flags. In *Data Compression Conference*, pages 138–147. IEEE Computer Society Press, Los Alamitos, California, 1993.

[FG89] E.R. Fiala and D.H. Greene. Compression with Finite Windows. *Communications of the ACM*, 32(4):490–505, 1989.

[FRT96] Peter Franaszek, John Robinson, and Joy Thomas. Parallel Compression with Cooperative Dictionary Construction. In *Data Compression Conference*, pages 200–209. IEEE Computer Society Press, Los Alamitos, California, 1996.

[Gai93] Jean-Loup Gailly. GZIP version 1.2.4 (a data compression program). Computer software, electronic distribution, ftp://prep.ai.mit.edu/pub/gnu/-gzip-1.2.4.tar, 1993.

[Gal78] Robert G. Gallagher. Variations on a Theme by Huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978.

[GL96] A. Gavish and A. Lempel. Match Length-Functions for Data Compression. *IEEE Transactions on Information Theory*, 42(5):1375–1380, 1996.

[GN96] Jean-Loup Gailly and Mark Nelson. *The Data Compression Book (second edition)*. M&T Books, New York, 1996.

[Gro91] H.F. Groot. Data Compression of Computer Files in Hardware. Master's thesis, Eindhoven University of Technology, 1991.

[HCPAG94] Brian Ta-Cheng Hou, Craig D. Cohen, J.A. Pasco-Anderson, and Michael Gutman. Associative CAM apparatus and method for variable length string matching. US Patent 5329405, 1994.

[Hed95] Marco Heddes. *A Hardware/Software Codesign Strategy for the Implementation of High-Speed Protocols*. PhD thesis, Eindhoven University of Technology, 1995.

[Hor95] R. Nigel Horspool. The Effect of Non-Greedy Parsing in Ziv-Lempel Compression Methods. In *Data Compression Conference*, pages 303–311. IEEE Computer Society Press, Los Alamitos, California, 1995.

[HP87] Derek J. Hatley and Imtiaz A. Pirbhai. *Strategies for real-time system specification*. Dorset House, New York, 1987.

[HS94] Daniel S. Hirschberg and Lynn M. Stauffer. Parsing Algorithms for Dictionary Algorithms on the PRAM. In *Data Compression Conference*, pages 136–145. IEEE Computer Society Press, Los Alamitos, California, 1994.

[Huf52] D.A. Huffman. A Method for the Construction of Minimum Redundancy Codes. *Proceedings of the IRE*, 40:1098–1101, 1952.

[Jon92] S. Jones. 100Mbit/s Adaptive Data Compressor Design Using Selectively Shiftable Content Addressable Memory. *IEE Proceedings-G, Circuits, Devices and Systems*, 139(4):498–502, 1992.

[Knu85] Donald E. Knuth. Dynamic Huffman Coding. *Journal of Algorithms*, 6:163–185, 1985.

[LC90] Ming-I Lu and Chang-Fuu Chen. A Huffman-Type Code Generator with Order-N Complexity. *IEEE Transactions on Acousitcs, Speech and Signal Processing*, 38(9):1619–1626, 1990.

[Lee62] C.Y. Lee. Intercommunicating Cells — Basis for a Distributed Logic Computer. In *Proc. Fall Joint Computer Conf., Philadelphia, Pa.*, 1962.

[Lee76] J. van Leeuwen. On the Construction of Huffman Trees. In *Automata, Languages and Programming*. Edinburgh University Press, 1976.

[LG91] Didier Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(4):46–58, 1991.

[LH90] Lawrence L. Larmore and Daniel S. Hirschberg. A Fast Algorithm for Optimal Length-Limited Huffman Codes. *Journal of the ACM*, 37(3):464–473, 1990.

[LP63] C.Y. Lee and M.C. Paul. A Content Addressable Distributed Logic Memory with Applications to Information Retrieval. *Proceedings of the IEEE*, pages 924–932, 1963.

[LR82] G.G. Langdon and J. Rissanen. A Simple General Binary Source Code. *IEEE Transactions on Information Theory*, 28(5):800–803, 1982.

[MNM91] A. Mukkerjeee, Ranganathan N., and Bassiouni M. Efficient VLSI Designs for Data Transformation of Tree-Based Codes. *IEEE Transactions on Circuits and Systems*, 38(3):306–314, 1991.

[MSWB93] A. Moffat, N. Sharman, I.H. Witten, and T.C. Bell. An Emprical Evaluation of Coding Methods for Multi-Symbol Alphabets. In *Data Compression Conference*, pages 108–117. IEEE Computer Society Press, Los Alamitos, California, 1993.

[Oku90] Haruhiko Okumura. AR002 (a data compression program). Computer software, electronic distribution, ftp://ftp.funet.fi/pub/msdos-/programming/compression/lds_10.zip, 1990.

[Par94] Terry J. Parks. Incremental search content addressable memory for increased data compression efficiency. US Patent 5369605, 1994.

[PH90] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach (second edition)*. Morgan Kaufmann, San Francisco, 1990.

[PML93] M. Patel, N. MacLean, and G. Langdon. An Economical Hardware-Oriented High-Speed Data Compression Scheme. In *Data Compression Conference*, page 444. IEEE Computer Society Press, Los Alamitos, California, 1993.

[PS93] Harry Printz and Peter Stubley. Multialphabet Arithmetic Coding at 16 MBytes/sec. In *Data Compression Conference*, pages 128–137. IEEE Computer Society Press, Los Alamitos, California, 1993.

[Ris83] J.J. Rissanen. A Universal Data Compression System. *IEEE Transactions on Information Theory*, 29(5):656–664, 1983.

[SA85] M. Schroeder and B.S. Atal. Code Excited Linear Prediction (CELP): High Quality Speech at Low Bit Rates. In *International Conference on Acoustics, Speech and Signal Processing*, pages 937–940. IEEE, 1985.

[Sew96] Julian Seward. BZIP, a block-sorting lossless data compressor. Computer software, electronic distribution, http://www.cs.man.ac.uk/arch/-people/j-seward, 1996.

[SRM90] J.A. Storer, J.H. Reif, and T. Markas. A Massively Parallel VLSI Design for Data Compression Using a Compact Dynamic Dictionary. In *IEEE VLSI Signal Processing Conference*, pages 329–338. IEEE Press, 1990.

[SS82] J.A. Storer and T.G. Szymanski. Data Compression via Textual Substitution. *Journal of the ACM*, 29(4):928–951, 1982.

[Sto88] James A. Storer. *Data Compression, Methods and Theory*. Computer Science Press, 1988.

[Sto92] James A. Storer. *Image and Text Compression*. Kluwer Academic Publishers, 1992.

[Tho92] Clark Thoborson. The V.42bis Standard for Data Compressing Modems. *IEEE Micro: Chips, Systems, Software and Applications*, 12(5):41–53, 1992.

[TO92] Yooichi Tagawa and Masaru Oki. LHA for UNIX, version 1.00 (a data compression program). Computer software, electronic distribution, ftp://ftp.win.tue.nl/pub/compression, 1992.

[VBW89] R. Veldhuis, M. Breeuwer, and R. van der Waal. Subband Coding of Digital Audio Signals Without Loss of Quality. In *International Conference on Acoustics, Speech and Signal Processing*, pages 2009–2012. IEEE, 1989.

[Ver90] A.C. Verschueren. *IDaSS for ULSI, Interactive Design and Simulation System for Ultra Large Scale Integration*, 1990.

[Ver92] A.C. Verschueren. *An Object Oriented Modelling Technique for Analysis and Design of Complex (Real-Time) Systems*. PhD thesis, Eindhoven University of Technology, 1992.

[Ver96] A.C. Verschueren. Informal communication, 1996.

[Vit87] J.S. Vitter. Design and Analysis of Dynamic Huffman Codes. *Journal of the ACM*, 24(6), 1987.

[Vol96] P.A.J. Volf. Text Compression Methods based on Context Weighting. Technical report, Stan Ackermans Institute, 1996.

[Wal91] Gregory K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34(4):30–44, 1991.

[WB90] I.H Witten and T.C. Bell. Text Compression Corpus. Set of computer files, electronic distribution, ftp, 1990.

[Wel84] T.A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, 17(6):8–19, 1984.

[Wie93] René Wientjes. Informal Communication, 1993.

[Wie95] R.J.M. Wientjes. Implementing Abstract Data Types in Hardware. Technical report, Instituut Vervolgopleidingen, 1995.

[Wij96] A. Wijffels. Design of an Embedded Microprocessor for Array Intensive Tasks. Master's thesis, Eindhoven University of Technology, 1996.

[Wil91] Ross N. Williams. *Adaptive Data Compression*. Kluwer Academic Publishers, Boston, 1991.

[WM85] Paul T. Ward and Stephen J. Mellor. *Structured development for real-time systems*. Yourdon Press, London, 1985.

[WNC87] Ian H. Witten, Radford M. Neil, and John G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30:520–541, 1987.

[WOF+96] Kel D. Winters, Patrick A. Owsley, Catherine A. French, Robert M. Bode, and Peter S. Feeley. Adaptive data compression system with systolic string matching logic. US Patent 5532693, 1996.

[WST95] F.M.J. Willems, Yu.M. Shtarkov, and Tj.J. Tjalkens. The Context-tree Weighting Method: Basic Properties. *IEEE Transactions on Information Theory*, 41(3):653–664, 1995.

[Yok92] H. Yokoo. Improved Variations Relating the Ziv-Lempel and Welch Type Algorithms for Sequential Data Compression. *IEEE Transactions on Information Theory*, 38(1):73–81, 1992.

[ZL77] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[ZW90] R. Zito-Wolf. A Systolic Array for Sliding Window Data Compression. In *IEEE VLSI Signal Processing Conference*, pages 339–351. IEEE Press, 1990.

Appendix A

Compression Algorithms

This chapter describes the algorithms that can be used to implement the compression methods mentioned in this thesis. These compression methods use a combination of two methods, sliding window coding and Huffman coding. Both methods and the algorithms to implement them are briefly described here.

The sliding window algorithm reads the input byte steam sequentially and converts it to a stream of literal bytes and length/address pairs that indicate substrings of the previous input stream. Only the last N bytes of the input stream are searched for substrings. The next output of the sliding window coder is either a literal or a length plus an address. The stream of literals, lengths and addresses is converted to a compressed byte stream in such a way that it can be unambiguously decoded. Literals, lengths and addresses can be Huffman compressed to gain extra compression.

Fist we give formal meanings to the notions 'compression format', 'compression method', 'compression algorithm' and 'compression implementation'.

At the highest level of abstraction, data compression methods can be described by functions that transform byte strings into other byte strings. The compression function turns the input data (a byte string) into the compressed data and the decompression function reconstructs the original input data from the compressed data.

Let B be the set of 8-bit bytes. B^* is the set of strings of 8-bit bytes of length zero or more. A compression method is described by the compression function f_c and the decompression function f_d.

$$f_c : B^* \to B^*$$

$$f_d : B^* \to B^* \cup \{\perp\}$$

where $\perp$ is a special value outside the set B^*, that that indicates that the function argument was not the result of the compression of any string. In other words, the decompression function can be applied to malformed compressed data, in which case the result is an error. The compression function, on the other hand, must be applicable to each and every possible byte string and produce a valid compressed byte string as a result.

Compression is supposed to be lossless. We formulate the lossless criterion as follows.

$$\forall_{s \in B^*} f_d(f_c(s)) = s$$

This implies that f_c is an injective function.

We also desire that f_c does indeed compress. Therefore we want

$$\forall_{s \in T} |f_c(s)| < |s|$$

where T is a large class of byte strings that are considered 'typical' input data for a compressor. It is supposed that the large majority of the English texts, computer programs and database files are in T.

It is desirable that the compressed data contains an indication of its own length, so that the compressed data may be stored as a number of fixed sized blocks, where the last block contains extra bytes not originally put there by the compressor. This feature of the compression format can be formulated as follows.

$$\forall_{s,t \in B^*} f_d(s) \neq \perp \Rightarrow f_d(s) = f_d(s.t)$$

The key notions of this and the next chapter are defined as follows:

- A *compression format* is a description of the compressed data and how it should be decompressed into the original data. The compression format uniquely describes and is uniquely described by f_d.

- A *compression method* uniquely describes what compressed data will result from each input. It also describes the compression format. The compression method uniquely describes and is described by both f_c and f_d. Note that f_c alone does not uniquely describe the compression method. f_c does specify the function values of f_d for arguments in the range of f_c, but it does not do so for arguments outside its range.

- A *compression algorithm* is an algorithm that can be used for all or part of the computations necessary for the implementation of a compression method.

- A *compression implementation* is a concrete hardware or software implementation of a certain compression method or class of compression methods that uses certain compression algorithms. The program `dcompr` (see appendix C, comparison of compression methods) and the VLSI circuit under design are both compression implementations.

A.1 Sliding Window Coding

The sliding window compression method belongs to the class of dictionary-based algorithms, also called Lempel-Ziv algorithms. Dictionary-based methods operate on a sequential stream of uncompressed bytes and convert them into a stream of codes.

Dictionary-based methods generally work as follows:

- There is a set of byte strings, which is called the dictionary. This set contains strings of bytes that previously occurred in the uncompressed data stream or that are likely to occur in the future. The single-byte strings are always part of the dictionary. All strings can be identified by a code.
- When data is compressed, the uncompressed byte stream is divided into finite length strings that match strings in the dictionary. Those matching strings are replaced by their dictionary codes.
- Whenever a string is converted, the dictionary is updated. Zero or more strings are added to it and zero or more other strings are deleted from it. This updating operation depends only on the data processed thus far, so that the decompressor can reproduce it.
- When the data is decompressed, the codes are looked up in the dictionary and are converted back to the original input byte strings.
- The decompressor has the same initial dictionary as the compressor and it updates its dictionary in exactly the same way as the compressor. Therefore the dictionary need not be conveyed separately.

With sliding window compression the dictionary consists of all substrings of the last N uncompressed bytes that are processed thus far. The minimum length of these substrings must be M_{min} and the maximum length is M_{max}. In addition, all one-byte strings are always part of the dictionary (the literals). The last N uncompressed bytes are kept in a FIFO data structure. Whenever a matching string is found and encoded, its bytes are shifted into that FIFO structure. The updating operation is thus very simple. The code of each substring consists of its position in the FIFO and its length. The code of each single-byte string is a literal code, which is uniquely distinguishable from an address/length pair. If more maximum length substrings can be found, the one with the address closest to the head of the FIFO is selected. This, and certain statistical properties of most data lead to an uneven frequency distribution of addresses. Special address encoding can take advantage of this.

The original sliding window algorithm [ZL77] is a bit different from the above description. With this variation each code consists of an address, a length and a literal. Each code represents a (possibly zero length) substring from the sliding window buffer plus the byte following it. None of the popular compression programs use this particular variation of sliding window coding because it offers less compression than the more frequently used method with separate codes for literals and substrings.

A.1.1 Variations of the Method

The primary parameters that characterize various sliding window methods are the buffer size N, the minimum acceptable substring length M_{min} (which is often 2 or 3) and the maximum acceptable substring length M_{max}. Of course there are numerous variations in the way how the sliding window coder output is further encoded.

Some less obvious variations in the sliding window coding process itself are:

initial window contents Before compression starts, the buffer contains something that is not part of the input data. Some compression methods declare these buffer contents undefined and substrings of the undefined buffer part are therefore unacceptable. Other compression methods define these contents as zero (or something else) and they permit substrings of this as sliding window coder output.

self matching Some compression algorithms place the input bytes in the sliding window buffer as matching is in progress. Therefore it could occur that the input matches part of itself. Take the following example. Suppose that the next input is "`ABAB`" and the buffer contents are

```
XXXXXXAB
```

After the first two bytes are matched, the buffer contains

```
XXXXABAB
```

And now the whole substring "`ABAB`" can be found in the buffer. Some methods allow this particular behavior, others do not.

nonexhaustive search Some (most) software implementations do not always find the longest matching substring due to certain speed improvements in the hashing or tree update algorithms.

nongreedy matching Greedy matching is not optimal. Finding optimum substrings (in the general case) would require dynamic programming which is prohibitive [FG89]. Variations of non-greedy matching are discussed in [Hor95]. A one-byte lookahead already offers significant improvement and nearly all modern software implementations use it.

The lookahead algorithm works as follows:

- Find the longest matching substring (which we will call the 'first' string).
- Skip the next input byte and find the longest matching substring one byte ahead. We will call this the 'second' string.
- If the second string is longer, encode the skipped input byte as a literal and use the second substring, else encode the first substring.

The following example shows the working of lookahead. Suppose that the next input is "`ABCDEFG`" and the buffer contains:

```
ABCBCDEFG
```

The longest matching substring is "ABC" of length 3. But if we skip the "A" in the input, we can match the string "BCDEFG" of length 6. We decide to encode "A" as a literal and "BCDEFG" as a 6-byte substring. This would take less space than the two shorter substrings that would have been found with greedy matching.

Most software implementations will not directly encode the second substring if they find that it was longer than the first string, but they will apply lookahead to it iteratively.

A.1.2 Existing Implementations.

Decompression of sliding window encoded data is straightforward. In software the FIFO can be implemented as a circular buffer and in hardware it can be implemented either as a circular buffer using ordinary RAM or as an addressable hardware FIFO. The one task in compression that is not straightforward is finding matching substrings efficiently.

No practical implementations of sliding window use linear search in the buffer. Most of the software implementations use a variation of one of the following methods:

- All substrings of the minimum length M_{min} are stored in a hash table. Of these strings a hash code is computed both when they are inserted into the table and when they are looked up. Only the strings that share the same hash code need to be searched. Of these the substring that has the longest match with the input is used. A disadvantage of this method is the need for extra memory to store the hash table and the possibility of degenerate behavior. Suppose that the uncompressed data contains a large block of zero bytes. Then the buffer will contain a very large number of identical substrings, that have the same hash code. If the search algorithm looks at all of them, it will be very slow, if the number of substrings that are searched is limited, suboptimal compression results. The program gzip [Gai93] uses this algorithm and it has a parameter to select the search limits, so that a trade-off between speed and compression ratio can be made.

- A search tree is maintained for all substrings in the sliding window buffer. They typically contain one node for each position in the sliding window buffer. These data structures occupy much memory and comparatively complex algorithms are necessary to update them. For each input byte one node is added and one node is deleted. In [FG89] a possible algorithm is described. These methods usually take more time than hashing but there is no degenerate behavior. Search trees range in complexity from simple binary trees to suffix tries and Patricia trees. Algorithms using Patricia trees run in an amortized linear time.

Both of these methods use more memory for the search data structures (hash tables, trees) than for the sliding window buffer itself. The above-mentioned algorithms are designed for software implementation. They are optimized for speed with the constraint that all operations are performed sequentially.

Storer [Sto88, Sto92] designed several hardware implementations of the sliding window algorithm with systolic arrays. For each byte in the sliding window buffer (or for every two bytes) there exists a cell in the systolic array. Each cell contains several registers. These algorithms can perform sliding window compression at one input byte every few clock ticks. Three variations are proposed:

- The match tree algorithm. This is described in depth in [Sto88]. It employs an array of $2N$ cells, each of which is capable of containing two characters (bytes) and a pointer (substring description). One of the characters is part of the old sliding window contents, the other one is part of the input that must be matched. The operation is very complex. The array cells are at the leaves of a tree. Signals from the cells travel upwards in the tree, then downwards to determine the optimal match.

- The broadcast reduce tree algorithm. This is briefly described in [Sto92]. It employs an array of N cells, each of which is capable of containing one character. The array is at the leaves of two trees, one to distribute (broadcast) the next input character to all array cells and one to collect the match results. Apart from the presence of the broadcast tree and the way the match results are collected this algorithm looks remarkably similar to the proposed hardware sliding window coder with greedy matching.

- The wrap architecture. This is described briefly in [Sto92] and an implementation is described in [ZW90]. It employs $N/2$ cells, each capable of containing two characters and a pointer. Input characters first travel in one direction through the array, then in the opposite direction. The two characters in each cell are compared to each other. This is a true systolic array without additional trees.

A.1.3 Proposed Hardware Algorithm.

In 1993 René Wientjes [Wie93], professor Stevens and I invented a different algorithm, that I described in [Ben93]. This algorithm also uses one cell per byte in the sliding window buffer, but these cells are much simpler than those described by Storer.

In the proposed hardware algorithm, each cell stores a byte and a flag. All cells access a common 8-bit bus, the match bus on which the next input byte is presented. If this is the first byte of a string, all cells set their flag if their stored byte matches the byte on the match bus. For the following bytes of the string, each cell sets its flag only if it was already set and if its own byte matches the byte on the bus. After each byte matching operation the bytes stored in the cells are shifted to the next cell and the input byte is shifted into the first cell. The flags are not shifted. All cells output their match result and these outputs are fed to a priority encoder to determine the address of the first matching substring. If no more matches occur, the address of the last match (and the match length) are output and then the cells match the input byte as the first byte of a new string. This data compressor combines a hardware FIFO with a content addressable memory. There is a small central controller. The concept of a match flag in a parallel string searcher was first described by Lee [Lee62].

Note that:

- The match result is output by the cells and fed to the priority encoder, is the logical 'and' of the contents of the match flag and the 'equal' output of the comparator.
- There is a global signal that determines whether the current byte is the first byte of a new string or a subsequent byte. This signal determines the flag setting behavior of each cell. This signal is derived from the logical 'or' of the match flags of all cells.

To further explain this algorithm look at the following example. Suppose that $N = 16$ and the array already contains data. The next input is `ABCX`. First the array looks as follows.

Flags	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bytes	A	B	D	C	A	B	C	D	E	F	A	D	X	Y	Z	Z
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Now the first byte `A` of a new string is matched. A cell sets its flag if it matches.

Flags	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
Bytes	A	B	D	C	A	B	C	D	E	F	A	D	X	Y	Z	Z
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Next the byte (`A`) is shifted in.

Flags	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
Bytes	B	D	C	A	B	C	D	E	F	A	D	X	Y	Z	Z	A
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Next the second byte `B` of the string is matched. A cell can only clear its flag if it does not match; it cannot set its flag.

Flags	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Bytes	B	D	C	A	B	C	D	E	F	A	D	X	Y	Z	Z	A
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The new byte (`B`) is shifed in.

Flags	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Bytes	D	C	A	B	C	D	E	F	A	D	X	Y	Z	Z	A	B
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Next the third byte `C` is matched. Again a cell clears its flag if it does not match.

Flags	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Bytes	D	C	A	B	C	D	E	F	A	D	X	Y	Z	Z	A	B
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

This byte (`C`) is shifted in.

Flags	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Bytes	C	A	B	C	D	E	F	A	D	X	Y	Z	Z	A	B	C
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Finally the byte `X` is matched. As no cells can match this byte as part of the current substring, the current substring is output (address 11, length 3). `X` is the start of a new substring.

Flags	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
Bytes	C	A	B	C	D	E	F	A	D	X	Y	Z	Z	A	B	C
Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

It is possible to extend the hardware sliding window coder with one extra flag bit per cell so that it performs one-byte lookahead. It cannot iteratively apply lookahead to the second substring, it must use it right away. This hardware lookahead is slightly inferior to the iterative lookahead method.

To implement lookahead in hardware, each cell has two match bits, one indicating a match for the 'first' string and one indicating a match for the 'second' string. All cells have one match output bit that either gives information about the first string or about the second string, depending on an input signal from the central controller. The central controller sends two global signals to all cells. The first signal indicates whether the central controller wants information about the first string or the second string (the flag setting behaviour of the cells depends on this signal as well). This signal is independent of the result of the match outputs of the current cycle. The second signal indicates whether the current string continues to match and it does depend on the match outputs of the current cycle.

Each cell internally generates an equality signal, which is true if the stored byte is equal to the input byte on the bus. The flag setting behaviour of the cells is as follows, depending on these signals:

first string, not continued The first match bit is set to equality signal, the second match bit is 'and'-ed with the equality signal. This is the situation in which the first string ceases to match. A new first string is started, the second string is continued. During the next cycle, the 'second string' signal will be active.

first string, continued The first match bit is 'and'-ed with the equality signal, the second match bit is 'and'-ed with the equality signal. The first string still matches, update the match bits for both the first and the second string.

second string, not continued The first match bit is 'and'-ed with the equality signal, the second match bit is set to the equality signal. The second string matches no longer. A new first string was started during the previous cycle and it is continued. A new second string is started.

second string, continued The second match bit is copied to the first match bit, then the second match bit is set to zero. The second string still matches while the first string ceased to match already during the previous cycle. From now the second string is treated as the first string; there is no string to replace the second string though. This is the situation in which the second string is longer than the first string. The central controller outputs the first byte of the old first string (that it saved in a register) as a literal and the second string is used.

The match output of each cell only depends on the first/second string signal:

first string Output the logical 'and' of the first match bit and the equality signal.

second string Output the logical 'and' of the second match bit and the equality signal.

The match outputs of all cells are collected, the logical 'or' is made available and a priority encoder generates the address the same way as with the greedy algorithm. The 'continued' signal is the logical 'and' of the 'or' of all match outputs and some validity checks. The address must be within the window as defined by the N parameter and if matching against the uninitialized part is not allowed, it must also be less than the total number of bytes already shifted in. The length must be less than M_{max}.

A.1.4 Encoding the Sliding Window Output.

The codes generated by the sliding window coder must be converted to a stream of bytes in such a way that they can uniquely be decoded. Each output of the sliding window coder is either a literal or a length/address pair and first we need a way to distinguish these in the encoded data. This can be accomplished in two ways:

- Precede each code by one bit that indicates if it is a literal or a substring code. Depending on the value of that bit the decoder can decode either a literal or an address plus a length.

- Make one set that contains the literals and the lengths. The decoder can decode the next length or literal and if the decoded value is a length it decodes the address following it.

The first method is a special case of the second method provided that the encoded length precedes the encoded address. This is, if the combined length/literal code of the second example encodes all literals with an initial bit of 0 and all lengths with an initial bit of 1, then an encoding arises that is equivalent to the first method. The second method is more general and it is expected to yield a slightly improved compression ratio. Variations of the first method that are not a special case of the second method are possible. Fiala and Greene [FG89] describe an encoding for either strings of literals or substrings.

The literals, lengths and addresses themselves can be encoded in a number of ways:

- Fixed length codes. All literals, lengths and addresses are encoded using a fixed number of bits. If there are 256 literals, 256 different lengths and 8192 different addresses, than all literals are encoded in 8 bits, all lengths in 8 and all addresses in 13 bits.
- Variable length codes. Small lengths are more likely to occur in compressed data than large lengths. Similarly addresses near the head of the FIFO occur most frequently. One can encode lengths and addresses in a variable number of bits so that more frequent codes have shorter code lengths. These codes are fixed for all data and are not constructed for each file being compressed. Therefore it does not make sense to encode literals in a variable number of bits as the frequency distribution of these is inherently data dependent. Several variations are discussed in [Fen93].
- Huffman Codes. Optimum prefix codes are constructed depending on the actual frequency distributions of lengths, literals and addresses. Static and dynamic coding are possible.
- Arithmetic codes. With arithmetic coding one starts with the real number interval $[0, 1)$. This is subdivided in a finite number of subintervals, one for each symbol, where the size of each subinterval is proportional to the symbol probability. This subinterval is again subdivided into subintervals for the second symbol, etc. After a number of codings, one has a small subinterval. The compressed data is a binary fractional number, represented with enough bits to uniquely identify the required subinterval. An introduction is found in [WNC87].

It is possible that addresses, lengths and literals are each encoded in a different way, though it is hard to combine arithmetic coding with one of the other three methods. Lengths and literals are usually considered one set and they are encoded with a single method. If they are encoded using Huffman codes there is one Huffman tree for them. Addresses have a code independent of lengths and literals. If they are encoded using Huffman codes, they use a different Huffman code from the lengths and literals. In [MSWB93] variations of Huffman coding and arithmetic coding are discussed.

As the buffer size N in a practical sliding window coder is a few thousand, it is often not practical to maintain a Huffman code of all addresses. Therefore most implementations use a system of address classes. The address range is divided into a number of classes, either of a fixed size or of a variable size. An address is represented as the class code and an offset within the class. The class code is then Huffman encoded and the offset is coded unchanged. If the classes are of fixed size, all address offsets are coded with the same number of bits. If the classes are of variable size, the offsets are encoded with a variable number of bits, depending on the class.

Thus a literal is represented by an encoded literal and a substring is represented by an encoded length, followed by an encoded class number, followed by an address offset. In most compression methods lengths and literals are encoded by the same code and the decoder can determine from the decoded value whether it had decoded a literal or a length. In some compression methods the length/literal code also contains special symbols for end of block or end of file. Some compression methods use a class-offset system for lengths as well. The length/literal code then encodes the set of all literals united with the set of all length classes.

A.2 Static Huffman Coding

Huffman codes are prefix codes that have short code words for frequently occurring code symbols and long code words for infrequently occurring code words.

Static Huffman coding is a two-pass compression method. First all of the data are collected and frequency statistics are compiled of it. From these frequency statistics the Huffman code is computed. This code is transmitted or stored with the compressed data and finally the data is compressed with the Huffman code. Therefore the data are read twice in the process, one time when collecting statistics and one time for the actual coding.

A.2.1 Buffering and Frequency Statistics

The sliding window compressor delivers the lengths, literals and addresses as a stream. As static Huffman coding is a two-pass process this data must be saved so that it can be read again later. When the data enters the buffer, statistics are collected.

Statistics collection involves an array of counters, one for each distinct symbol. When a symbol is stored in the buffer, the counter that belongs to that symbol is incremented. If a class-offset system for addresses or lengths is used, the class must be decoded and the counter belonging to that class must be incremented.

As a memory buffer is of finite size, the sliding window coded data is divided into large blocks, each of which fits into the buffer. Each block has its own statistics and its own Huffman codes. Each compressed block is preceded by its own Huffman code descriptions. Alternatively one can collect one block of data for statistics and use the same Huffman code for all the data that follows. If the data that follows has different statistical properties from the first block, compression will be bad. Furthermore, the Huffman codes must have a code word for every possible symbol, even if that did not occur in the first data block. Therefore this variation is rarely used.

A.2.2 Tree Generation

The Huffman algorithm constructs an optimal prefix code from a list of symbol frequencies. This code is represented as a tree. In pseudo code the basic Huffman algorithm can be written as follows:

```
store all symbols with their frequencies in a list;
while(list has more than one element) do
begin
 remove two nodes A and B with smallest frequency from list;
 create tree node that has A and B as children
                      and the sum of their frequencies.
end;
```

One could implement the Huffman algorithm in the following ways:

heap data structure The most frequently used implementations of the Huffman algorithm represent the list of nodes with a heap data structure. A heap is a balanced binary tree represented in an array, such that the root is at position 1 and the children of node k are at position $2k$ and $2k + 1$. Further every node has a lower frequency than any of its children. The root node has thus the lowest frequency. Construction of a heap of size n takes $O(n \log n)$ and deletion of the minimum or replacement of the minimum by a different node takes $O(\log n)$. The whole Huffman algorithm therefore takes $O(n \log n)$

hardware sorting In hardware one could implement the node list as an associative self-sorting array. The array must support insertion and deletion of the minimum. Each cell contains a 'less than' comparator that is used for insertion at the correct location. Insertion and deletion can take one cycle each. In this case the Huffman algorithm takes $O(n)$

linear algorithms with presorting If the symbols are already sorted by frequency, it is possible to implement the Huffman algorithm in $O(n)$ in software. The easiest way to do this is maintaining two separate node lists, one that contains the symbols and their frequencies and one FIFO that contains the combined nodes. The overall minimum frequency node is either the next symbol in the list or at the tail of the FIFO (combined nodes are inserted with monotonously increasing frequencies). This was first described by [Lee76]. Sorting the symbol frequencies can be performed very fast using a variation of radix sort (also called bin sort or distribution sort), if the symbol frequencies are small integers. Radix sort breaks the

$O(n \log n)$ barrier by using part of the number as a multi-way selector instead of using ordinary comparisons.

An alternative to the sort algorithm is maintaining the statistics in sorted order on the fly as they are collected. For each symbol stored in the buffer, only one frequency count is incremented by one and given a suitable data structure the sorted list can be updated in $O(\log n)$ cycles.

In reality the goal of the Huffman algorithm is mostly not a tree, but a list of the code word lengths of all symbols. There are two reasons for this:

- This list is what is transmitted with the compressed data. If the compressor and the decompressor possess this list, they can both generate the same code from it. This can be done in $O(n)$ time. The code words can be generated directly without the use of a tree. In most cases this code is different from the original Huffman code, but it offers the same compression anyway because the lengths of all code words are the same. A code like this is called a canonical code.

- Most implementations of coding and decoding do not use a tree but they use the code words directly. Maintaining a tree is a costly operation and it should be avoided if possible.

The algorithm described by Lu and Chen [LC90] can also generate a Huffman code in $O(n)$ from a list of sorted symbol frequencies. It inserts the combined nodes at the right position in the list, but it maintains a large gap to avoid moving large amounts of data with every insertion. What makes this algorithm interesting is that it completely avoids the use of a tree. During the first phase (in which nodes are combined) a list of expansion indices is created. During the second phase the number of code words for each code word length is computed from the expansion indices. From this, either the code words themselves or the code word lengths are trivially computed.

The algorithm uses the following data structures:

- Queues Q1 and Q2 for nodes. Each node contains only the frequency count (The list of sorted symbols is stored separately). In the original paper the queues were both implemented with a single array of size $2n$ and several pointers. Originally the leaf nodes are contained in Q1 and during the execution of the algorithm they are moved to Q2. Combined nodes are also inserted into Q2. Nodes are taken from Q2 to form new combined nodes.

- An array of expansion indices E_1 to $E_n - 1$. This array describes the topology of the Huffman tree. For each combined node i, E_i is the number of leaf nodes encountered earlier minus the number of combined nodes encountered earlier minus 1.

- An array of length counts A_1 to A_{16}, in case the maximum code word length is 16. Each entry stores the number of code words of a given length.

The algorithm consists of the following steps:

Contraction phase In this stage the Huffman algorithm itself (node combining) is performed. Instead of a tree, the array of expansion indices is generated.

```
Put the leaf nodes into Q1 from low to high frequency;
Take first 2 nodes from Q1;
sum := sum of frequencies of these nodes;
level := 0;
i := 1;
while (Q1 and Q2 together have at least two nodes) do
begin
 while (tail of Q1 has frequency not greater than sum
        and Q1 not empty) do
 begin
  Copy node from Q1 to Q2;
  level := level + 1;
 end;
 Put sum node into Q2;
 E[i] := level;
 i := i + 1;
 If Q2 has one node then
 begin
  copy one node from Q1 to Q2;
  level := level + 1;
 end;
 Take two nodes from Q2;
 sum := sum of frequencies of these nodes;
 level := level - 1;
end;
```

Expansion phase In this phase the number of code words of each length in array A are computed from the expansion indices. It is assumed that the number of symbols n is at least 3.

```
hilength := 2;
A[1] := 1;
A[2] := 2;
i := n - 3;
while( i > 0 ) do
begin
 if E[i] < A[hilength] then
 begin
```

```
    A[hilength] := A[hilength] - 1;
    hilength := hilength + 1;
    A[hilength] := 2;
   end
   else
   begin
    A[hilength-1] := A[hilength-1] - 1;
    A[hilength] := A[hilength] + 2;
   end;
   i := i - 1;
  end;
```

The array A can easily be manipulated to impose a code word length restriction on the tree. If one code word of length i is removed, two of length $i+1$ can be added. By repeated application of this principle one can create enough code words with a length below the limit. Note that there are formal algorithms to create optimal codes with a limited code word length [LH90], but an ad-hoc strategy seems to be more practical. In practice it occurs very rarely that a Huffman algorithm yields code words longer than 15 (the limit for gzip).

Code word generation phase The symbols are traversed in order of decreasing frequency and each one is assigned a code word length in increasing order. A_i indicates how many symbols are assigned length i. The code words themselves could be assigned during this phase as well, but if a canonical code is used, this is postponed.

A.2.3 Encoding

Using the Huffman tree it is easy to implement Huffman coding one bit at a time. Start at the leaf that belongs to the encoded symbol and traverse the tree to the root, recording the branch directions that one came from. These branch directions must be output in the reverse order as the code bits.

It is more efficient to store all code words and their lengths in an array indexed by symbol. One can then retrieve the code word in just one operation. It is now only necessary to convert the variable length bit fields to a stream of bytes. To do this efficiently one needs a device that contains a variable-distance shifter (a barrel shifter).

A.2.4 Decoding

Huffman decoding can be performed one bit at a time using the Huffman tree. One starts at the root of the tree and one traverses the tree to a leaf node, using the next compressed bit as the direction.

One can perform Huffman decoding one symbol at a time in the following ways:

- Store all code words in a content addressable memory with a mask register in every cell. A cell that contains a code word of length 3 will have a mask register with only 3 bits set and it will report a match if only the three first bits match. As the code is a prefix code, only one cell will report a match. This CAM will take much space and its lay-out would need to be hand-designed.
- Use a array that is indexed by the greatest possible code word length. If this greatest code word length is 16 (as is often the case) the array would have to store 65536 symbols and their code word lengths. This is often too much and this RAM must be filled too, which takes time.
- Use an array that is indexed by the first k bits of the compressed data, where k is a chosen such that the RAM will be reasonably small and that most code words are under k bits. In this case most code words are decoded in one cycle. If a code word is longer than k bits, the array location refers to a tree, which must be traversed one bit at a time from the $k+1$th bit. This is a reasonable compromise between bit-serial decoding and fully parallel decoding. Must software programs use this decoding algorithm. In a worst-case situation most of the code words have a length greater than k and nearly the whole Huffman code must be represented as a tree. This can take up to $2N$ memory places for the decoding tree. Most practical software implementations use this method.
- Use an array that is indexed by the first k bits of the compressed data as above and make use of the special (canonical) nature of the Huffman tree to represent it more compactly. In this particular application the Huffman tree is chosen such that from left to right (left branches correspond to bit 0, right branches correspond to bit 1) one encounters code words in monotonously increasing length order. Therefore, seeing the next input bits as a binary number the code word length increases with the number.

  If one decodes a code with N symbols of maximum length m where k bits are directly decoded one needs the following storage:

  – 2^k words of $\lceil \log_2 N \rceil$ bits each, for direct symbol decoding.
  – 2^k words of $\lceil \log_2 m \rceil$ bits each, for the code word lengths of the directly decoded symbols.
  – $m - k$ words of $\lceil \log_2 N \rceil$ bits each for the number of symbols having a code word length from $k + 1$ to m.
  – N words of $\lceil \log_2 N \rceil$ bits each for the symbols that have code words greater than k, in left-to-right order.

  The algorithm works as follows:

  – Collect the next k bits of input data and use it as a binary number.

- Use this binary number to directly index into the first two RAMs and directly decode the symbol and find its length.
- If this succeeded (symbol found with code word of at most k bits) then the decoding is finished.
- If this did not succeed, use a special bit-serial decoding algorithm to dynamically find the correct code word and symbol. This works as follows:

```
Incoming := next m bits of input data;
Codeword := bit pattern of first code word with
                    bit length greater than k;
Increment := 2 ^ (m-k);
n1 := number of code words with length k+1;
n2 := 0;
Length := k + 1;
while Codeword + Increment * n1 < Incoming do
begin
 Codeword := Codeword + Increment * n1;
 Increment := Increment div 2;
 n2:=n2+n1;
 Length := Length + 1;
 n1:= number of code words with code word
              length equal to Length;
end;
n2 := n2 + (Incoming-Codeword) div Increment;
Decoded symbol is in symbol list at index n2,
                  its code word length is Length;
```

This algorithm uses less storage than the previous one, but it has increased computational complexity.

Conversion from a byte stream to the variable length bit fields works as follows:

- First collect at least enough bits to decode the maximum length code word.
- Decode the next code word and determine its length.
- Take away as many bits as indicated by the length.

A.2.5 Transmission of Tree Descriptions

As said earlier each Huffman encoded block is preceded by a description of the Huffman codes. In most compression formats this description takes the form of a list of code word lengths. For each possible symbol the code word length is transmitted (or 0 if there is no code word for that symbol).

Code words can be seen as binary numbers. The following algorithm converts the code word lengths to the actual code words.

```
Order the symbols by increasing code word length;
Increment := 2 ^ (maximum code word length - 1);
Codenumber := 0;
For each code word with nonzero length in order
                              of increasing length do
begin
 If next length > previous length then
           Increment := Increment div 2;
 Assign Codenumber to next symbol;
 Codenumber := Codenumber + Increment;
end;
```

Such code descriptions are often very well compressible. Runs of zeros are converted to special codes and then the data is Huffman compressed. Therefore a compressed data block often consists of the following parts in this order:

- Code description of Huffman code that encodes the code description.
- Huffman encoded code description of Huffman code that encodes the data.
- Huffman encoded data.

In appendix B.1 (compression format) this is described in greater detail.

A.3 Dynamic Huffman Coding

With dynamic Huffman coding the Huffman tree is adjusted to the new symbol frequency distribution each time a symbol is encoded or decoded. Thereby the Huffman tree is gradually adapted to the symbol frequency distribution. A description of the Huffman tree need not be transferred separately and no large buffers are needed to store a complete block of data for two-pass processing. A great advantage in some applications is that there is no long delay between data input and the output of the compressed data.

In [Gal78] an algorithm for dynamic Huffman coding is presented. This algorithm is based on the so-called sibling property of a well-formed Huffman tree. A binary tree has the sibling property if:

- each leaf node has a count associated with it that corresponds to the symbol frequency and each other node has a count that is the sum of the counts of its children.

- all nodes can be put in a list in order of increasing counts, such that all sibling pairs (the two children of the same node) are next to each other in the list.

It can be proved that all Huffman trees have the sibling property and that all trees that have the sibling property are Huffman trees.

One can start with a Huffman tree where all symbols have a count of 1. After encoding or decoding each symbol the tree is adapted, which takes $O(\log_2 N)$ steps. This is far more efficient that constructing a Huffman tree from scratch, which takes $O(N \log_2 N)$ (or $O(N)$)steps. The tree is stored in a list ordered by symbol count. Internal nodes are also stored along with the sum of the counts of their children. Siblings in the tree are stored next to each other in the list . This is possible because of the sibling property of Huffman trees. After a symbol is coded or decoded, the count of the leaf node corresponding to it is incremented by one. If this causes the list to go out of order, the node is exchanged with the last node with a count equal to the unincremented count of the node, both in the list and in the tree. Then the count of the parent node of the (possibly moved) node is incremented, which may give rise to another node exchange. The counts of all nodes on the path from the leaf node to the root are incremented. This adaptation operation leaves the sibling property of the new code invariant. Therefore the adapted code is a Huffman code (and therefore an optimal code) for the changed symbol frequency distribution.

in figure A.1 the adaptation algorithm is illustrated. There are four symbols: A, B, C and D. The left tree corresponds to a frequency distribution of A=1, B=1, C=1 and D=2. Internal nodes e, f and g are also shown. The list of nodes ordered by count is shown below the tree. It can be seen that siblings in the tree are adjacent in the list. Suppose that the symbol D is encoded. Now the count of D must be increased by 1. The count of D is now higher than that of e, so D and e must be exchanged. After this, the count of D's new parent g is also incremented. The right tree is the updated tree.

The algorithm can be implemented using a doubly linked tree whose node records are kept in an array ordered by count. Additionally there is a table to look up the leaf node that corresponds to a given symbol. In order to encode, the tree is traversed from leaf to root. This yields the bits in reversed order, so they must be reversed before transmission. In order to decode, the tree is traversed from root to leaf, using each bit to decide whether to take the left child or the right child. Both operations take at least as many memory references as there are bits in the code. In order to adapt, the tree is traversed from leaf to root, thereby incrementing counts and exchanging nodes, whereby the pointers must be adjusted properly. This takes at least two memory references per bit in the code for each traversal and count increment. Each node exchange will take several more memory references.

Rather than starting with an initial tree where each symbol is included with frequency 1, one can introduce an escape code as an extra symbol (see [Knu85]). Initially the tree only contains the escape code and each time a symbol is coded that does not yet occur in the tree, it is preceded by the escape code. Then it is added to the tree.

As counts are constantly incremented, a finite precision counter will eventually over-

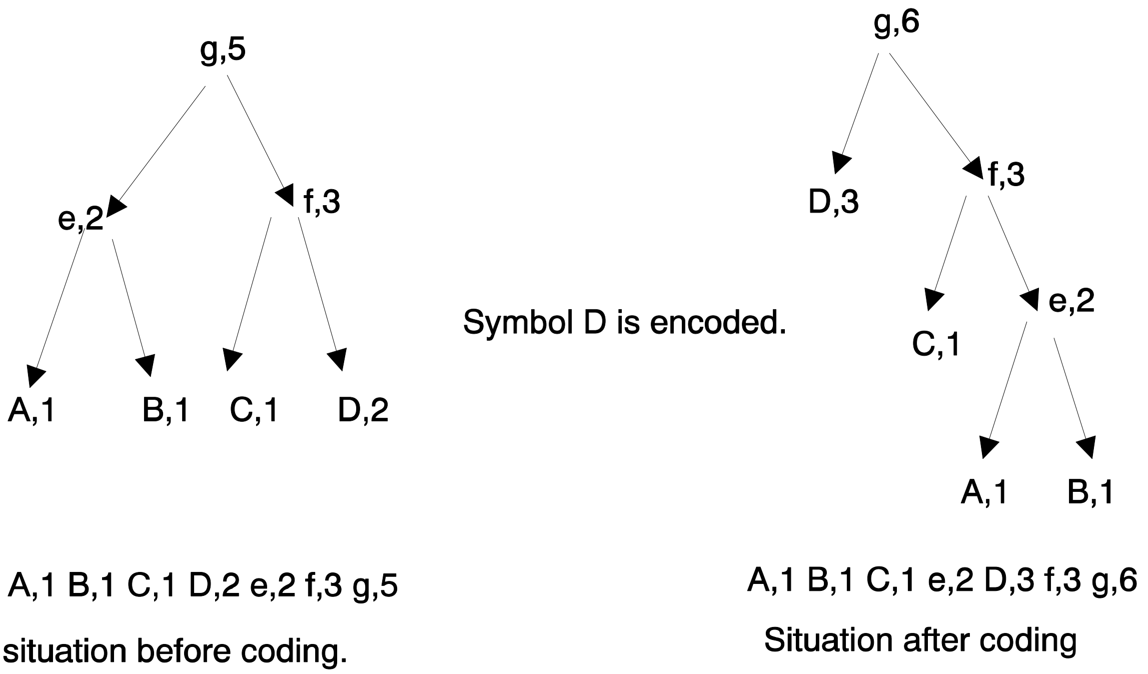

Figure A.1: Example of Tree Adaptation with Dynamic Huffman Coding.

flow. The root node has a count equal to the sum of the counts of all symbols, so it has the largest count. Therefore it overflows first. When this occurs, several solutions are possible:

- All counts are divided by two and then a new Huffman tree is computed. It is necessary to recompute the Huffman tree from scratch as due to rounding the order of node counts may change considerably.
- All counts are reset to 1 and the starting Huffman tree is used.
- All counts are frozen and no adaptation takes place from now.

A.3.1 The Dynamic Huffman Algorithm in Hardware.

The algorithm described in the previous section could be implemented in software or it could be implemented in hardware using the same data structures, but dedicated state machines. In this case it could perform one memory reference per clock cycle. Now consider a dynamic Huffman coder for both lengths and literals. Assume an average code word length of 5 and an average string length of 4. Suppose that the implementation codes and decodes in one cycle per bit and that it performs the adaptation (without swaps but with count updates) in one clock cycle per bit. Assume that there is one swap in every five lengths or literals and that the swap takes 6 extra cycles (very optimistic).` Then on average 12 cycles per length or literal are needed. As the sliding window encoder can process one input byte per clock cycle, one length or literal is delivered every

4 cycles. The dynamic Huffman encoder is slow by a factor of 3. This is a rather optimistic situation. Data statistics are not always as favourable as this and the number of expected cycles for all operations is chosen very optimistically, as it is assumed that complete tree nodes can be accessed in one memory reference and that a count can be incremented at once. These figures apply to an optimized hardware architecture. A software implementation of the same method would be an order of magnitude slower.

One of the bottle necks of the algorithm is that it works bit-serially. There is a way to encode and decode static Huffman codes at one symbol per clock cycle. It employs a memory that contains both the code word and the length or a mask word with as many bits set as there are in the code word. Encoding is straightforward. Decoding can be done using a content-addressable memory with a mask word for each cell. Therefore if 101 is a code word and a pattern starting with 101 is fed to the CAM, only the cell that contains 101 (with only 3 bits significant) will match.

The idea of Huffman coding one word at a time instead of one bit at a time could be extended to dynamic Huffman coding as well. The idea is as follows:

- There is an array of identical CAM cells, that are interconnected by buses (see figure A.2).
- For each node of the Huffman tree (both leaves and internal nodes) a CAM cell is reserved (see figure A.3).
- Each CAM cell contains a flag to indicate that it is a leaf node, the bit pattern of the code (as the bit pattern itself and a mask), a count and the character (if it is a leaf node).
- All cells are physically ordered by count as they would be in the sibling probability list.

The operation of the system would be roughly as follows:

- Encoding and decoding can be done in one cycle. Only leaf nodes take part in such operations. For encoding we match on the node number, for decoding we match on the code word.
- After coding and decoding we can decide where to do the first swap (if any) in one cycle. All cells containing prefixes of the given code (whose counts should be incremented) check if their counts are equal to the count of the adjacent cell.
- Any node swap takes place in 3 more cycles, which will be described later. Note that actual code words are stored in cells instead of references to nodes. Many cells may have to change the first part of their code words. They do so in parallel.
- After a swap a test cycle is performed to determine where to do the next swap.

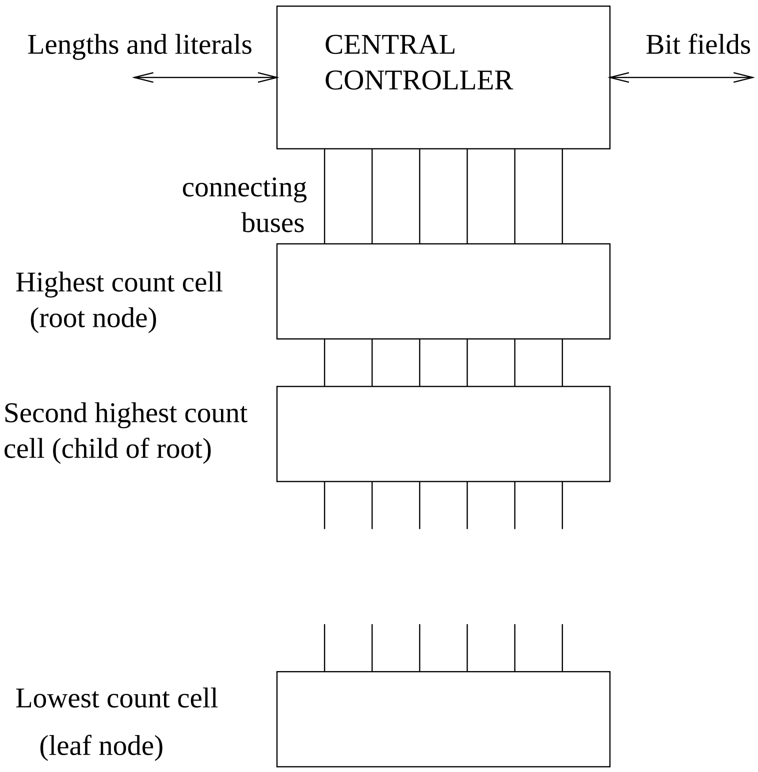

Figure A.2: The Dynamic Huffman Coding Array.

It seems that it is possible to code and decode in 2 clock cycles per word and that each swap takes 4 more cycles. This is far more efficient than could be hoped for any bit-serial implementation.

Next the detailed workings of the dynamic Huffman coder are described. Only the encode/decode/update behaviour for dynamic Huffman codes is described. How the cells are initialized, is yet to be determined. The cell structure can probably be used for static or dynamic initial tree computation. It can certainly be used for static Huffman encoding and decoding, if properly initialized.

The dynamic Huffman coder has as many cells as there are nodes in the Huffman tree. For the length/literal Huffman code in LHA there are 512 code words, so there are 1023 nodes in the tree.

The following buses interconnect all cells (see figure A.3):

- A 9 bit wide bus carries the symbol being encoded or decoded. This bus is called CHARBUS.
- A 15 bit wide bus carries the frequency count for a node. It is called COUNTBUS.
- A 1 bit wide bus carries a flag that indicates that a node is a leaf. It is called LEAFBUS.
- A 16 bit wide bus carries the code word. It is called CODEBUS.

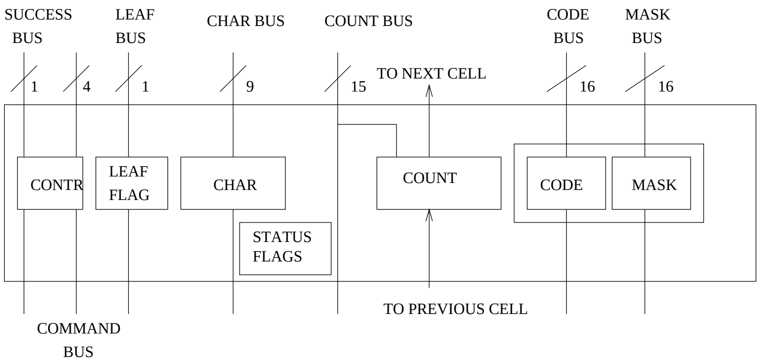

Figure A.3: Single Cell of the Dynamic Huffman Coder.

- A 16 bit wide bus carries the code mask. A 1 indicates that the corresponding bit in the code word is not part of the code word. This bus has a wired 'and' function, e.g. if more cells drive it, the bus carries the logical 'and'. This bus is called MASKBUS.
- A 1 bit wide bus carries a signal that indicates that some condition appeared. It is called SUCCESSBUS. It has a wired 'or' function.
- A 4 bit wide bus, COMMANDBUS, carries the command that is given to all cells. At least six commands are needed for normal operation. Add to this a no-op and several commands to initialize and read the array.

Each cell has access to the COUNT register of its higher neighbour cell. There are connections between neighbouring cells for that.

The cells have the following registers:

CHAR is 9 bits wide. It contains the symbol that is encoded by the codeword contained in that cell. Only leaf nodes have a meaningful value in the CHAR register.

COUNT is 15 bits wide. It contains the count of the node (the number of times the node has been used). All leaf nodes are initialized to a count of at least 1 and the count is incremented by 1 each time the node is used. Non-leaf nodes have a count that is the sum of the counts of their children.

LEAF is 1 bit wide. It is a flag that indicates whether the node is a leaf node.

CODE is 16 bits wide. It contains the code word. The length of the code word (at most 16) is determined by the MASK register in the same cell. The code word of any node is determined by the path in the Huffman tree from the root node to that node. Going to the lefthand child is a 0-bit, going to the righthand child is a 1-bit. The root node itself has a zero-length code word (all bits in the mask register are set).

MASK is 16 bits wide. It determines the length of the corresponding code word. The leftmost bits (as many as there are bits in the code word) are zero, the other bits are 1. Note that during a node swap operation the CODE/MASK registers do not exactly conform to the above description.

In addition to the registers mentioned above the cell has several status flags that are used during an update action:

MARKINC indicates that the count in the cell must be incremented.

TOOLOW indicates that the cell is too low (position in the list) to be suitable for a node swap.

MARKA and **MARKB** indicate that the cell is either of the two nodes that take part in a node swap (A-cell and B-cell).

MARKAR and **MARKBR** indicate that the cell is a descendant of one of the nodes that are to be swapped. The cell will then receive a new code prefix.

All cells are (physically) ordered by count. They are stored in such a way that all sibling pairs in the Huffman tree are in neighbouring cells. This is the order indicated by the sibling property of Huffman trees. When the cells are initialized, they must be initialized such that this is true. An update action will preserve the truth of the abovementioned proposition.

Apart from the cells there is a global controller that sends the appropriate commands to the cells through the command bus. It has a set of CHAR, CODE, BCODE MASK and LEAF registers for temporary storage during node swaps.

Apart from the encode and decode cycles, the behaviours during encoding and decoding are the same. The operation starts with a code or decode cycle.

- **Encode cycle** The central controller puts the symbol to be encoded on the CHARBUS. A cell whose LEAF flag is set and whose CHAR register matches the value on CHARBUS, puts its CODE and MASK registers on the CODE and MASK buses respectively. Any cell that had its MARKINC flag set, increments its count. All cells clear their MARKINC and TOOLOW flags.

- **Decode cycle** The central controller puts the next 16 input bits on the CODEBUS. A cell whose LEAF flag is set and whose CODE (as many bits as indicated by its MASK register) matches the value on the code bus, puts its MASK and CHAR registers on the MASKBUS and CHARBUS respectively. Any cell that had its MARKINC flag set, increments its count, All cells clear their MARKINC and TOOLOW flags.

After either an encode cycle or a decode cycle, the controller has the CHAR, CODE and MASK values of the leaf node corresponding to the code word.

If the count register of the root node equals 32766, then the tree is recomputed, How this is done is not described here. Otherwise the normal update procedure is started. This procedure starts with a test cycle to determine where the next swap operation (if any) will be performed. If not, the next cycle can be an encode or decode cycle again. If a swap is required, it is performed in 3 cycles. The swap operation is followed by a test cycle. Swap operations and test cycles are repeated until the test cycle indicates that no more swap operations are needed.

The following actions are performed during the update operation:

- **Test cycle** The central controller puts the contents of its CODE register on the CODEBUS and all 1s on the MASKBUS. All cells clear their MARKA, MARKB, MARKAR and MARKBR flags, unless stated otherwise in this description.

  Each cell that has a code that agrees to the code on the CODEBUS (according to the cell's own mask), whose TOOLOW flag is not set and whose COUNT is equal to the COUNT of the next higher neighbour, sets SUCCESSBUS high (wired or) and puts its MASK register on the MASKBUS (driving certain bits low). If such a cell senses that no other bits on the MASKBUS are driven low (by other cells), it sets its MARKA flag. There is only one cell that sets its MARKA flag, called the A-cell. The A-cell emits its CHAR, LEAF and COUNT registers on the CHARBUS, LEAFBUS and COUNTBUS respectively.

  Each cell that has a code that agrees to the code on the CODEBUS (according to the cell's own mask), whose TOOLOW flag is not set and whose MASK register has more zero bits than there are zero bits on the maskbus[1], sets its MARKINC flag. The root node cell sets its MARKINC flag anyway and it never sets SUCCESSBUS or its MARKA flag.

  If SUCCESSBUS is set by any cell, then each cell whose code agrees to the code on CODEBUS (according to the mask on MASKBUS and whose own MASK has more zero bits than the mask on the MASKBUS, prepares itself for B-reception. It does this by setting its own MARKBR flag and by 'or'-ing its mask register with the complement of the mask on the MASKBUS. The mask register now starts with one or more 1-bits indicating the size of the code in the A-cell, followed by one or more 0-bits (indicating the size of its own code) followed by zero or more 1-bits. Note that a cell who marks itself for B-reception always has a longer code that the A-cell.

  The central controller reads the CELL, LEAF and COUNT values of the A-cell from their buses into its registers.

  If SUCCESSBUS was set, then the three exchange cycles are performed, otherwise the next encode or decode cycle is performed.

[1] In this case and the similar case in exchange cycle #1, this can be determined by a logical operation on the masks. The bit counts need not be determined and compared.

- **Exchange cycle #0** The count from the A-cell is put on the COUNTBUS.

  Each cell whose own count is equal to that on the COUNTBUS and whose next higher neighbour has a count larger that that, sets its MARKB flag. Further it emits both its CODE and MASK registers on the CODEBUS and MASKBUS There is one and only one such cell, called the B-cell. It is never the root node cell. The B-Cell also sets its MARKINC flag.

  The central controller reads the CODE value of the B-cell into its BCODE register.

- **Exchange cycle #1** The cell that has its MARKB flag set, puts the contents of its CHAR, LEAF, CODE and MASK registers on the buses.

  The A-cell loads its CHAR and LEAF registers from the buses (put there by the B-cell).

  Each cell that does not have its MARKBR flag set and whose code agrees to the code on CODEBUS (according to the mask on MASKBUS and whose own MASK has more zero bits than the mask on the MASKBUS, prepares itself for A-reception. It does this by setting its own MARKAR flag and by 'or'-ing its mask register with the complement of the mask on the MASKBUS.

  All cells, that have either MARKAR or MARKBR set, or whose mask has more zeros than the mask on the MASKBUS, will set their TOOLOW flag.

  Each cell whose MARKBR flag is set, reads the new code prefix (of the B-cell) from the CODEBUS. First it shifts its CODE and MASK registers one place to the left, one place to the right or not at all, depending on the size difference between the CODE in the B-cell and the code in the A-cell. The size of the code in the B-cell is determined by the value on MASKBUS, the size of the code in the A-cell is determined by the length of the first block of 1-bits in the MASK register of the cell[2].

  The shift operation makes the code word either one bit shorter, or leaves its size unchanged or makes it one bit longer. If the code word length would exceed 16, SUCCESSBUS is set. The first block of 1-bits in the MASK register is cleared and the new code prefix is loaded into the first bits of the CODE register. The other bits in the CODE register are unaffected.

  If the central controller finds SUCCESSBUS set, one of the code words has got a size greater than 16, which cannot be handled by the hardware. In this case the update operation is aborted and the tree is recomputed.

- **Exchange cycle #2** The central controller puts its CODE, MASK, CHAR, and LEAF registers (all from the A-cell) on the buses.

  The B-cell loads its CHAR and LEAF registers from the buses.

[2]This size difference can be computed using certain logical operations on the bit masks. The difference can have three values: -1, 0 or 1.

Each cell whose MARKAR flag is set, reads the new code prefix from the CODEBUS. See the remarks about how the MARKBR cells read the B-cell code prefix,

If SUCCESSBUS is set due to one of the code words having a length greater than 16. the central controller aborts the update operation and recomputes the Huffman tree.

The central controller copies its BCODE register to its CODE register.

The update algorithm for the hardware implementation is exactly the same as that of the conventional software implementation if viewed at the abstract tree level. Therefore both implementations yield exactly the same compressed data. The hardware and software implementation differ in how the tree is represented.

A few remarks about the hardware algorithm:

- All cells whose count should be incremented are marked for increment during the test cycles and the actual incrementation is done during the subsequent encode or decode cycle.
- The conventional software implementation visits the nodes from leaf to root, exchanging nodes if necessary. The hardware implementation visits all nodes between any necessary exchanges in parallel, marking each one for increment (performed during the test cycle). The exchanges themselves are performed one at a time.
- When an exchange operation is performed the node currently visited (the A-node) is exchanged (in the tree) with the last node (in the list) that has an equal count. The nodes are moved along with all their descendants in the tree. In the software algorithm the exchange involves only local pointer exchanges.

  In the hardware implementation all cells contain (in their CODE and MASK) registers the encoded path in the tree from the root to that node. Therefore all descendants of the A-cell must replace the A-part of their code word with the B code word and vice versa. The A and B cell may have code words of different sizes, so that the descendants must shift the rest of their code words.

  According to [Vit87], the A-node is at a tree depth that is either one higher or the same or one lower than the depth of the B-node, if there is no zero-count node (there is not one in this implementation). Therefore the A-cell has a code word that is either one longer, or the same length or one shorter than the code word of the B-cell. Therefore the cells that must change their A-code prefix to the B-code prefix or vice versa, need only shift the rest of the code word one bit in either direction or not at all. This simplifies the shift hardware per cell considerably.
- The TOOLOW flag in each cell is set when the cell should not be marked for increment or checked for being the A-cell in any subsequent test cycle in the current update operation . It prevents descendants of the former B-cell (which should

not be visited) from being selected as the new A-cell or from being marked for increment.

This algorithm has been implemented as a C program where all cells are simulated at the register level. It was tested on several files and it delivered exactly the same compressed data as the conventional software implementation. The conventional software implementation was used to perform all other tests as it runs far more efficiently.

Due to the data dependencies between the steps of the algorithm we currently believe that a faster implementation of dynamic Huffman coding is practically infeasible.

Appendix B

Format of the Compressed Data.

B.1 The LHA-based format of `dcompr`

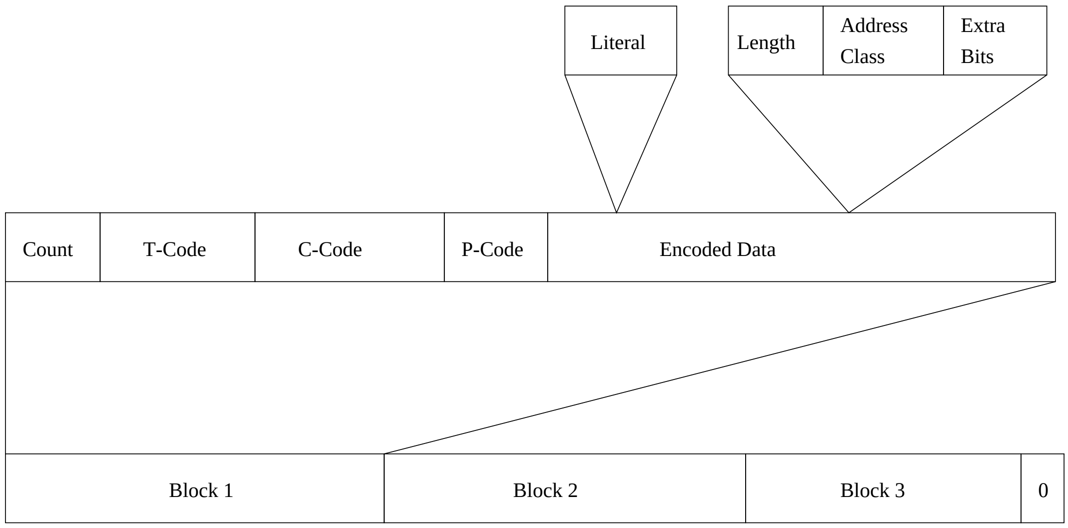

Figure B.1: Overview of Compressed Data Format with Static Coding.

This section describes the class of compression formats used by the program `dcompr` and by the VLSI circuit under design. By setting certain parameters, one selects a particular compression format. These parameters are summarized at the end of this section.

The uncompressed data consists of a sequence of 8-bit bytes. These bytes are processed by a sliding window encoder. The sliding window coder uses a window of a size that can be set by the user. The maximum size is 32768 bytes. The substrings have a minimum length of either 2 or 3 bytes and a maximum length which is settable by the user up to 256 bytes. This encoder outputs a sequence of lengths and literals. Each length is accompanied by an address. Literals are not accompanied by an address. If the maximum length is less than the minimum length, the sliding window coder outputs literals only.

The lengths and literals are represented as follows:

- The first 256 codes (0 to 255) are reserved for the 256 literal bytes.
- If an end of block code exists, it is represented by 256.
- If an end of section code exists, it is represented by the next consecutive code.
- The following consecutive codes represent the lengths from minimum substring length (either 2 or 3) to maximum substring length (maximum 256).

There may be no more than 512 codes in the length/literal code. Therefore it is not permissible to have an end of block code, an end of section code and lengths from 2 to 256. If sliding window coding is not used, the length/literal code encodes only literals and end of block and end of section codes.

The end of block code is present in the following cases, when:

- No Huffman coding is used at all (both addresses and lengths/literals are encoded by fixed codes.)
- Dynamic Huffman coding is used.
- One set of Huffman trees is used during the entire compression session.

In the other cases the compressed data stream consists of Huffman compressed blocks containing a count, a tree description and the compressed data.

The end of section code is only present if the end of section command is enabled. It only works properly with either no Huffman compression or dynamic Huffman compression.

Addresses are represented as follows. Address 0 means that the substring starts at the last position (the byte last shifted in), address 1 means the byte just before it. The highest address means that the substring starts at the fist position in the sliding window (the byte that will be shifted out first).

The compressed data consists of a sequence of 8-bit bytes. Logically this byte stream may be viewed as stream of bits, whereby the most significant bit in each byte comes first. With the exception of end of section codes and the beginning or end of a compression session, there are no gaps between bits. If a 3-bit code must be transferred and 7 bits of the current byte are already used, the most significant bit of the code is in the least significant bit of the current byte and the two least significant bits of the code are the two most significant bits of the next byte.

Length/literal codes can be represented as follows:

- Using a fixed code (see table B.1)
- Using a Huffman code. All permissible values (literals, lengths, end of block and end of section) have their own entry in the Huffman code. This code can be either dynamic or static.

If the length/literal code represents a length, it is immediately followed in the compressed byte stream by the representation of an address. Address codes can be represented as follows:

- Using a fixed code (see table B.2). This code is only defined for addresses up to 8191.
- Using a Huffman code over the most significant bits and coding the six least significant bits unchanged immediately after the Huffman encoded address in the compressed byte stream. Up to seven most significant bits are used for the Huffman code, resulting in a code of up to 128 code words. We refer to this as the type-1 Huffman code for addresses. The type-1 Huffman code is only defined for addresses up to 8191. This Huffman code is either static or dynamic.
- Using a Huffman code for the class number and transferring the rest of the address bits unchanged (see table B.3) This Huffman code can either be static or dynamic. It contains 14 code words if the window size is 8192 or less. If the window size is between 8193 and 16384, there are 15 code words and if the window size is between 16385 and 32768 there are 16 code words. We refer to this as the type-2 Huffman code for addresses.

Length	bits	code
1 (lit)	9	0 +8
end block	9	111111111
2 (or end sect.)	3	100
3	3	101
4–7	5	110 +2
8–15	7	1110 +3
16–31	9	11110 +4
32–63	11	111110 +5
64–127	13	1111110 +6
128–255	15	11111110 +7
256	9	111111110

Table B.1: Fixed Length/Literal Code.

If there is either no Huffman coding or dynamic Huffman coding, then the compressed data stream consists of encoded lengths/literals and addresses only, the last length/literal code is the end-of block code.

If there is static Huffman code with one set of trees for the entire compression session, then the compressed data stream starts with an encoded set of trees, followed by encoded lengths/literals and addresses. The last code is the end of block code.

Address	bits	code
0–511	10	0 +9
512–1023	11	10 +9
1024–2047	13	110 +10
2048–4095	15	1110 +11
4096–8191	16	1111 +12

Table B.2: Fixed Address Codes

If there is static Huffman code with a set of trees for each block, then the compressed data stream consists of a sequence of compressed blocks. Each block consists of the following:

- A 16-bit number that indicates the number of lengths and literals in that block. A value of 0 indicates the end of the compressed data.
- An encoded set of trees.
- The encoded lengths/literals and addresses. There is no end of block code. The end of the block is determined by the count that precedes the block.

As a very special case it is possible that only one length or literal value occurs in the block or one address class value occurs in the block. In that case there is no need to explicitly encode the lengths/literals or address classes of which it is known that there is only one value. Consider it as a degenerate Huffman code with one zero-length code word.

B.1.1 Tree Encoding

In all cases where static Huffman coding is used, the trees are represented very compactly.

Three Huffman codes are used, each of which has its own tree description that must be transferred:

- The T-code is used to encode the description of the C-code and, if the type-1 Huffman code for addresses is used, the P-code.
- The C-code is used to encode lengths and literals.
- The P-code is used to encode the addresses.

Table B.4 shows which of the three tree descriptions are transmitted in which case.

Each code description consists of a sequence of lengths of code words in the Huffman tree. These are the lengths of the code words for all of the encoded symbols from

Address	class number	extra bits
0	0	0
1	1	0
2–3	2	1
4–7	3	2
8–15	4	3
16–31	5	4
32–63	6	5
64–127	7	6
128–255	8	7
256–511	9	8
512–1023	10	9
1024–2047	11	10
2048–4095	12	11
4096–8191	13	12
8192–16383	14	13
16384–32767	15	14

Table B.3: Address Classes.

the lowest to the highest numbered symbol. A code word length of zero indicates that the corresponding symbol does not occur in the tree. The Huffman tree is constructed from that description with shortest code words to the left (0-branch) and longest code words to the right (1-branch). Therefore the all-zero code word is always one of minimum length and the all-one code word is always a maximum length code word. Code words of the same length are assigned the from left to right, from the lowest numbered symbol to the highest numbered symbol.

length/literal	Address	Code descriptions transmitted
fixed	fixed	(none)
fixed	type-2 Huffman	P-code(direct)
fixed	type-1 Huffman	T-code P-code(T encoded)
Huffman	fixed	T-code C-code(T encoded)
Huffman	type-2 Huffman	T-code C-code(T encoded) P-code(direct)
Huffman	type-1 Huffman	T-code C-code(T encoded) P-code(T encoded)

Table B.4: Code Descriptions Transmitted

The T-code encodes 19 different symbols: 16 lengths and three special codes to represent runs of zeros. The T-code description is built up as follows:

- A five-bit code indicates the number of symbols whose length is given. This includes the number of lengths skipped by the skip count.

  - If this code is zero, it is followed by another 5-bit code that indicates the symbol implicitly encoded by the only zero-length code word.
  - If the code is not zero it is followed by the lengths of the code words for symbols 0 through 2 encoded using the direct code of table B.5.
    * This is followed by a 2-bit code that indicates the number of code word lengths to skip (0–3) which are considered zero.
    * This is followed by the code word lengths for the remaining symbols, encoded using the direct code of table B.5

The C-code can encode up to 511 symbols. Its description is built up as follows:

- A 9-bit count indicates the number of symbols whose code word length is given. This includes the symbols covered by the runs of zeros encoded by the three special T-code symbols.
  - If this is 0, it is followed by another 9-bit code indicating the symbol implicitly encoded by the only zero-length code word.
  - Otherwise it is followed by a sequence of T-encoded length codes and special symbols (see table B.6).

symbol	code
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1 0
8	1 1 1 1 0
...	...
18	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0

Table B.5: Direct Encoding for T-code and P-code symbols.

The P-code for the type-2 address Huffman code encodes 14 symbols (or 15 or 16 as needed if the window size is larger than 8192). Its description is built up as follows:

- A 4-bit code indicates the number of symbols whose code word length is given. If the window size is greater than 16384, this code is 5 bits.

symbol	extra bits	meaning
0		Zero length (symbol does not occur)
1	4	Run of 3–18 zero lengths
2	9	Run of 20–531 zero lengths
3		Length 1
4		Length 2
...		...
18		Length 16

Table B.6: Meaning of T-code Symbols.

  - If this is 0, it is followed by a 4-bit code, indicating the symbol implicitly encoded by the only zero-length code word.
  - Otherwise it is followed by a sequence of code word lengths, encoded using the direct code of table B.5.

The P-code for the type-1 address Huffman code encodes up to 128 symbols. Its description is built up as follows:

- An 8-bit code indicates the number of symbols whose code word length is given. This includes code words covered by the special run of zero length codes
  - If this is 0, it is followed by another 8-bit code, indicating the symbol implicitly encoded by the only zero-length code word.
  - Otherwise it is followed by a sequence of code word lengths, encoded using the T-code.

B.1.2 Dynamic Huffman coding

The length/literal Huffman code and the address Huffman code are implemented using two independently working dynamic Huffman coders. A the start of each compression session the length/literal Huffman coder is initialized using the frequency distribution shown in table B.7. When the type-1 address Huffman coder is used, it is initialized using the frequency distribution shown in table B.8. When the type-2 address Huffman code is used, it is initialized with a count of 1 for each symbol.

From the initial frequency distribution the initial Huffman tree is constructed as follows:

- All symbols are inserted in the list in the order of symbol number.
- At each step the minimum frequency item and the second minimum frequency item are removed from the list. An internal tree node with the sum of the frequencies of the two removed items is inserted into the list. The new tree node connects

with a 0-branch to the first removed item and with a 1-branch to the second removed item.

- Items with the same frequency are removed in the same order as they were inserted.
- The Huffman tree is complete when there is only one item in the list (the root node of the Huffman tree).

Symbol	Count
Length 3	60
Length 4	30
Length 5	20
Length 6	15
All others (literals, other lengths, end of block flush)	1

Table B.7: Initial Frequency Distribution of Lengths and Literals.

Address prefix	Count
0	32
1	10
2	10
3	6
4	6
5	6
All others	1

Table B.8: Initial Frequency Distribution of Addresses in the Type-1 Address Huffman Code.

Each time a length or literal is encoded the length/literal code is updated; each time an address is encoded or decoded, the address code is updated. Adaptation of the Huffman tree is performed using the algorithm described in section A.3 (dynamic Huffman coding).

The tree is recomputed when either of two events happens as a result of a tree update (before any subsequent encoding or decoding):

- The count of the root node of the Huffman tree reaches 32767 (One knows this if this count equals 32766 before the update operation).

- One or more code words have got a length greater than 16 bits.

A tree recomputation is carried out as follows:

- The frequency count of each symbol is divided by 2 and rounded up to the nearest integer.
- The tree is completely recomputed from that modified frequency distribution using the same assumptions as the initial tree construction.

B.1.3 Selectable Parameters

This section summarizes the user selectable parameters of the compression formats supported by `dcompr`.

- The sliding window size (0...32768).
- The minimum substring length (2 or 3).
- The maximum substring length (0...256). If the maximum substring length is less than the minimum length, no sliding window compression takes place and only literals are generated.
- Whether or not a substring may match part of itself.
- Whether or not a substring may match the zero filled part of the sliding window buffer in which no data is shifted.
- Whether or not Huffman coding is used for lengths and/or literals.
- The type of address coding (No Huffman, type 1 or type 2).
- Whether static Huffman is used or dynamic Huffman coding.
- Whether or not one set of trees is used with static Huffman.
- Whether end of section commands are allowed.
- The byte alignment of these end of section commands (1, 2 or 4)

The program `dcompr` also supports a number of parameters that pertain to the compression `method` and not strictly to the format:

- Whether or not hash chains are truncated to speed up substring searches. This pertains only to the software algorithm used in `dcompr`. It has no counterpart in the VLSI circuit. This speedup was never used in the benchmarks.
- The lookahead method used by the sliding window coder (none, hardware compatible or software).
- The number of lengths and literals that is buffered before a Huffman code for the block is computed.

B.2 The GZIP-Based Format of `gdcompr`

The GZIP format is described in the documents rfc-1951 and rfc-1952 [Deu96a, Deu96b]. Both documents can be freely obtained by ftp. The file rfc-1951 describes the 'deflate' compression method, rfc-1952 describes how this method is used in the `gzip` file format.

A `gzip` file consists of a simple header followed by deflated data and it is terminated by the CRC and length of the original file.

The 'deflate' compression format is based on the same principles as the LHA format. It has the following differences with LHA:

- The sliding window size is 32768.
- The maximum string length is 258.
- There is a class-offset system for lengths (29 classes). The length-literal Huffman code has 286 members (256 literals, end-of-block, 29 length classes).
- The address Huffman code has 30 address classes.
- The bit ordering is different.
- The tree encoding is different, but based on the same principles.
- There are three types of data blocks in the compression format. These blocks (except uncompressed blocks) do not have a length field, but an encoded end-of-block mark instead. The following block types exist (see 3.2.3 of [Deu96a]):
  - No compression (BTYPE=00). The block contains uncompressed data bytes.
  - Fixed Huffman codes (BTYPE=01). A fixed prefix code is used.
  - Static Huffman coding (called dynamic Huffman codes) (BTYPE=10). The block contains an encoded description of its Huffman trees.

The program `gdcompr` generates `gzip` compatible files with a minimal file header. All optional header fields are absent. The program cannot generate uncompressed data blocks, but it can decompress them.

B.2.1 Nonstandard Extension

The program `gdcompr` supports the following extensions with respect to [Deu96a]:

- The sliding window code size can be extended to 65536. The offset class table (section 3.2.5 of [Deu96a]) contains the extra codes 30 and 31, each with 14 extra bits and covering the offsets 32769...49152 and 49153...65536 respectively.

- End-of-section codes can be supported with fixed and dynamic coding. It is assigned code 286 in the length/literal Huffman tree.
- Dynamic Huffman coding is supported. It is assigned the reserved block type (BTYPE=11). The initial length/literal tree contains 288 entries with count 1 and the initial address tree contains 32 entries with count 1. The update algorithm is the same as `dcompr`.

B.2.2 Selectable Parameters

This section summarizes the user selectable parameters of the compression format (and method) supported by `gdcompr`.

- The sliding window size (0…32768, 32769…65536 as nonstandard extension).
- The maximum substring length (0…258). If the maximum substring length is less than 3, no sliding window compression takes place and only literals are generated.
- Whether or not a substring may match part of itself.
- Whether or not a substring may match the zero filled part of the sliding window buffer in which no data is shifted (nonstandard).
- The type of Huffman coding:

  fix Compressed data consists of one fixed block (BTYPE=01).

  hold Compressed data consists of one static block (BTYPE=10). The tree descriptions have codes for all possible symbols.

  stat Compressed data consists of one or more static blocks (BTYPE=10) and optionally one small fixed block (BTYPE=01). Each static block has its own tree descriptions and the trees contain only symbols that are actually used in the block.

  dyn Compressed data consists of one dynamic block (BTYPE=11). This is a nonstandard extension.

  The decompressor can handle all combinations of block types.
- Whether end of section commands are allowed (nonstandard extension).
- The byte alignment of these end of section commands (1, 2 or 4)
- Whether or not hash chains are truncated to speed up substring searches. This pertains only to the software algorithm used in `dcompr`. It has no counterpart in the VLSI circuit. This speedup was never used in the benchmarks.

- The lookahead method used by the sliding window coder (none, hardware compatible or software).
- The number of lengths and literals that is buffered before a Huffman code for the block is computed (1 ... 32768).

Appendix C

Comparison of Compression Methods.

In order to select an appropriate compression method (or range of compression methods) for the VLSI compression circuit, we need to compare the compression ratios of many compression methods. The design requirements already specified the use of a sliding window method with Huffman coding, therefore we only need to consider methods within this category.

Within this category there are many existing compression programs and countless possible variations of the format. The sliding window coder can roughly be characterized by the following parameters:

- The size of the sliding window buffer.
- The maximum substring length.
- The minimum substring length.
- Whether or not a substring can match part of itself.
- The method of lookahead (if any).

The Huffman coding that is performed on the lengths, literals and addresses can roughly be characterized by the following parameters:

- Whether addresses, lengths and literals are Huffman coded or not.
- Whether the lengths and literals each have their own code or share one Huffman code.
- Whether static or dynamic Huffman coding is used.
- For static Huffman coding the size of the blocks that are statically coded.
- For static Huffman coding whether one set of static codes is generated based on the statistics of the first part of the file or a different set of Huffman codes is generated for each block.

- The way an address is split into a Huffman coded part (class) and a non Huffman coded part (offset).
- Whether a similar class-offset system is used for lengths.

There are nearly infinitely many possible ways to perform the exact Huffman coding of the lengths, literals and addresses and to encode the static Huffman trees. For dynamic Huffman coding there is a similarly great variety in the initial Huffman tree, the update algorithm and the recomputation algorithm. Therefore we cannot examine every single possible variation within the category of LZH based algorithms. We have to restrict the field a bit. Existing compression programs and their associated compression formats tend to differ in many respects, so by comparing the compression performance of existing programs, one cannot compare the influence of every aspect in isolation. Therefore we decided to use `one` existing compression format as a basis and make several parameters changeable.

Originally we decided to adhere to the compressed data format used in the popular program LHA [TO92]. This format was originally implemented by Haruhiko Okumura [Oku90]. Later this code was adapted and included in the current version of LHA. We wanted that the compressor supported at least one commonly used compression format and we chose LHA because that was the most advanced format that had a window size of 8k. Extensions were made to this format to include larger window sizes, dynamic Huffman codes, fixed codes for addresses or lengths and literals and an alternative address coding scheme. This format is described in great detail in section B.1. Note that the parameters can be set to use a format that is compatible with LHA.

Later we decided to move to the more widely used deflate compression format, as used by the `gzip` program.

We wrote the program `dcompr` to be able to compress and decompress all formats described in section B.1. Later we moved to the `gzip` format and adapted `dcompr` accordingly. The resulting program was called `gdcompr`. The program includes some extensions over the normal `gzip` format, such as sliding window sizes of up to 64kB and dynamic Huffman coding.

We checked the influence on the compression ratio of many parameters. Some had a comparatively large influence, some had a negligible influence.

We compared the compression ratio of existing LZH based programs with those of `dcompr` when set to similar compression parameters. This was done to assert the claim that the influence of parameters not changeable with `dcompr` (e.g. those countless tree encoding variations) have a comparatively minor effect on the compression ratio. The results obtained in this chapter are thus representative for all LZH based compressors.

Finally we compared the compression ratios obtained with LZH to those obtainable by different methods to assert the claim that they are the best practical lossless compression methods within the state of the art.

C.1 The Programs dcompr and gdcompr.

Most compression ratio comparisons in this chapter were obtained by the program `dcompr`.

The program `dcompr` is a software implementation of the original compression algorithms that are included in the VLSI chip or that were candidates for inclusion. As the format was changed to GZIP, this program was replaced with `gdcompr`. The purposes of this program are:

- To have a program that is capable of decompressing data that is compressed by the VLSI circuit in case the VLSI circuit is unavailable. This purpose is now obsolete, as the standard `gzip` program fulfills this requirement.
- To verify the compressed data generated by the VLSI circuit (or parts thereof) against what should be generated. That is, the program acts as a behavioral model of the designed system.
- To have a means to compare the compression ratios of several variations of the algorithm. Compression parameters can be changed, often beyond those that will eventually be available on the real VLSI circuit. This enables the designer to select algorithm variations for inclusion in the VLSI circuit.
- To have a means to collect rate statistics on which architectural decisions can be based. As the program acts as a behavioral model of the circuit, it can be used to compute the rate at which several operations take place.

The program was written in ANSI C to facilitate porting to various computer systems. Several versions have been created, some of which include rate statistics collection. Some versions have buffer sizes that extend beyond those that will be present in the real VLSI circuit. These versions can process a superset of the compression formats supported by the hardware.

Although `dcompr` was intended to be a behavioral model of the circuit under design, it does not mimic the behavior of the hardware in great detail. It does however produce exactly the same compressed data.[1] The program `dcompr` and the VLSI implementation use different algorithms to obtain the same results. The following operations are performed with different algorithms in hardware and software:

- The sliding window compression is performed in software using hashing of all substrings of length 2 or 3. In hardware an array of parallel cells is used.
- The Huffman coding itself is performed bit-serially in software, using a tree data structure. In hardware a table-lookup method is used whereby the Huffman code

[1] Both the LZ77 algorithm and static Huffman coding can produce different compressed output for the same input data, while all different outputs are still compatible with the compression format. These outputs will all produce the same original data when decompressed.

for each symbol is looked up as a variable length bit field. These bit fields are later converted to a byte stream. Decoding is performed in hardware using a table lookup method that decodes the symbol in a single operation most of the time.

- The computation of the Huffman tree is performed in software using the standard heap-based algorithm. The VLSI circuit uses a faster algorithm by Lu and Chen.
- Dynamic Huffman coding is performed bit-serially in software with the algorithm described by Gallagher. In hardware an array with parallel cells would perform the task, though we decided not to implement this yet.

C.2 Test Data

The "Text Compression Corpus" is a set of files available from the University of Calgary in Canada [WB90]. This set of files has become a standard benchmark set for text compression algorithms. It was initially used in [BCW89].

The Text Compression Corpus contains several types of computer files, totalling 3 megabytes. Most, but not all of the files are text files. The files are:

- English text, two books (`book1` and `book2`) and two scientific papers (`paper1` and `paper2`).
- English text in non-typical formats, a batch of USENET news (`news`) and a bibliography (`bib`).
- Computer program source code in various languages (`progc`, `progl` and `progp`).
- Computer program object code (`obj1` and `obj2`)
- Binary geophysical data (`geo`). This is badly compressible.
- A black-and-white picture (`pic`). This is highly compressible data.
- Data as it is transported to a computer terminal (`trans`).

All comparisons of compression ratio are performed on this Text Compression Corpus. The files `paper3` through `paper6` are included in the corpus, but they are mostly left out of compression benchmarks. This leaves 14 files in the set for comparison.

The compression ratio is specified in bits per byte, this means bits of compressed data per byte of uncompressed data. If, for example, the uncompressed size is 8000 bytes and the compressed size is 2000 bytes (16000 bits), the ratio is 2 bits per byte. A lower figure for the compression ratio means better compression. The average compression ratio is the nonweighted average compression ratio of all 14 files. This way of specifying the compression ratio is what elsewhere in the scientific literature is used.

C.3 Results from `dcompr`

All results in this section are obtained using the program `dcompr`. This program produces data in the format described in section B.1. Unless explicitly stated otherwise the compression parameters are set as follows:

- The sliding window buffer size (N) is set to 8k (8192 bytes).
- The minimum substring length (M_{min}) is set to 3.
- The maximum substring length (M_{max}) is set to 256.
- A substring may match part of itself.
- A substring may not match the initial zero contents of the sliding window buffer.
- Exact matching is enabled. Hash chains are not cut off to speed up `dcompr`.
- Hardware-compatible lookahead is enabled.
- Lengths and literals are encoded using static Huffman coding.
- The code buffer size is set to 8k (8192 lengths or literals).
- For each block a new static Huffman code is generated.
- Addresses are coded using the type-2 address code.

These parameters yield a compressed data format compatible with the LHA program. The actual compressed data differs from that generated by the LHA program because:

- LHA generates a new static Huffman code for every 8192 input bytes whereas `dcompr` does so for every 8192 lengths or literals.
- LHA uses a slightly different lookahead algorithm that cannot be implemented in the proposed hardware compressor.

First we checked the influence of the size of the sliding window buffer on the compression ratio. Tables C.1, C.2 and C.3 show the results for no lookahead, hardware-compatible lookahead (as used by the VLSI coder) and software lookahead (as used by LHA) respectively.

The average compression ratio is shown in figure C.1. This figure shows that the total size of the compressed data decreases with increasing window size. When the window size is doubled, the total compressed size decreases by nearly 5%. The use of lookahead has at least as much influence as doubling the window size. The hardware compatible lookahead is only slightly inferior to the ideal software lookahead.

file	size	0.5k	1k	2k	4k	8k	16k
bib	111261	4.049	3.831	3.599	3.332	3.098	2.855
book1	768771	4.255	4.131	3.992	3.838	3.684	3.531
book2	610856	3.872	3.672	3.489	3.311	3.133	2.966
geo	102400	5.612	5.588	5.533	5.539	5.483	5.500
news	377109	4.303	4.071	3.724	3.545	3.438	3.335
obj1	21504	4.111	3.980	3.940	3.947	3.948	3.961
obj2	246814	3.556	3.276	3.072	2.927	2.875	2.840
paper1	53161	3.903	3.675	3.463	3.259	3.099	2.977
paper2	82199	3.961	3.772	3.578	3.408	3.267	3.122
pic	513216	0.841	0.847	0.853	0.862	0.870	0.880
progc	39611	3.634	3.416	3.265	3.095	2.935	2.848
progl	71646	2.856	2.585	2.277	2.057	1.967	1.931
progp	49379	2.950	2.613	2.350	2.028	1.928	1.907
trans	93695	3.470	3.195	2.772	2.359	2.010	1.855
Average		3.670	3.475	3.279	3.108	2.981	2.894

Table C.1: Compression without Lookahead with Varying Window Size.

file	size	0.5k	1k	2k	4k	8k	16k
bib	111261	3.974	3.731	3.474	3.207	2.970	2.742
book1	768771	4.153	4.017	3.850	3.704	3.560	3.416
book2	610856	3.783	3.575	3.374	3.190	3.017	2.863
geo	102400	5.534	5.471	5.434	5.446	5.356	5.382
news	377109	4.244	3.993	3.633	3.437	3.317	3.211
obj1	21504	4.074	3.931	3.901	3.878	3.857	3.860
obj2	246814	3.514	3.217	3.000	2.835	2.762	2.704
paper1	53161	3.827	3.584	3.358	3.139	2.980	2.867
paper2	82199	3.861	3.664	3.468	3.291	3.152	3.025
pic	513216	0.814	0.815	0.818	0.825	0.831	0.841
progc	39611	3.568	3.339	3.184	3.007	2.836	2.753
progl	71646	2.787	2.522	2.210	1.991	1.902	1.868
progp	49379	2.896	2.554	2.286	1.972	1.874	1.854
trans	93695	3.420	3.136	2.714	2.304	1.960	1.811
Average		3.603	3.396	3.193	3.016	2.884	2.800

Table C.2: Compression with Hardware Lookahead with Varying Window Size.

file	size	0.5k	1k	2k	4k	8k	16k
bib	111261	3.948	3.695	3.430	3.159	2.927	2.703
book1	768771	4.131	3.985	3.820	3.669	3.524	3.385
book2	610856	3.764	3.546	3.343	3.158	2.988	2.838
geo	102400	5.532	5.469	5.432	5.443	5.355	5.380
news	377109	4.230	3.975	3.610	3.409	3.287	3.180
obj1	21504	4.069	3.922	3.892	3.872	3.844	3.846
obj2	246814	3.509	3.211	2.993	2.825	2.751	2.690
paper1	53161	3.811	3.558	3.329	3.116	2.957	2.846
paper2	82199	3.840	3.632	3.430	3.258	3.120	2.999
pic	513216	0.806	0.804	0.805	0.808	0.812	0.821
progc	39611	3.551	3.320	3.163	2.983	2.812	2.728
progl	71646	2.770	2.503	2.195	1.977	1.886	1.852
progp	49379	2.885	2.541	2.277	1.962	1.862	1.842
trans	93695	3.406	3.117	2.693	2.283	1.940	1.792
Average		3.590	3.377	3.172	2.994	2.862	2.779

Table C.3: Compression with Software Lookahead with Varying Window Size.

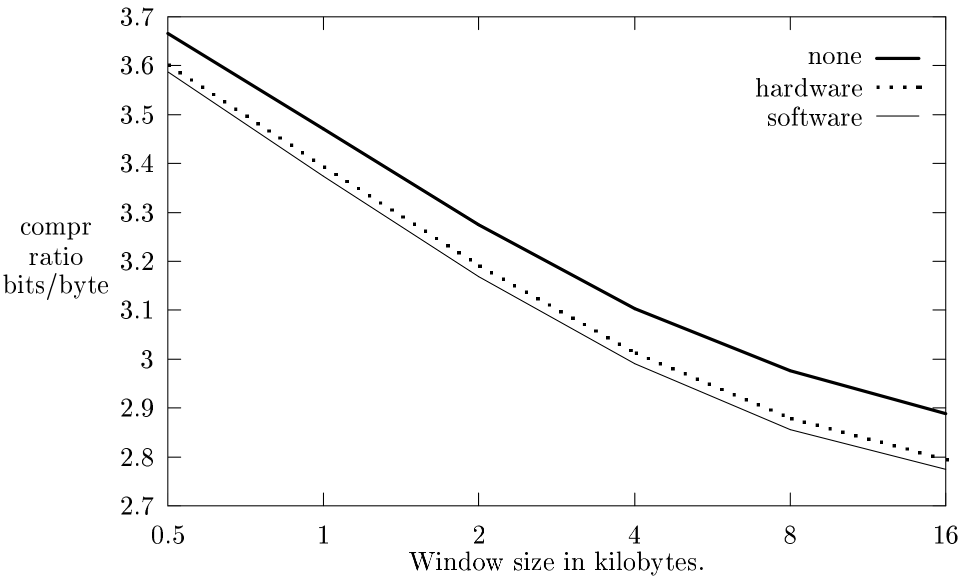

Figure C.1: Compression Ratio versus Window Size with Various Types of Lookahead.

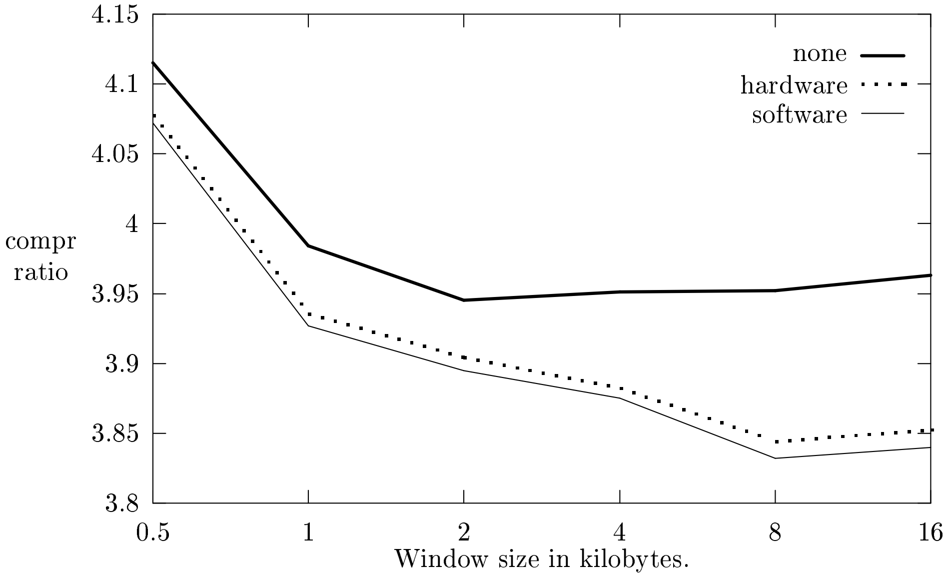

Figure C.2: Compression Ratio versus Window Size for File `obj1`.

Increasing the window size is not beneficial for all files. For some binary files the compressed size increases when the window size is increased. With lookahead the deterioration starts at a larger window size than without. Figure C.2 shows the effect for the file `obj1`. This deterioration effect is well-known for many Lempel-Ziv based algorithms. It is caused by the fact that, if the window size is increased, more bits are needed to represent addresses. With binary files substrings tend to be short and therefore the extra substrings found in a larger window do not provide enough improvement to make up for this effect.

Next we test the influence of the maximum string length on the compression ratio. The results are in table C.4. The average compression ratio is shown in figure C.3. Increasing the maximum string length beyond 64 has a negligible influence on the compression ratio, except for the file `pic`, which contains very long strings of mostly zero bytes. Increasing the maximum string length never (almost never, see the file `trans`)[2] has a negative influence on the compression ratio.

Next we check the influence of the code buffer size on the compression ratio. The results are shown in table C.5. The average compression ratio is shown in figure C.4. The code buffer size has a very small influence on the compression ratio. Compression nearly always improves with increasing code buffer size[3], because fewer code descriptions have to be included in the compressed data.

[2]This degradation with increasing maximum string length is due to the fact that the LZ77 algorithm, even with lookahead, does not pick the optimum set of substrings. It so happened that after a substring longer than 128 was parsed, a worse set of substrings was parsed.

[3]With the file `obj2` the compression ratio deteriorates when the buffer size is increased from 16k to 32k. This cannot be caused by the sliding window coder. It is probably caused by the fact that the statistics of the data vary over the file and it is better compressed using more smaller static code blocks.

file	size	8	16	32	64	128	256
bib	111261	3.285	3.040	2.984	2.971	2.970	2.970
book1	768771	3.602	3.562	3.561	3.560	3.560	3.560
book2	610856	3.224	3.054	3.022	3.017	3.017	3.017
geo	102400	5.425	5.383	5.362	5.356	5.356	5.356
news	377109	3.621	3.408	3.342	3.322	3.318	3.317
obj1	21504	4.153	3.964	3.886	3.871	3.860	3.857
obj2	246814	3.230	2.925	2.812	2.774	2.764	2.762
paper1	53161	3.222	3.033	2.989	2.981	2.980	2.980
paper2	82199	3.296	3.171	3.155	3.153	3.152	3.152
pic	513216	1.124	1.002	0.933	0.880	0.845	0.831
progc	39611	3.143	2.924	2.850	2.838	2.837	2.836
progl	71646	2.484	2.120	1.977	1.925	1.905	1.902
progp	49379	2.486	2.125	1.969	1.908	1.883	1.874
trans	93695	2.648	2.234	2.060	1.983	1.953	1.960
Average		3.210	2.996	2.921	2.896	2.886	2.884

Table C.4: Compression with Hardware Lookahead with Varying Maximum String Length.

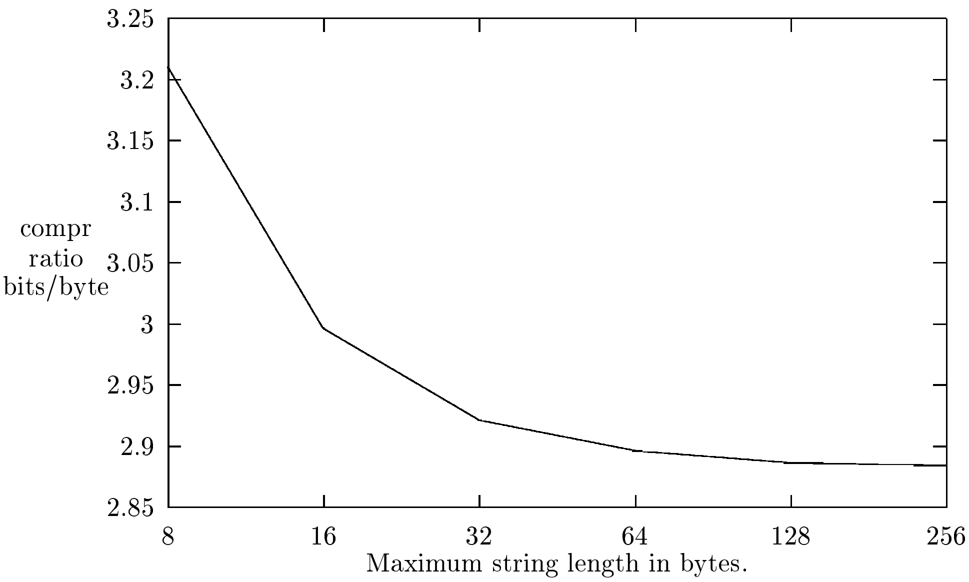

Figure C.3: Compression Ratio versus Maximum String Length with Hardware Lookahead.

file	size	0.5k	1k	2k	4k	8k	16k	32k
bib	111261	3.103	3.037	2.999	2.981	2.970	2.961	2.956
book1	768771	3.705	3.636	3.596	3.573	3.560	3.553	3.550
book2	610856	3.133	3.076	3.044	3.026	3.017	3.011	3.008
geo	102400	5.681	5.527	5.439	5.384	5.356	5.342	5.336
news	377109	3.468	3.393	3.352	3.329	3.317	3.310	3.306
obj1	21504	4.051	3.932	3.889	3.853	3.857	3.846	3.846
obj2	246814	2.906	2.835	2.791	2.773	2.762	2.759	2.766
paper1	53161	3.080	3.018	2.991	2.984	2.980	2.975	2.975
paper2	82199	3.265	3.207	3.175	3.158	3.152	3.147	3.145
pic	513216	0.880	0.856	0.843	0.835	0.831	0.831	0.830
progc	39611	2.936	2.881	2.850	2.836	2.836	2.827	2.827
progl	71646	1.973	1.938	1.920	1.909	1.902	1.897	1.897
progp	49379	1.941	1.907	1.891	1.881	1.874	1.874	1.874
trans	93695	2.025	1.992	1.972	1.964	1.960	1.957	1.957
Average		3.011	2.945	2.911	2.892	2.884	2.878	2.877

Table C.5: Compression with Hardware Lookahead with Varying Code Buffer Size.

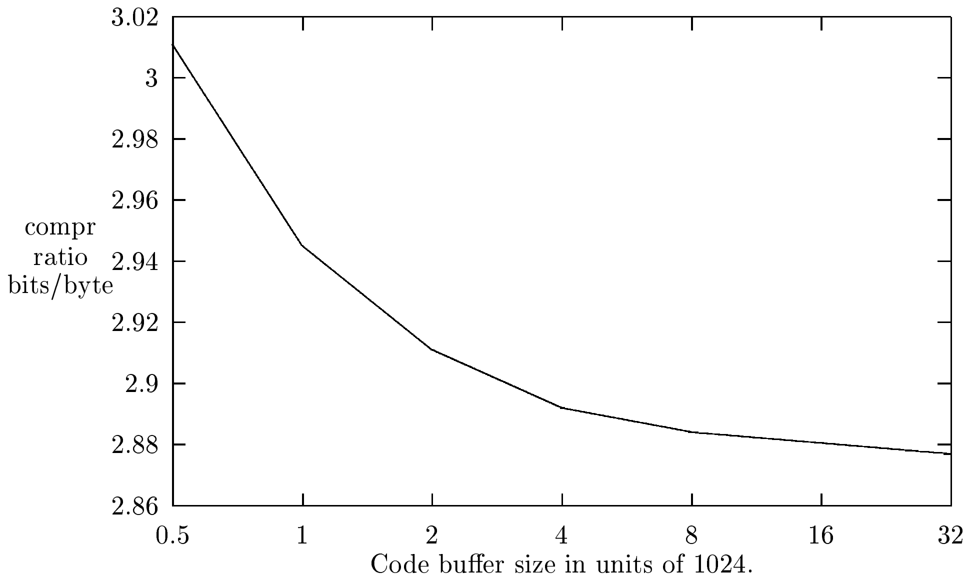

Figure C.4: Compression Ratio versus Code Buffer Size with Hardware Lookahead.

file	size	fix	hold	dyn	stat	min2	noself	addr1
bib	111261	3.390	3.103	3.102	3.098	3.131	3.098	3.102
book1	768771	4.127	3.704	3.679	3.684	3.707	3.685	3.690
book2	610856	3.429	3.159	3.131	3.133	3.154	3.136	3.138
geo	102400	5.853	5.503	5.478	5.483	5.568	5.483	5.498
news	377109	3.759	3.452	3.432	3.438	3.481	3.449	3.444
obj1	21504	4.189	3.964	3.978	3.948	3.941	3.970	3.969
obj2	246814	3.002	3.003	2.889	2.875	2.873	2.878	2.883
paper1	53161	3.367	3.118	3.127	3.099	3.129	3.101	3.104
paper2	82199	3.602	3.283	3.286	3.267	3.298	3.267	3.272
pic	513216	1.002	0.874	0.869	0.870	0.916	0.876	0.864
progc	39611	3.176	2.949	2.964	2.935	2.963	2.940	2.942
progl	71646	2.131	1.973	1.979	1.967	1.993	1.971	1.972
progp	49379	2.093	1.938	1.949	1.928	1.946	1.929	1.933
trans	93695	2.190	2.015	2.017	2.010	2.036	2.021	2.013
Average		3.236	3.003	2.991	2.981	3.010	2.986	2.987

Table C.6: Compression without Lookahead with Varying Coding Techniques.

Table C.6 shows the compression results for various coding variations. without lookahead. Table C.7 shows the results for some of the same coding variations where hardware lookup is used. One can see that the standard parameters yield the best compression ratio among all these coding variations. The results of dynamic Huffman coding are directly comparable to those of static coding with a buffer size of 8k. The following variations are compared:

- **fix** Fixed prefix codes are selected for both the address and the length/literal code. The minimum string length is 2 when no lookahead is used, 3 with lookahead.
- **hold** One set of static Huffman codes is used for the entire file. The codes are based on the frequency statistics of the first 8192 lengths or literals.
- **dyn** Dynamic Huffman coding is used for the address and the length/literal code.
- **stat** All parameters are set to the standard values; static Huffman coding is therefore used.
- **min2** The minimum substring length is set to 2 instead of 3.
- **noself** A substring cannot match part of itself.
- **addr1** The type-1 address code is used instead of the type-2 code.

file	size	fix	hold	dyn	stat	addr1
bib	111261	3.342	2.978	2.976	2.970	2.974
book1	768771	4.105	3.573	3.556	3.560	3.565
book2	610856	3.400	3.034	3.013	3.017	3.021
geo	102400	5.807	5.382	5.350	5.356	5.372
news	377109	3.735	3.323	3.310	3.317	3.322
obj1	21504	4.134	3.871	3.888	3.857	3.869
obj2	246814	2.942	2.852	2.770	2.762	2.771
paper1	53161	3.330	2.994	3.006	2.980	2.985
paper2	82199	3.576	3.169	3.170	3.152	3.159
pic	513216	0.979	0.833	0.830	0.831	0.823
progc	39611	3.147	2.849	2.865	2.836	2.843
progl	71646	2.107	1.906	1.913	1.902	1.907
progp	49379	2.074	1.884	1.896	1.874	1.879
trans	93695	2.187	1.974	1.970	1.960	1.964
Average		3.205	2.902	2.894	2.884	2.890

Table C.7: Compression with Hardware Lookahead with Varying Coding Techniques.

C.4 Results with the Program `gdcompr`.

During the course of the design process the compression format was changed to `gzip` instead of LHA and the `dcompr` program was changed accordingly. Figure C.8 shows the compression ratio as a function of the window size using hardware lookahead. Further the usual `gzip` compatible parameters are used: minimum string length is 3, maximum string length is 258, self matching is allowed and static Huffman coding is used. It can be seen that the compression results are nearly identical to LHA.

Further we tested the compression ratio for the following Huffman methods:

- **fix** Fixed coding is used. RFC-1951 provides fixed settings for the Huffman codes, not dependent on the statistics of the data, and these are used here.
- **hold** Statistics are based on the first part of the data and Huffman codes are computed from these. The same Huffman codes are used throughout the file.
- **static** The data is split into blocks. New Huffman codes are computed for each block.
- **dynamic** Dynamic Huffman coding is used. This does not conform to RFC-1951.

In all cases the window size was 8k, the maximum string length was 258 and hardware compatible lookahead was used. Where applicable the code buffer contained 8192 entries.

file	size	0.5k	1k	2k	4k	8k	16k	32k
bib	111261	3.968	3.725	3.470	3.202	2.966	2.738	2.553
book1	768771	4.152	4.013	3.848	3.701	3.559	3.414	3.283
book2	610856	3.779	3.571	3.370	3.187	3.013	2.859	2.726
geo	102400	5.527	5.465	5.426	5.431	5.348	5.374	5.382
news	377109	4.240	3.988	3.628	3.431	3.311	3.205	3.102
obj1	21504	4.077	3.935	3.904	3.882	3.844	3.852	3.853
obj2	246814	3.505	3.225	2.991	2.827	2.754	2.698	2.647
paper1	53161	3.824	3.581	3.355	3.137	2.977	2.863	2.818
paper2	82199	3.863	3.662	3.467	3.291	3.151	3.023	2.909
pic	513216	0.803	0.804	0.808	0.815	0.821	0.832	0.841
progc	39611	3.567	3.338	3.182	3.005	2.835	2.752	2.724
progl	71646	2.784	2.515	2.205	1.984	1.895	1.862	1.830
progp	49379	2.895	2.549	2.280	1.971	1.868	1.846	1.828
trans	93695	3.425	3.132	2.710	2.297	1.952	1.804	1.646
Average		3.601	3.393	3.189	3.012	2.878	2.794	2.724

Table C.8: Compression with Hardware Compatible Lookahead for Various Window Sizes (`gzip`).

file	size	fix	hold	static	dynamic
bib	111261	3.585	2.968	2.966	2.971
book1	768771	4.535	3.561	3.559	3.551
book2	610856	3.689	3.035	3.013	3.011
geo	102400	6.579	5.351	5.348	5.329
news	377109	3.985	3.320	3.311	3.305
obj1	21504	4.174	3.844	3.844	3.862
obj2	246814	3.051	2.842	2.754	2.762
paper1	53161	3.588	2.987	2.977	2.999
paper2	82199	3.910	3.163	3.151	3.167
pic	513216	1.011	0.825	0.821	0.823
progc	39611	3.330	2.838	2.835	2.857
progl	71646	2.223	1.899	1.895	1.910
progp	49379	2.166	1.877	1.868	1.892
trans	93695	2.264	1.968	1.952	1.967
Average		3.435	2.891	2.878	2.886

Table C.9: Compression with 8k Sliding Window and Various Types of Huffman. Lookahead is Hardware Compatible.

C.5 Comparisons of Dynamic Huffman Codes.

Prior to the development of the program `dcompr` several experiments were performed with sliding window coding combined with dynamic Huffman coding. The best method that emerged from these experiments was the basis for the dynamic Huffman coding used in `dcompr`. Further it is believed that some of these results also hold for static Huffman coding. Tests with `dcompr` support this belief.

Several varieties of dynamic Huffman coding of sliding window encoded data were compared. Table C.10 shows the compressed size of the files from the Text Compression Corpus using several of these methods. The sliding window size was 8192 and the maximum string length was 256. No lookahead was used.

The following programs were compared. Note that their names are unique to this paper; they do not refer to existing algorithms published in the literature, which may accidentally have the same name.

- **lza** One Huffman code is used for both lengths and literals. The code encodes 512 symbols, 256 literals, 255 lengths and the end-of-file symbol. The code is initialized with a count of 1 for each symbol. If the total count reaches 32767, the code is reinitialized to that same state. Addresses are encoded by a fixed variable-length code, the same as described in appendix B.1 (compression format).

- **lzb** One Huffman code is used for literals only. It encodes 256 different literals. The same initialization and reinitialization policy is used. Both addresses and lengths are encoded by a fixed variable-length code. The code for lengths is very similar to that described in appendix B.1 (compression format). One bit precedes every length and literal to distinguish between lengths and literals.

- **lzc** Two Huffman codes are used. The code for lengths and literals is equal to that of lza. Another code encodes the 7 most significant bits of the address in 128 symbols. The 6 least significant address bits are transmitted unchanged (this is the type-1 address code). The address code is also initialized and reinitialized to a code with a count of 1 for each symbol.

- **lzd** Three Huffman codes are used, one for literals that encodes 256 symbols, one for lengths that encodes 256 symbols and one for addresses that is equal to that of lzc. All codes are initialized and reinitialized to a code with a count of 1 for each symbol. A single bit precedes each length and literal to distinguish between lengths and literals.

- **lzx** This scheme is similar to that of lzc, but the codes are initialized such that small lengths and small addresses have higher initial counts (and hence shorter initial code words). After the total count reaches 32767, all symbol counts are divided by 2 (rounded up) and the Huffman tree is recomputed. This is very similar to the final variation described in appendix B.1 (compression format).

file	size	lza	lzb	lza(3)	lzb(3)	lzc(3)	lzd(3)	lzx(3)	lzy(3)
bib	111261	3.224	3.337	3.188	3.253	3.105	3.105	3.103	3.103
book1	768771	3.832	4.085	3.823	3.997	3.699	3.717	3.678	3.703
book2	610856	3.236	3.385	3.221	3.323	3.146	3.148	3.130	3.139
geo	102400	5.644	5.860	5.536	5.826	5.492	5.570	5.474	5.566
news	377109	3.563	3.681	3.515	3.585	3.450	3.442	3.430	3.432
obj1	21504	4.075	4.057	4.039	4.125	3.993	4.059	3.980	4.038
obj2	246814	2.943	2.960	2.934	2.993	2.893	2.901	2.889	2.899
paper1	53161	3.216	3.316	3.187	3.265	3.133	3.138	3.127	3.132
paper2	82199	3.388	3.556	3.367	3.486	3.290	3.302	3.287	3.298
pic	513216	0.938	0.958	0.886	0.908	0.863	0.864	0.860	0.864
progc	39611	3.060	3.116	3.016	3.065	2.972	2.971	2.964	2.963
progl	71646	2.045	2.096	2.012	2.049	1.985	1.987	1.981	1.982
progp	49379	2.008	2.040	1.985	2.016	1.960	1.964	1.952	1.955
trans	93695	2.087	2.125	2.051	2.082	2.021	2.021	2.018	2.017
Average		3.090	3.184	3.054	3.141	3.000	3.013	2.991	3.007

Table C.10: Comparison of Various Dynamic Huffman Schemes.

- **lzy** This scheme is similar to that of lzd, but with the initialization and reinitialization policy of lzx.

It turned out to be favorable to discard substrings of length 2 and convert them back to the two literals they were composed of. In table C.10 these results are shown with a 3 in parentheses after the name to indicate a minimum substring length of 3. The programs were not adapted to eliminate the encoding of length 2. The programs lzx and lzy (that were only tested with minimum substring length 3) had no initial code with a high count for length 2.

From the table we can see the following:

- If substrings of length 2 are discarded and replaced by two literals, the compression ratio improves.
- Schemes with a combined Huffman code for lengths and literals are better than schemes with separate codes for lengths and literals.
- Using a Huffman code instead of a fixed code for addresses improves the compression ratio.
- Recomputing the Huffman code (instead of resetting it to all counts 1) after a count overflow improves the compression ratio.

C.6 Other LZH Based Methods.

We compared the compression ratio of existing LZH-based compression methods to the program `dcompr` with the parameters set as closely as possible to those used by the existing program. It is expected, that if we set the characteristic parameters close to that of a different program, compression ratios will be very close as well.

The following programs were tested. Of all except PKZIP1.1 the source code was available and it was examined to find out some characteristics of the compression method.

- **lharc** It uses a sliding window of 4k, a maximum substring length of 64, dynamic Huffman coding for lengths and literals and a fixed encoding for addresses. The original program uses no lookahead. It is compared to `dcompr` with the parameters set as mentioned.

- **lha ao** The same compression format as lharc, but created by the more recent program lha. As opposed to the original lharc, this method does use lookahead. It is compared to `dcompr` with the same parameters as above but with lookahead enabled.

- **lha** It uses a sliding window of 8k, a maximum substring length of 256, static Huffman coding for lengths, literals and addresses and the type-2 address code. This is the program whose format was used in `dcompr`. It is compared to `dcompr` with the standard settings.

- **pkzip 1.1** This compression format uses a sliding window of 8k, a maximum substring length of 321, static Shannon-Fano codes(similar to Huffman), one set of codes for the whole file and address classes of fixed size. How the compression method differs from Info-Zip 1.1 is unknown.

- **Info Zip 1.1** This program uses the same compression format as PKZIP 1.1, but achieves noticeably better compression. It was compared to `dcompr` with a sliding window size of 8k, a maximum string length of 256, static Huffman coding, the type-1 code for addresses and a single set of codes for the whole file.

- **gzip -9** This uses a sliding window of 32k, a maximum substring length of 258, static Huffman coding, variable size length and address classes. It was compared to `dcompr` with a window size of 32k, a maximum substring length of 256, static coding, a code buffer size of 32k and type-2 address coding.

`lharc` and `lha ao` perform slightly better than `dcompr` with similar parameters. The fixed address code of `dcompr` is optimized for a sliding window of 8k, not for 4k. This could have caused the difference. The difference that lookahead makes, is clearly visible in the comparison. Why `PKZIP 1.1` performs worse than `InfoZip 1.1`, is unknown. It could be an inferior implementation of hashing in the sliding window coder (this is a guess). The difference is too small to be caused by the lack of lookahead. The program `gzip -9` has a slightly better performance than `dcompr` and that is probably caused by the address coding in `gzip` that uses more address classes.

file	size	lharc	dcompr	lha ao	dcompr	lha	dcompr
bib	111261	3.344	3.397	3.176	3.232	2.929	2.927
book1	768771	3.845	3.926	3.680	3.763	3.528	3.524
book2	610856	3.307	3.371	3.163	3.222	2.992	2.988
geo	102400	5.543	5.555	5.415	5.468	5.357	5.355
news	377109	3.523	3.597	3.410	3.469	3.290	3.287
obj1	21504	3.999	4.048	3.915	3.947	3.836	3.844
obj2	246814	2.945	3.001	2.860	2.897	2.754	2.751
paper1	53161	3.273	3.325	3.135	3.186	2.961	2.957
paper2	82199	3.433	3.489	3.278	3.340	3.124	3.120
pic	513216	0.957	0.958	0.913	0.912	0.814	0.812
progc	39611	3.110	3.159	3.004	3.047	2.816	2.812
progl	71646	2.095	2.125	2.013	2.044	1.889	1.886
progp	49379	2.072	2.108	2.011	2.041	1.864	1.862
trans	93695	2.399	2.430	2.310	2.339	1.928	1.940
Average		3.132	3.178	3.020	3.065	2.863	2.862

Table C.11: Comparison of `dcompr` with Existing LZH Methods (lha,lharc).

file	size	pkzip 1.1	Infozip 1.1	dcompr	gzip -9	dcompr
bib	111261	2.973	2.927	2.938	2.509	2.525
book1	768771	3.648	3.537	3.540	3.250	3.252
book2	610856	3.046	3.005	3.014	2.700	2.703
geo	102400	5.951	5.456	5.395	5.345	5.383
news	377109	3.338	3.285	3.324	3.063	3.070
obj1	21504	3.923	3.914	3.958	3.839	3.844
obj2	246814	2.921	2.809	2.857	2.628	2.649
paper1	53161	3.016	2.974	2.983	2.790	2.794
paper2	82199	3.199	3.143	3.142	2.887	2.892
pic	513216	0.995	0.886	0.811	0.817	0.826
progc	39611	2.861	2.818	2.844	2.678	2.703
progl	71646	1.927	1.893	1.906	1.805	1.816
progp	49379	1.924	1.872	1.879	1.812	1.824
trans	93695	1.975	1.941	1.971	1.611	1.641
Average		2.978	2.890	2.897	2.695	2.709

Table C.12: Comparison of `dcompr` with Existing LZH Methods (pkzip,Info-ZIP,gzip).

file	size	1	2	3	4	5
bib	111261	3.154	2.986	2.859	2.689	2.580
book1	768771	3.798	3.647	3.501	3.423	3.312
book2	610856	3.259	3.114	2.990	2.849	2.742
geo	102400	5.454	5.420	5.428	5.383	5.368
news	377109	3.483	3.375	3.294	3.165	3.084
obj1	21504	3.981	3.949	3.936	3.866	3.839
obj2	246814	3.044	2.969	2.900	2.762	2.686
paper1	53161	3.251	3.120	3.025	2.893	2.813
paper2	82199	3.413	3.280	3.165	3.034	2.932
pic	513216	1.022	0.996	0.972	0.914	0.898
progc	39611	3.120	3.000	2.929	2.780	2.695
progl	71646	2.237	2.120	2.044	1.920	1.845
progp	49379	2.167	2.087	2.029	1.906	1.842
trans	93695	2.046	1.954	1.893	1.734	1.653
Average		3.102	3.001	2.926	2.808	2.735
Time:		5.2	5.7	7.3	7.9	10.4

Table C.13: Compression of gzip-1.2.4 Levels 1–5.

All in all, the differences between existing programs and `dcompr` set to similar compression parameters, are very small, much smaller than the differences between the different programs.

C.7 `gdcompr` Compared to `gzip`.

Compression ratios were tested both with `gdcompr` and with the original `gzip`, version 1.2.4. The program `gzip` always uses a sliding window of size 32k, but it has a `compression level' parameter, adjustable from 1 to 9. This parameter determines how thoroughly the 32k sliding window is searched for the best match. It has a tremendous influence on compression time. Compression time given is the user-CPU-time in seconds needed to compress all the files, measured on a Pentium 90. A modified version of `minigzip`, a version of `gzip` included with the `zlib` library, was used to compute the 64k deflate ratio. This is not possible with standard `gzip`.

In all tests, except with the 64k deflate, the compressed file was a valid `gzip` file with a minimal file header. The decompressibility by `gzip` was verified in all those cases.

The compression device would compress the whole set in about 0.25 seconds (at 12.5 Mbytes/sec). GZIP with maximum compression gives an average compression ratio of 2.695 bits/byte and needs 22.4 seconds. The proposed device would achieve a compression ratio of 2.878 with a sliding window size of 8kB and 3.012 with a sliding window size of 4kB. To achieve comparable compression ratios GZIP would take 7.9 or 5.7 seconds respectively, which is still more than 20 times as much as the hardware.

file	size	6	7	8	9	64k
bib	111261	2.521	2.514	2.509	2.509	2.413
book1	768771	3.261	3.253	3.250	3.250	3.134
book2	610856	2.707	2.702	2.700	2.700	2.591
geo	102400	5.351	5.347	5.345	5.345	5.331
news	377109	3.073	3.069	3.063	3.063	2.928
obj1	21504	3.839	3.838	3.838	3.837	3.837
obj2	246814	2.646	2.636	2.628	2.628	2.597
paper1	53161	2.795	2.790	2.789	2.789	2.773
paper2	82199	2.895	2.890	2.887	2.887	2.850
pic	513216	0.880	0.865	0.828	0.816	0.824
progc	39611	2.680	2.677	2.677	2.677	2.677
progl	71646	1.816	1.808	1.804	1.804	1.801
progp	49379	1.821	1.816	1.812	1.811	1.798
trans	93695	1.620	1.614	1.610	1.610	1.577
Average		2.707	2.701	2.696	2.695	2.652
Time:		14.4	16.9	20.5	22.7	

Table C.14: Compression of gzip-1.2.4 Levels 6–9, plus 64k Deflate.

C.8 LZH Compared to Other Methods.

LZH based compression methods are a subclass of all lossless data compression methods. Of all methods that are practically implementable at high speeds, LZH based methods perform the best compression.

The following compression methods were compared:

- **ahuf** Dynamic Huffman coding of the bytes in the files.
- **lz77** Sliding window coding with a window size of 8k and a maximum substring length of 256. Lengths and literals are coded in 9 bits, addresses in 13.
- **compress** Modified LZW algorithm with code size up to 16 bits and dictionary clearing after compression rate decreases.
- **lzfix** Sliding window coding with fixed prefix codes for lengths, literals and addresses.
- **lharc** and **gzip** Existing LZH variations with the worst and the best compression respectively.
- **ha a2** Arithmetic coding and context modeling.

LZH clearly wins over sliding window coding without Huffman coding and Huffman coding without sliding window. It also wins over LZW based methods.

file	size	ahuf	lz77	compress	lzfix	lharc	gzip -9	ha a2
bib	111261	5.290	4.004	3.346	3.390	3.344	2.509	1.936
book1	768771	4.613	5.023	3.455	4.127	3.845	3.250	2.453
book2	610856	4.841	4.140	3.284	3.429	3.307	2.700	2.142
geo	102400	5.701	7.432	6.076	5.853	5.543	5.345	4.637
news	377109	5.256	4.535	3.864	3.759	3.523	3.063	2.616
obj1	21504	6.026	4.870	5.226	4.189	3.999	3.839	3.645
obj2	246814	6.138	3.585	4.170	3.002	2.945	2.628	2.605
paper1	53161	5.054	4.070	3.774	3.367	3.273	2.790	2.359
paper2	82199	4.680	4.369	3.519	3.602	3.433	2.887	2.332
pic	513216	1.683	1.133	0.970	1.002	0.957	0.817	0.805
progc	39611	5.298	3.844	3.866	3.176	3.110	2.678	2.382
progl	71646	4.826	2.557	3.031	2.131	2.095	1.805	1.708
progp	49379	4.953	2.509	3.112	2.093	2.072	1.812	1.701
trans	93695	5.559	2.563	3.265	2.190	2.399	1.611	1.531
Average		4.994	3.902	3.640	3.236	3.132	2.695	2.347

Table C.15: Comparison of LZH with Other Compression Methods.

Arithmetic coding with context modeling is a comparatively new method, which is rarely used in practice due to its high memory usage and low compression speed. Of all implementations in this class, `ha` has moderate memory requirements, comparatively high speed and one of the best compression ratios. Even this method would be infeasible to implement in dedicated, high-speed hardware. Compression methods in this class offer the best compression ratio for lossless data compression that is possible within the state of the art.

C.9 Conclusions

- Within the LZH methods the sliding window size N, the use of lookahead and the use of Huffman coding, for lengths and literals as well as addresses, are the main determinants of compression ratio. Methods with similar parameters perform similarly.
- Static and dynamic Huffman coding have comparable performance.
- LZH coding offers a better compression ratio than other methods that could be implemented at a very high speed in hardware. It outperforms non-Huffman coded LZ77, Huffman coding and LZW in particular.
- Arithmetic coding with context modeling offers a superior compression ratio to LZH, but this class of algorithms is not yet suitable for implementation at the desired speed.

Appendix D

Simulations

Apart from the program `gdcompr` and its predecessor `dcompr` we do not have an executable (simulatable) model of the entire chip. Only parts are simulated.

The following types of simulations were performed:

- To the programs `dcompr` and `gdcompr` some counters were added to compute the rate at which certain types of operations had to be performed.
- The parallel sliding window coding algorithm was simulated in C. The behaviour of single cells was simulated using the Compass Tools and Spice.
- The static Huffman computation algorithm was implemented in C and later it was converted to assembler so it could run on a simulated model of the internal processor. Both an instruction set simulator and the IDaSS model were used.
- The parallel dynamic Huffman coding algorithm (though not part of the device) was simulated in C.
- Some other parts (the byte converter) were simulated in IDaSS.

D.1 Rate Statistics

In order to give an accurate estimation of the typical work load for the various modules, several files have been compressed and some data have been collected. The files are the files of the Calgary Corpus. The results are shown in table D.1. The sliding window coder did not use lookahead[1].

The figures in the table have the following meaning:

- **size**. The original size in bytes.

[1] These figures were computed before lookahead was added to the circuit. Later we will examine the influence of lookahead.

file	size	lit.	substr.	lit. + substr.	compr size	compr ratio	avg. substr	byte conv	byte conv ratio.
bib	111261	13564	14701	28265	43082	2.58	3.94	57667	1.93
book1	768771	71989	146065	218054	354062	2.17	3.53	510184	1.51
book2	610856	54098	92813	146911	239209	2.55	4.16	332537	1.84
geo	102400	32346	21360	53706	70184	1.46	1.91	96426	1.06
news	377109	62228	52281	114509	162084	2.33	3.29	219071	1.72
obj1	21504	6407	2140	8547	10612	2.03	2.52	12827	1.68
obj2	246814	36522	25277	61799	88693	2.78	3.99	112353	2.20
paper1	53161	5412	7620	13032	20595	2.58	4.08	28272	1.88
paper2	82199	7209	13373	20582	33566	2.45	3.99	47328	1.74
pic	513216	27698	15104	42802	55800	9.20	11.99	73010	7.03
progc	39611	4561	5055	9616	14530	2.73	4.12	19726	2.01
progl	71646	4685	6411	11096	17615	4.07	6.46	23918	3.00
progp	49379	3454	4219	7673	11900	4.15	6.44	16111	3.06
trans	93695	8682	7364	16046	23537	3.98	5.84	30774	3.04

Table D.1: Essential Data to Compute Work Load of Various Modules.

- **lit**. The number of literals generated by the sliding window coder.
- **substr**. The number of substrings generated by the sliding window coder.
- **lit+substr**. The sum of the previous two figures.
- **compr size**. The size of the compressed file in bytes.
- **compr ratio**. The original size divided by the compressed size.
- **avg substr**. The average length of a substring (where literals are counted as substrings of length 1).
- **byte conv**. The number of conversions from bit fields to bytes where the byte conversions in the transmission of the tree description are not counted. Each length or literal takes one conversion, each address takes two (one for the Huffman encoded part, one for the rest).,
- **byte conv ratio**. The original size divided by the number of byte conversions.

All modules communicate using handshake protocols and therefore the device will function correctly if one of the modules is not fast enough. As the various rates are highly data dependent, these things will happen at least for some data sets. It is however undesirable to have one small module as a bottle neck for typical data sets. It is therefore necessary to design the modules such that the work load is balanced for most typical data sets.

The files `pic` and `geo` are considered atypical as far as work load balancing is concerned, the other files should be considered typical. The modules should be designed such that they can cope with the various data rates implied by these files. The proposed sliding window coder can run at 1 uncompressed byte per clock cycle and can do so constantly. This is the module that will take most effort to increase the speed. Other modules should run at whatever speed it takes to cope with the other rates.

The relevant rates are:

- R_u. The rate at which uncompressed bytes arrive. This is the rate that all other rates are referenced to.
- R_c. The rate at which compressed bytes arrive. This is equal to R_u/K_c, where K_c is the compression ratio.
- R_s. The rate at which substrings or literals are processed. This is equal to R_u/K_s, where K_s is the average substring length.
- R_b. The rate at which byte conversions are performed. This is equal to R_u/K_b, where K_b is the byte conversion ratio.
- R_t. The rate at which new trees are computed. This is equal to $R_u/(K_s B)$, where B is the block size, which is 8192.

Given the fact that maximum utilization of the sliding window coder is desired, the K factors should be chosen close to their minimum typical values. We choose $K_c = 2$, $K_s = 3$ and $K_b = 1.5$.

We choose $R_u = 12.5$Mbyte/s and therefore $R_c = 6.25$Mbyte/s, $R_s = 4.2$ million substrings per second, $R_b = 8.3$ million byte conversions per second and $R_t = 500$ tree recomputations per second.

The above figures are all based on the results displayed in table D.1. The validity of the data in this table can be improved upon for the following reasons:

- The real circuit will use hardware lookahead in the sliding window compressor. This results in a longer average substring length.
- The byte conversions needed for the tree transmissions were not counted.
- For Huffman decoding we want to use an algorithm that uses more than one cycle when decoding long code words. We think of an algorithm that uses 1 cycle for code words less than 10 bits long and 1 additional cycle for every bit above the 9th bit in longer code words. It is believed that the number of decoding cycles will be only slightly higher than the number of byte conversions, but this needs to be tested.

Table D.2 shows the measured rates when using lookahead and counts the extra Huffman decoding cycles. The meaning of the figures is as follows:

- **byte conv**. The number of byte conversions, not counting those in the transmission of tree descriptions.

file	size	lit	substr	avg length	compr size	compr ratio	byte conv	decoding cycles	decodings extra
bib	111261	11538	14436	4.28	41303	2.69	54846	56624	583
book1	768771	68143	139435	3.70	342138	2.25	486448	497388	4562
book2	610856	50076	89103	4.39	230332	2.65	317385	327292	4350
geo	102400	26918	22523	2.07	68558	1.49	94487	104857	6716
news	377109	52082	52801	3.60	156363	2.41	210485	217885	2849
obj1	21504	6214	2039	2.61	10367	2.07	12331	13695	755
obj2	246814	34280	24245	4.22	85224	2.90	107015	117256	4951
paper1	53161	4967	7330	4.32	19805	2.68	26957	27994	434
paper2	82199	6871	12781	4.18	32390	2.54	45214	46651	633
pic	513216	25439	14633	12.81	53304	9.63	69338	77214	3519
progc	39611	4115	4930	4.38	14043	2.82	18905	19653	252
progl	71646	4186	6231	6.88	17037	4.21	22879	24195	472
progp	49379	3208	4107	6.75	11566	4.27	15529	16402	325
trans	93695	7926	7314	6.15	22954	4.08	29868	31736	691

Table D.2: Compression and Rate Statistics for Lookahead Compressor.

- **decoding cycles**. The number of cycles used by the static Huffman decoder, which is the number of byte conversions plus the number of extra cycles spent on long code words. This metric does include the byte conversions used in reception of tree descriptions. The inclusion of these extra cycles leads to a typical increase of less than 10% over the 'byte conv figure'.

- **decodings extra**. The number of Huffman decodings that involve code words longer than 9 bits. This is only a small fraction of the total number of Huffman decodings.

Comparing the two tables one can conclude the following:

- The average substring length is about 0.2 longer with lookahead than without. This results in a slightly lower byte conversion rate.

- The Huffman decoding rate is in some cases slightly higher and in some cases slightly lower than the byte conversion rate in D.1.

- The typical K values would not differ much when they are based on table D.2.

Table D.3 shows the same data for `gdcompr` instead of `dcompr`. The *byte conv* value is not included in this table. The average string length and the compression ratio are almost identical in both cases. The number of cycles spent in the Huffman coder or decoder is slightly higher as `gdcompr` employs a class-offset system for lengths and `dcompr` does not. Hence some of the lengths consist of two parts. The number of decoding operations that require more than one cycle is less with `gdcompr` because the average code word length is smaller.

file	size	lit	substr	avg length	compr size	compr ratio	decoding cycles	decodings extra
bib	111261	11538	14436	4.28	41246	2.70	58294	375
book1	768771	68143	139435	3.70	341995	2.25	500488	4458
book2	610856	50076	89103	4.39	230097	2.65	334500	3751
geo	102400	26918	22523	2.07	68458	1.50	104455	6722
news	377109	52082	52800	3.60	156073	2.42	220559	2004
obj1	21504	6214	2039	2.61	10332	2.08	13695	719
obj2	246814	34280	24245	4.22	84952	2.91	120123	4204
paper1	53161	4967	7330	4.32	19780	2.69	28705	424
paper2	82199	6871	12781	4.18	32381	2.54	47543	551
pic	513216	25440	14628	12.81	52696	9.74	76733	973
progc	39611	4115	4930	4.38	14035	2.82	20203	164
progl	71646	4186	6231	6.88	16975	4.22	25200	292
progp	49379	3209	4106	6.75	11531	4.28	17008	161
trans	93695	7926	7313	6.15	22859	4.10	32610	315

Table D.3: Compression and Rate Statistics for Lookahead Compressor (gdcompr).

Table D.4 shows the number of coding cycles needed for dynamic Huffman codes. The number of coding cycles is based on the assumption that there are two dynamic coders working in parallel, one for the lengths and literals and one for the addresses. They have to wait for each other to finish before the next length or literal can be coded. The coders use two cycles for each coding or decoding operation and four more cycles for each node swap. This is the number of cycles used by the hardware dynamic coder described in A.3 (dynamic Huffman coding).

D.2 Sliding Window Coding

All sliding window coding simulations are based on the hardware algorithms described in section A.1.3 (proposed hardware algorithm).

This section describes the following aspects of the sliding window coder:

- The following implementation options:
  - The logic diagrams of the cells for greedy and non-greedy matching.
  - The implementation of other parts of the coder, such as priority encoders.
  - The linear and the matrix layout of the cell array.
  - The transistor level implementation of the cells.
- Behavioural simulation at the register transfer level. The non-greedy matching algorithm is simulated this way.

file	size	compr size	compr ratio	coding cycles
bib	111261	41318	2.69	94726
book1	768771	341211	2.25	619634
book2	610856	229907	2.66	441474
geo	102400	68217	1.50	244786
news	377109	155806	2.42	348242
obj1	21504	10381	2.07	67139
obj2	246814	85218	2.90	308964
paper1	53161	19930	2.67	52760
paper2	82199	32539	2.53	71763
pic	513216	52829	9.71	143445
progc	39611	14147	2.80	45116
progl	71646	17109	4.19	48201
progp	49379	11679	4.23	38774
trans	93695	23032	4.07	71150

Table D.4: Compression and Rate Statistics for Dynamic Huffman Coding.

- Timing simulations with Spice and Compass of a single cell for the greedy matching algorithm.

While the other parts of the circuit are designed at the register transfer level (from there the implementation would be created by automated tools), the cells of the sliding window coder were also designed at lower levels: the gate level, the transistor level and the layout level.

D.2.1 Greedy Matching.

Figure D.1 shows the basic logic diagram of the array cell for greedy matching. All other sliding window coder variations are based on it. The 'match' output is the logical AND of the flag flipflop and the comparator output. The signal that is clocked into the flag flipflop is either the output of the comparator or the 'match' output, depending on the 'first/next' signal. The register and the flipflop are clocked and both can either hold their original value, load a new value or reset their contents to zero at the active edge of the clock.

A tristate buffer can be added that connects the register output to the match bus. The buffer of exactly one cell in the array can be enabled by the output of an address decoder. This facility makes it possible to use the array for decompression.

Figure D.2 shows the cell array with the priority coder and address decoder. This is a one-dimensional array in which the priority encoder and address decoder have a connection for every cell. For large arrays (8192 cells) the priority encoder and address decoder become very large.

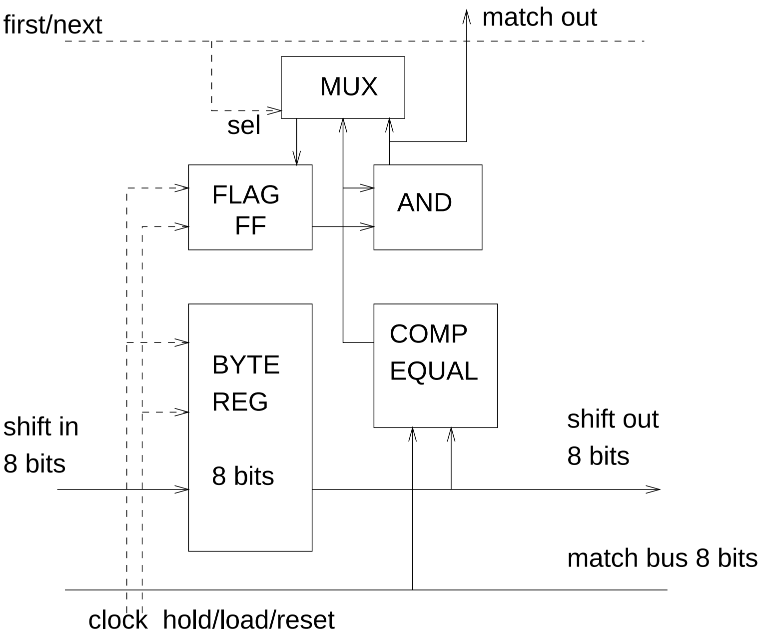

Figure D.1: Logic Diagram of Greedy Matching Cell.

It is possible to lay out the array as a matrix in such a way that much smaller priority encoders and address decoders can be used. Each cell has two match outputs: a row match output and a column match output. Instead of one read select input, there are two select inputs: a row select input and a column select input. There are separate priority encoders for rows and columns. The row match output is the same as the match output in a linear array. The column match output is the logical AND of the row match output and the column select input. Figure D.5 shows a cell that has everything including the read buffer, the matrix signals and non-greedy matching.

Figure D.3 shows the matrix layout. Note the following:

- An array of 8192 cells has 32 rows and 256 columns.
- The cells in the row 0 are laid out from left to right, the cells in row 1 are laid out from right to left. Row 2 is again laid out from left to right etc. This way the shifted cell contents never have to travel a long distance from one cell to the next.
- The 'row match' signals of the cells in one row are combined with a (wired) OR and are connected to one input of the row priority encoder.
- Likewise the 'column match' signals of all cells in the same column are combined with a (wired) OR and are connected to one input of the column priority encoder.
- Determining the address of the first matching cell is a two-stage process. First the row with the first match is determined. Its 'row select' signal is activated and only the cells on the selected row activate the 'column match' signals. Next the first matching column is found out.

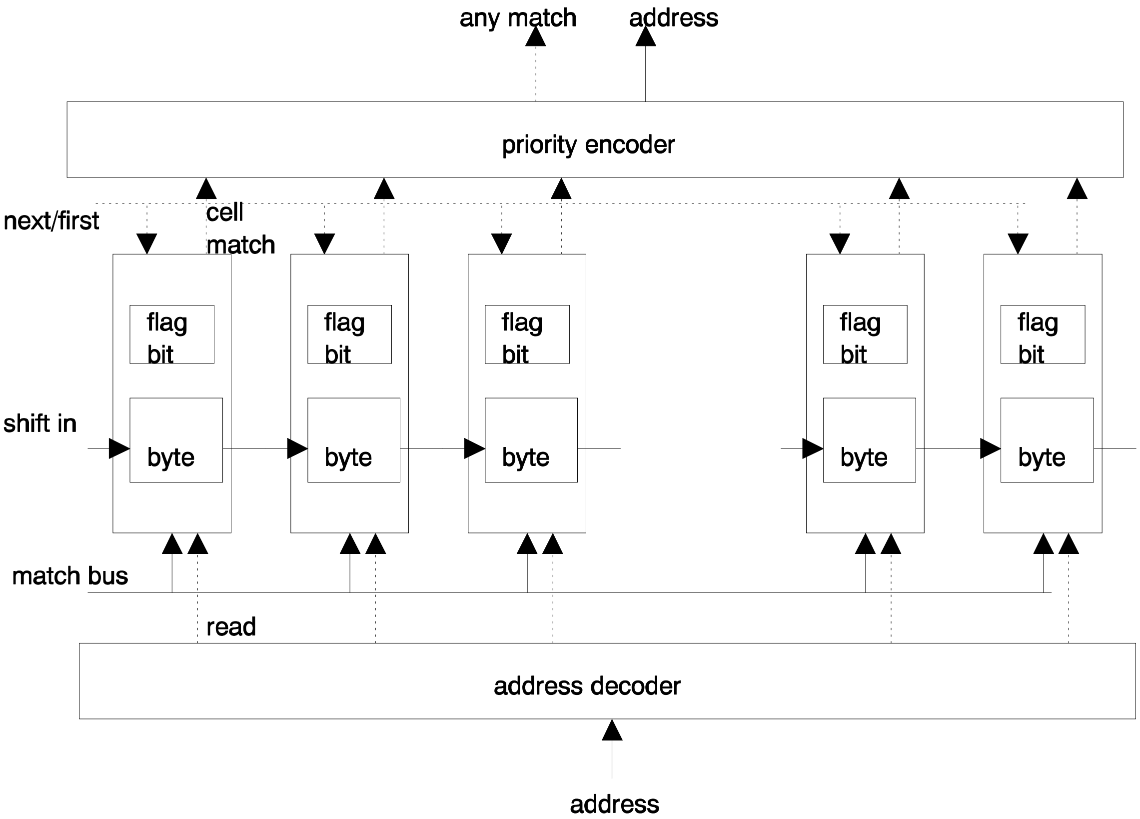

Figure D.2: Basic Diagram of Coding Array.

- Due to the fact that the odd rows are laid out mirrored with respect to the even rows, the wrong column is addressed by the address decoder and the wrong column is indicated by the priority encoder. To fix this the column address must be inverted on both the address decoder and the priority encoder whenever an odd-numbered row is addressed. A set of exclusive OR gates can be used.

- Despite the address correction the priority encoder would return the last matching column instead of the first one for odd-numbered rows. If absolutely necessary it could be fixed by adding a second priority encoder. However, the effect on compression ratio would be zero (LHA format) or negligible (gzip format). The contribution of the column address selection would only be significant for the first row (which is laid out normally).

- The cells in odd rows are physically laid out mirrored with respect to those in even rows.

The priority encoder can be implemented in the following ways:

- Tree implementation. A priority encoder of size $2N$ (N is a power of 2) can be constructed from two priority encoders of size N, a N-input NOR and $\log_2 N$ 2-input

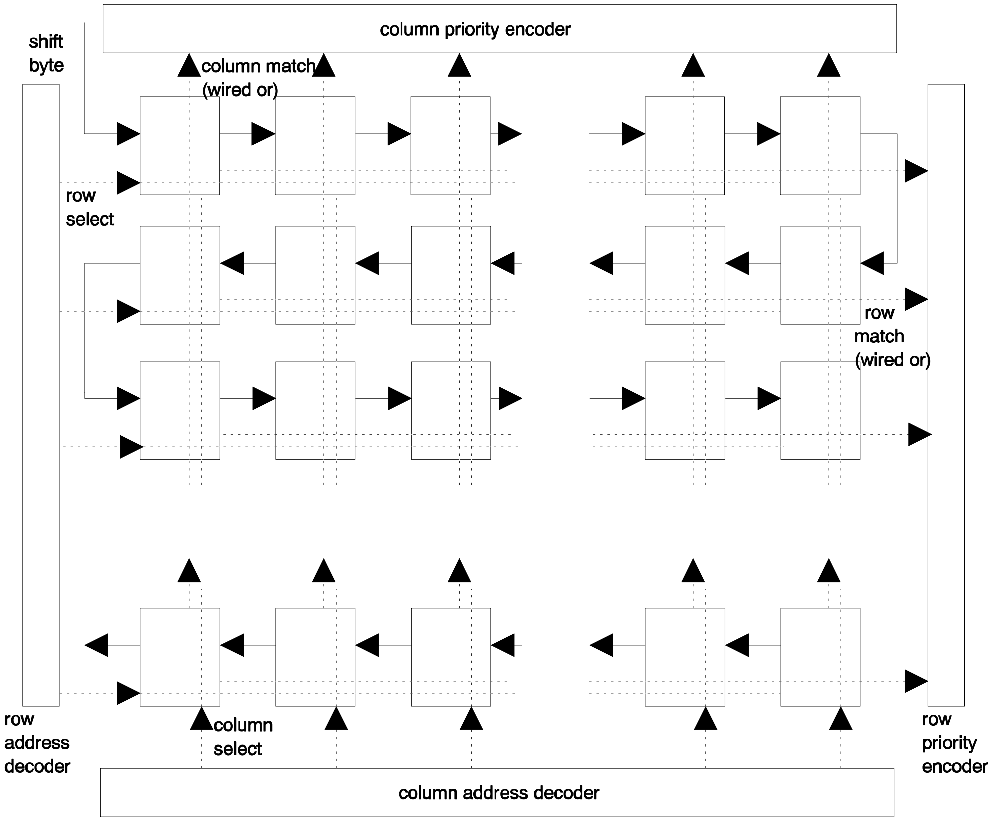

Figure D.3: Matrix layout of Array.

multiplexers. The multiplexers select the $\log_2 N$ address bits of one of the priority encoders, depending on the output of the NOR-gate. This output functions as a new address bit. This implementation can easily be described in languages such as VHDL. Larger NOR gates must be implemented as a tree of gates.

- Disabler and address encoding. The disabler has as many outputs as inputs. At most one output can be set, the output corresponding to the first input that is set.

  For each bit of the address there is a large AND gate whose inputs are connected to all the disabler outputs whose address has the corresponding bit set.

  This implementation is very suitable for layout as a regular structure. It can be laid out along the side of a large cell array. The address encoding AND gates can be similar to the AND field of a PAL.

  An additional advantage is that in the matrix implementation of the sliding window coder the disabler outputs can be connected directly to the row select signals.

  The disabler itself can be implemented as follows:

  – ripple through. A disabling signal is set by the first cell whose input is set and it ripples through the disabler, disabling the output of all subsequent cells whose input is set. As the signal must ripple through all the cells, this implementation is unsuitable for large coders.

– hierarchical set of OR gates. Each cell generates its own disabling signal, which is the logical OR of the inputs of all preceding cells. Large OR gates (whose number of inputs is a power of two) are shared among the cells.

The priority coder with a disabler is described in [DV92].

Address decoders can be implemented in the usual way with a large AND field with all the address bits and their inverses as input.

D.2.2 Non-greedy Matching.

The greedy algorithm can be simulated straightforwardly. The non-greedy algorithm (one byte lookahead) is more complicated. Therefore we created the C program `lzsim.c` to simulate the behaviour of the cell array.

An algorithm of an associative cell array can be simulated in C as follows:

- All state variables are modelled with C variables.
- All signals are modelled with C variables.
- The signals and state variables of the cells in the cell array are modelled as arrays of C variables (or as a single array of C structs).
- A sequential algorithm is created that brings about all changes in the signals and state variables. This algorithm iterates over all cells in the array. Each clock cycle of the cell array may require several passes over all the cells. In the typical case one pass iterates over all the cells to collect all the outgoing signals of the cells and create the relevant global signals. Another pass sets the state variables of all cells, depending on the global signals that are distributed to all cells.
- Some ad-hoc strategies are used to model the priority encoder.

In the `lzsim.c` program the cell behaviour is concentrated in a single procedure 'allcells'.

```
/* lzsim.c -- Simulation of cell array for sliding window coding */
#include <stdio.h>

#define WINDOWSIZE 8192

char bytes[WINDOWSIZE];
char flag1[WINDOWSIZE];
char flag2[WINDOWSIZE];

#define MAXMATCH 256
```

```
#define EOB 256

void shiftin(char c)
{
 int i;
 for(i=WINDOWSIZE-1;i>0;i--) {
  bytes[i]=bytes[i-1];
 }
 bytes[0]=c;
}

#define CYC1A 0
#define CYC1B 1
#define CYC2A 2
#define CYC2B 3

char matchbus,anymatch;
int matchaddr;

void allcells(int code)
{
 int i;
 char b;
 for(i=0;i<WINDOWSIZE;i++) {
  switch(code) {
   case CYC1A:
        b=flag1[i]&&(matchbus==bytes[i]);
        if(!anymatch&&b)matchaddr=i;
        anymatch|=b;
        break;
   case CYC1B:
        b=matchbus==bytes[i];
        if(anymatch) {
          flag1[i]&=b;
          flag2[i]&=b;
        } else {
          flag1[i]=b;
          flag2[i]&=b;
        }
        break;
   case CYC2A:
        b=flag2[i]&&(matchbus==bytes[i]);
        if(!anymatch&&b)matchaddr=i;
        anymatch|=b;
        break;
   case CYC2B:
        b=matchbus==bytes[i];
        if(anymatch) {
          flag1[i]=flag2[i]&b;
          flag2[i]=0;
        } else {
```

```
          flag1[i]&=b;
          flag2[i]=b;
        }
        break;
  }
 }
}

FILE *infile,*outfile;
char *inname,*outname;

void openfiles(void)
{
 if((infile=fopen(inname,"rb"))==NULL) {
  fprintf(stderr,"Cannot open input file\n");
  exit(2);
 }
 if((outfile=fopen(outname,"wb"))==NULL) {
  fprintf(stderr,"Cannot open output file\n");
  exit(2);
 }
}

void sendout(short l,short a)
{
 fwrite(&l,sizeof(l),1,outfile);
 if(l>EOB) fwrite(&a,sizeof(a),1,outfile);
}

int main(int argc,char **argv)
{
 int c,matchlen,oldmatchaddr;
 unsigned char cyc2next,lit1,lit2,oldlit1,oldlit2;
 inname=argv[1];
 outname=argv[2];
 openfiles();
 cyc2next=0;
 matchlen=0;
 while((c=getc(infile))!=EOF) {
  oldmatchaddr=matchaddr;
  matchbus=c;
  if(cyc2next) {
    oldlit2=lit2;
    lit2=c;
    anymatch=0;
    allcells(CYC2A);
    if(matchlen==MAXMATCH)anymatch=0;
    allcells(CYC2B);
    if(anymatch) {
      sendout(oldlit1,0);
      matchlen++;
```

```
    } else {
      switch(matchlen) {
       case 0: break;
       case 1: sendout(oldlit1,0);
               break;
       case 2: sendout(oldlit1,0);
               sendout(oldlit2,0);
               break;
       default: sendout(matchlen+255,8191-oldmatchaddr);
      }
      matchlen=2;
    }
    cyc2next=0;
  } else {
    anymatch=0;
    allcells(CYC1A);
    if(matchlen==MAXMATCH)anymatch=0;
    allcells(CYC1B);
    if(anymatch) {
     matchlen++;
    }else{
     cyc2next=1;
     oldlit1=lit1;
     lit1=c;
    }
  }
  shiftin(c);
 }
 if(!cyc2next) {
  oldlit1=lit1;
  oldlit2=lit2;
 }
 switch(matchlen) {
  case 0: break;
  case 1: sendout(oldlit1,0);
          break;
  case 2: sendout(oldlit1,0);
          sendout(oldlit2,0);
          break;
  default: sendout(matchlen+255,8191-matchaddr);
 }
 if(cyc2next)sendout(lit1,0);
 sendout(EOB,0);
 exit(0);
}
```

Instead of a single 'first/next' signal, there are two signals 'Cycle' and 'Continue'. The 'Cycle' signal determines whether the first or the second string should be matched. The 'Continue' signal determines whether the string should be continued or a new one should be started. To save a few gates per cell the 'Continue' signal can best be exclusive-ORed with 'Cycle'. Table D.5 shows the input signals for the flag flipflops as functions of these signals.

Cycle	Continue	Xcont	Flag1 in	Flag2 in
1	0	0	equal	m2
2	1	0	m2	0
1	1	1	m1	m2
2	0	1	m1	equal

Table D.5: Control Signals in Non-Greedy Matching Cell

Figure D.4 shows the diagram of the cell in a linear array. The non-greedy matching cell can also be used in a matrix. Figure D.5 shows a cell for a matrix complete with read logic.

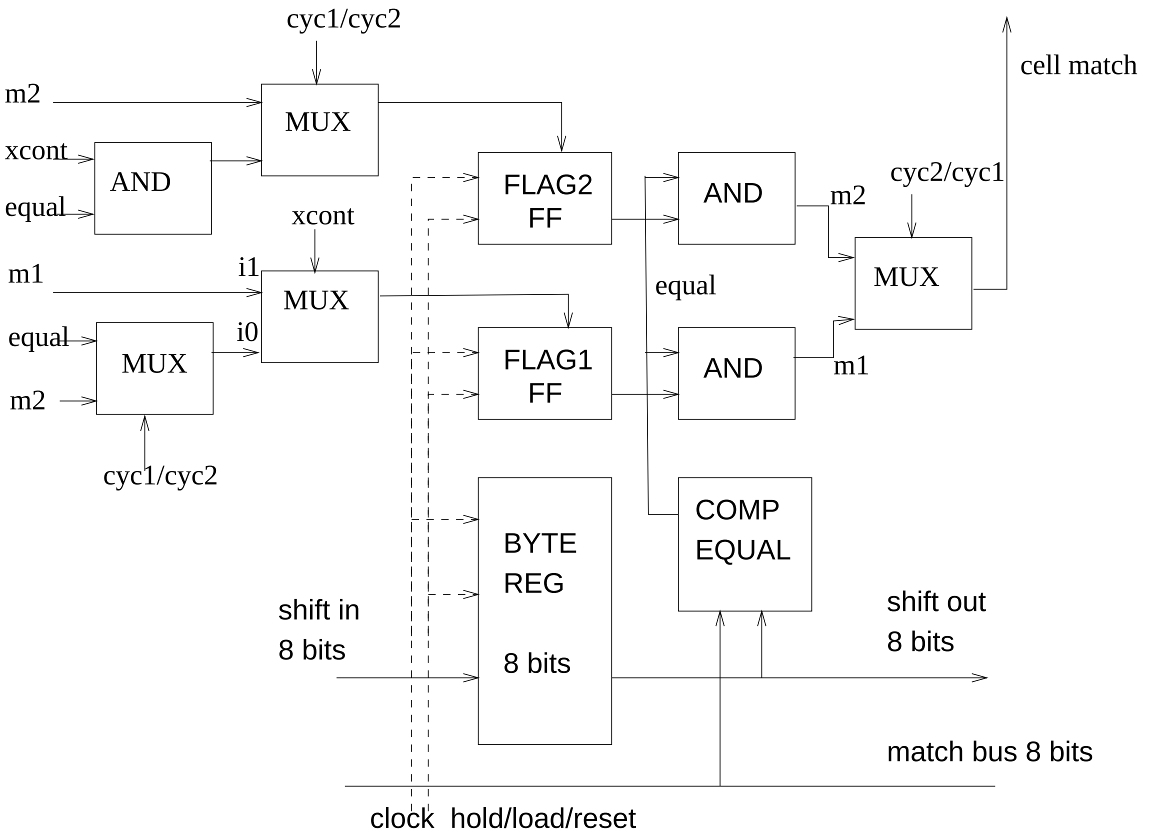

Figure D.4: Logic Diagram of Non-Greedy Matching Cell.

Table D.6 shows the gate count per cell for several variations of the array. The register, the comparator and the tristate buffer (if any) are all 8 bits wide. All cells may include hardware to enable reading their contents so that the array can be used for decompression. This feature is optional in both the linear and the matrix versions (in table D.6 it is only shown for the matrix versions). Omitting this feature saves the tristate buffer and (for the matrix versions) one AND port.

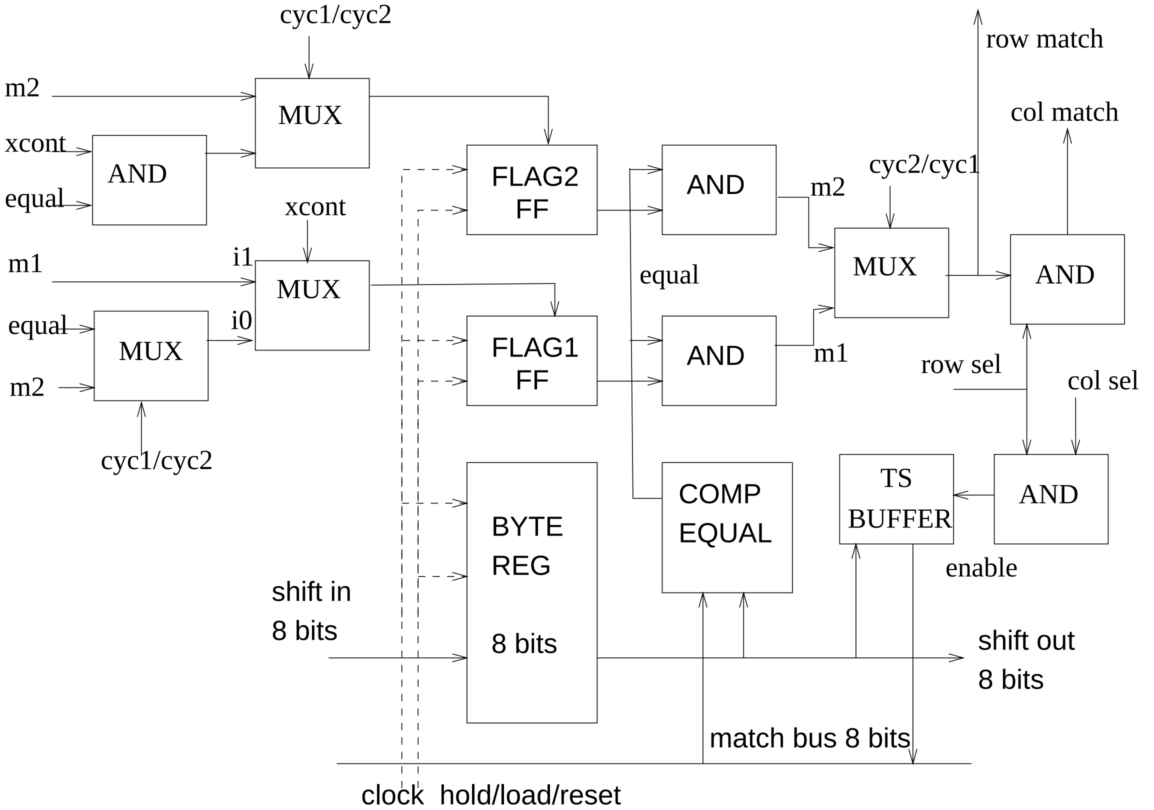

Figure D.5: Logic Diagram of Non-Greedy Matching Cell in Matrix.

D.2.3 Layout of Single Cell.

Initially we tried to implement the sliding window coder (with greedy matching) as a full-custom design. We implemented a cell from this array in CMOS using the sticks layout tool. We had the following reasons for that:

- Every manually designed cell appears 8192 times on the chip, so that every square micron saved is multiplied by 8192.
- The array depends on a large number of OR gates with many inputs. These should be implemented as wired or gates and be distributed over the array in a special way.

Cell Type	Register	Comparator	Tristate buffer	Flipflop	AND	MUX
Greedy, linear	1	1	0	1	1	1
Greedy, matrix	1	1	1	1	3	1
Nongreedy, linear	1	1	0	2	3	4
Nongreedy, matrix	1	1	1	2	5	4

Table D.6: Gate Count for Cells.

- Silicon compilers do not seem to take advantage of the fact that all cells are identical and their interconnection pattern is very regular. Silicon compilers seem to reserve large routing areas where the cells could abut and all necessary wiring could be routed through the cell without taking extra area.

We implemented all flipflops as dynamic CMOS master-slave register cells (shown in figure D.6. Such a cell has two inverters, each of which will keep its state (electrical charge) for a short time while its input is disconnected. During the first clock phase the input of the first inverter (master) is connected to the register input (or to the output if it must hold its present value). During the second phase the input of the second inverter (slave) is connected to the output of the first inverter. The transistors that are drawn horizontally act as pass transistors.

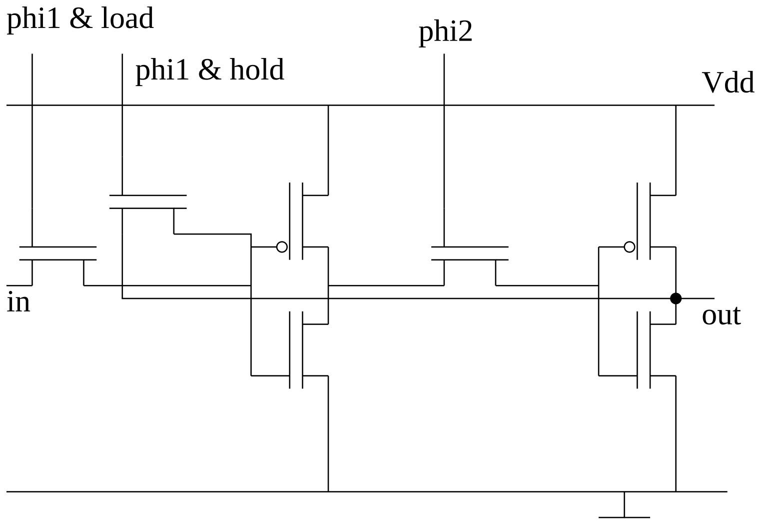

Figure D.6: Schematic of Dynamic CMOS Flipflop.

The OR gates for the row and column match signals (and the internal NOR gate of the 8-bit comparator) are implemented with precharged logic. En example is shown in figure D.7. During one phase of the clock the gate of the P transistor is set to VSS (it conducts) as are the gates of all the N transistors (they do not conduct). This way the capacity of the wire that connects all transistors gets a positive charge. During the other phase of the clock the gate of the P transistor is set to VDD (it does not conduct) and each cell puts a logic signal on the gate of its N transistor. When any N transistor conducts, the capacity gets discharged, otherwise it will remain charged. At the end of phase 2 the result of the OR operation is available.

Figure D.8 shows a single bit slice of the 8-bit register and the comparator. There are 'weak' P transistors added to the inverters. These are supposed to pull the input up and they make the circuit more symmetric with respect to 0 and 1. Further the 'phi1' signal has a separate pass transistor from the 'load' and 'hold' signals. The transistor on the extreme right implements the read buffer. It can connect the register output to the match bus.

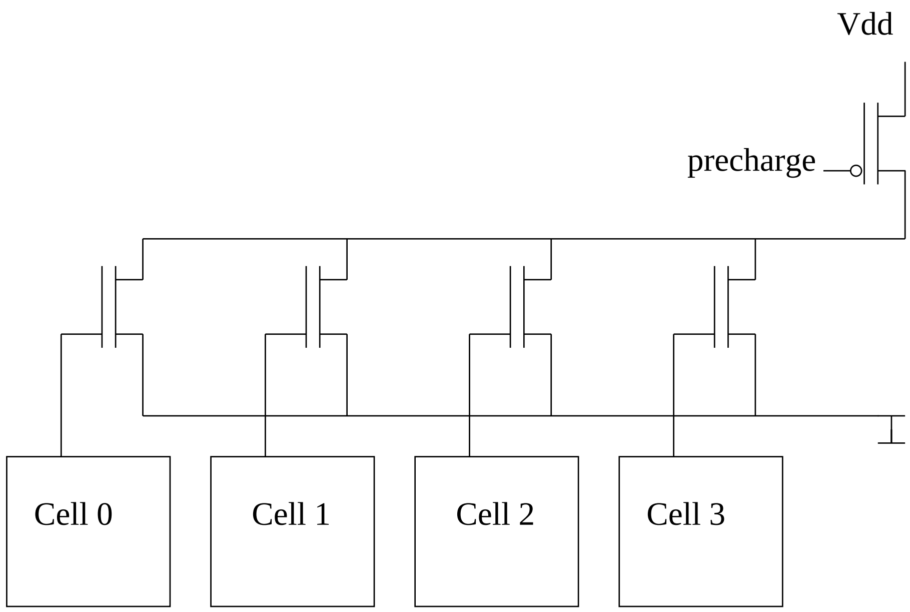

Figure D.7: Schematic of Precharged CMOS Wired Or.

The 'match' signal and its inverse are one bit of the byte on the match bus. The 'comp' signal is the central node in an 8-bit wired NOR gate for the comparator. Each bit slice contains one pull-down transistor. Two transistors are connected in series between the match signal and its inverse. They are controlled by the register value and its inverse. On the node between the two transistors one finds the exclusive or of the match value and the register contents. This node is connected to the comparator pull-down transistor.

Further we made a control unit, containing the pull-up transistor of the comparator NOR, a multiplexer, the flag flipflop and several simple AND and NAND gates. This control unit was placed on top of the 8 bit slices.

The 'phi1', 'phi2', 'hold', 'load' and 'reset' signals are routed vertically through the entire array. The signals 'comp' and 'read' are routed through the 8 bits of a single register. There are 8 signals on the match bus. They and their inverses are routed horizontally through each of the bit slices. The 'out' signals are connected to the corresponding 'in' signals of the next cell in the row.

The 'column match' and 'column select' signals are not used by the bit slices, but they are routed vertically throughout the array. The signals 'row match', 'row select' and the first-next signal (it is a greedy mathcing cell) and its inverse are routed horizontally through the control unit. Further a scan path connects all match flipflops in the same row.

We designed the cell such that it would abut the cells to the left and right of it in the same row (same orientation) and that it would abut the mirrored cells in the rows above and below it.

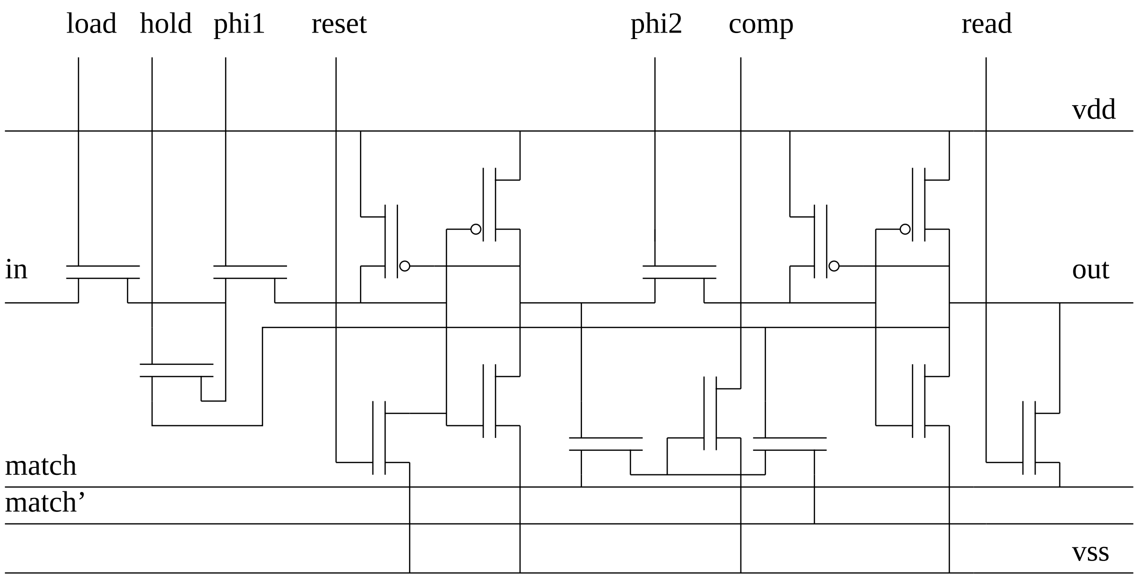

Figure D.8: Schematic of Bit Slice.

The cell had an area of 0.12mm^2 in a 1μm CMOS process. An array of 8192 cells would take 1000mm^2. The layout is shown in figure D.9. If for a 0.35μm process each cell is a factor 8 smaller, then it would take 125mm^2, which can certainly be fabricated.

The single cell was simulated both with the timing simulator of Compass and with Spice. The cell was tested with a clock frequency of 10MHz. Delays were such that it would still work at 20MHz, but the delay would be dominated by the path through the priority encoder, which is not measured. Power dissipation was 0.4mW per cell. The whole array would dissipate 3.2W.

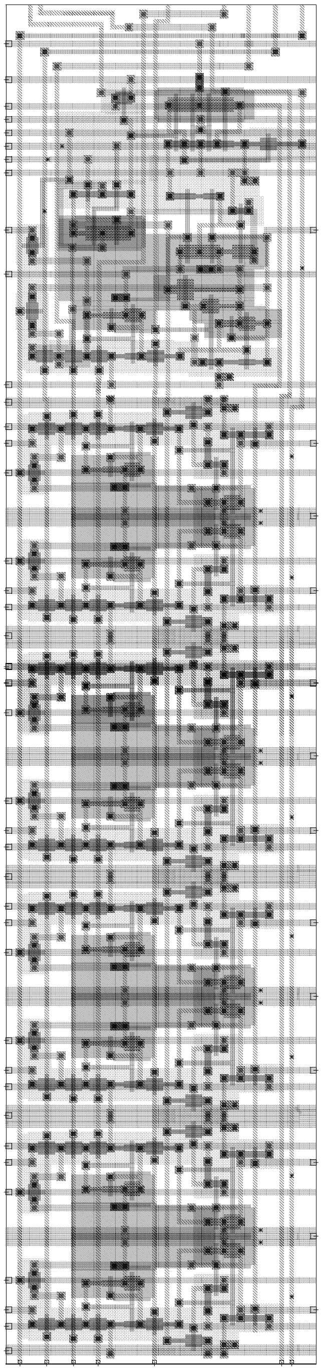

Figure D.9: Layout of Single Cell.

D.3 Byte Conversion

Byte conversion is the task of converting a stream of variable length bit fields (of 1 to 16 bits) to a stream of 8-bit bytes. This is an essential step of both Huffman coding and decoding.

Very early in the design process we designed and simulated both a byte encoder (from bit fields to bytes) and a byte decoder (from bytes to bit fields). Both circuits were simulated in IDaSS. The data were carried on one-phase handshake channels. Both modules would take at least two clock cycles per conversion.

The byte encoder contains a 23-bit register to hold a partially converted bit field, a count register and a shift operator that accepts a variable shift count. Before conversion the conversion register contains from 0 to 7 bits. The bit field adds up to 16 bits to it (register contains up to 23 bits). Next the converter outputs 0, 1, or 2 bytes until the register contains again less than 8 bits.

The byte decoder is very similar to the byte encoder. Note that the bit field length is set by the module that consumes the bit fields.

D.4 Static Huffman Tree Computation

For the simulations we computed the length/literal code of the LHA-based format, which has 512 symbols.

The program 'linhuf.c' computes this code for the first block of an LZ77-compressed file. Both the standard method (using the heap data structure) and the Lu/Chen algorithm could be selected. Both methods were timed.

The code fragment below shows the 'freqs_to_lengths2' function that computes the code word lengths (array 'lengths') from the symbol frequencies (array 'freqs'). Some of the variables have the same names as in the original article of Lu and Chen [LC90].

```
#define NULINDEX 0xffff
#define SETSIZE 512

unsigned short freqs[SETSIZE]; /* Frequencxies of all symbols */

char lengths[SETSIZE];   /* Code word lengths for symbols */

freqs_to_lengths2()
{
 /* Code word length computation using Lu/Chen algorithm. */
 listnode binlist[SETSIZE],binlist1[SETSIZE];
 unsigned short binheads[32],binheads1[16],bintails[32];
 int i,a,b,m,lp,mcl;
 unsigned short tot,aux;
 tot=0;
 /* Sort the frequencies using a three-pass 16/32-way bin sort. */
 for(i=0;i<=15;i++)binheads[i]=NULINDEX;
```

```
for(i=0;i<SETSIZE;i++) {
 if(!freqs[i]) {
   lengths[i]=0;
   continue;
 }
 tot++;
 a=freqs[i]&15;
 if(binheads[a]==NULINDEX)binheads[a]=i;
                  else binlist[bintails[a]]=i;
 bintails[a]=i;
 binlist[i]=NULINDEX;
}
/* second sort pass starts here. */
for(i=0;i<=15;i++) binheads1[i]=NULINDEX;
for(i=15;i>=0;i--) {
 b=binheads[i];
 while(b!=NULINDEX) {
  a=(freqs[b]>>4)&15;
  if(binheads1[a]==NULINDEX)binheads1[a]=b;
                     else binlist1[bintails[a]]=b;
  bintails[a]=b;
  binlist1[b]=NULINDEX;
  b=binlist[b];
 }
}
/*second sort pass ends here. */
for(i=0;i<=31;i++) binheads[i]=NULINDEX;
for(i=15;i>=0;i--) {
 b=binheads1[i];
 while(b!=NULINDEX) {
  a=(freqs[b]>>8)&31;
  if(binheads[a]==NULINDEX)binheads[a]=b;
                     else binlist[bintails[a]]=b;
  bintails[a]=b;
  binlist[b]=NULINDEX;
  b=binlist1[b];
 }
}
/* It's been three passes through the sorter, read them out and set
   sorted frequencies. */
a=0;
for(i=31;i>=0;i--) {
 b=binheads[i];
 while(b!=NULINDEX) {
  sfreqs[a]=freqs[b];
  b=binlist[b];
  a++;
 }
}
/* Now we can use the Lu/Chen algorithm. */
/* Contraction stage */
```

```
 i=1;lp=tot-3;m=0;aux=sfreqs[tot-1]+sfreqs[tot-2];
 while(1) {
  if(aux >= sfreqs[lp]) {
    sfreqs[lp+tot-i-2]=sfreqs[lp];m++;lp--;
    if(lp<0)break;
  } else {
    sfreqs[lp+tot-i-2]=aux; expind[i++]=m;
    if(i==tot-2)break;
    if(lp==tot-i-1) {aux=sfreqs[lp]+sfreqs[2*(tot-i-1)];lp--;}
             else {aux=sfreqs[2*(tot-i)-3]+sfreqs[2*(tot-i-1)];m--;}
  }
 }
 while(i<tot-2) {
  expind[i]=m;m--;i++;
 }
 /* Expansion stage, length generation */
 nclass[1]=1;mcl=2;
 nclass[mcl]=2;i=tot-3;
 while(i) {
  if(expind[i]<nclass[mcl]) {
   nclass[mcl]--;mcl++;nclass[mcl]=2;
  } else {
   nclass[mcl-1]--;nclass[mcl]+=2;
  }
  i--;
 }
 /* fill in the lengths of the actual code words. */
 a=1;
 for(i=31;i>=0;i--) {
  b=binheads[i];
  while(b!=NULINDEX) {
   while(!nclass[a])a++;
   lengths[b]=a;
   nclass[a]--;
   b=binlist[b];
  }
 }
}
```

The whole Lu/Chen algorithm (including code word generation) was translated into assembly language. The following is a straightforward translation of pass 2 of the sort algorithm in the assembly language of the internal processor. The entire program can be found in [Wij96]. The b0 register contains the start of the array of symbol frequencies (freqs), b2 contains binheads, b3 contains binlist, b4 contains bintails, b5 contains binheads1, b6 contains binlist1 and r6 contains the value 0ffffh (NULINDEX)

```
003F:               ; Prepare for pass 2 of the bin sort.
003F: 43C2                          mov r2,15
0040: EA86                          fill b5[r2],r6 ;Set binheads1 array.
0041: 43C1                          mov r1,15
0042: 0442          pas2loop        mov r2,b2[r1]
0043: A0C8                          js pas2next
0044: 0083          pas2inn         mov r3,b0[r2]
0045: C043                          sel2 r3
0046: 0AC5                          mov r5,b5[r3]
0047: A029                          jns pas2else
0048: 0ACA                          mov b5[r3],r2
0049: 804C                          jmp pas2endi
004A: 08C5          pas2else        mov r5,b4[r3]
004B: 0D4A                          mov b6[r5],r2
004C: 08CA          pas2endi        mov b4[r3],r2
004D: 0C8E                          mov b6[r2],r6
004E: 0682                          mov r2,b3[r2]
004F: AF49                          jns pas2inn
0050: 4059          pas2next        dec r1
0051: AF09                          jns pas2loop
```

Table D.7 shows the timing results for two files from the Calgary Corpus. The time for the linear or standard algorithm does not include the time to generate code words from lengths (but it does include sorting for the linear algorithm). The C programs were executed on a HP7000 work station that executed around 60 million instructions per second for the given code. We see that the linear algorithm (Lu/Chen) is 3 to 4 times as fast as the standard (heap) algorithm.

file	progc	obj1
file type	text	binary
number of nonzero frequencies	148	294
timing of standard algorithm in C	1.73ms	3.92ms
timing of linear algorithm in C	0.49ms	0.97ms
timing of 'lengths to codes' in C	0.15ms	0.21ms
timing of linear algorithm in assembler	17111 instr	30230 instr
timing of 'lengths to codes' in assembler	4165 instr	5624 instr

Table D.7: Timing for Huffman Coding Algorithms.

From the rate statistics we know that 500 static Huffman blocks are generated per second, which gives a total time budget of 2ms per block. Before simulating the internal processor we estimated that it would execute the given algorithm twice as slow as the work station. Therefore it would take 1.3ms (linear algorithm plus lengths to codes) for the text file and 2.4ms for binary file. It would be adequate for text files, but it would be too slow for binary files. Speed optimizing the internal processor would therefore be very important.

The assembler implementation was run on an instruction-level simulator of the internal processor and the number of instructions was determined in each case.

The processor was implemented in IDaSS and automatically converted to VHDL and using the Compass tools a layout was synthesized, which is shown in figure D.10. It measures 5.83×5.95mm in a 1μm CMOS process.

From an IDaSS simulation of the processor we know that nearly 60% of the cycles can be wasted on stalls if the program contains many tight loops. If the processor runs at 50MHz, it would execute about 20MIPS. By partially unrolling of loops and branch optimization it should be possible to boost the performance to 25MIPS or more. If the processor runs at 25MIPS, text and binary files should take 0.85ms and 1.4ms per block respectively.

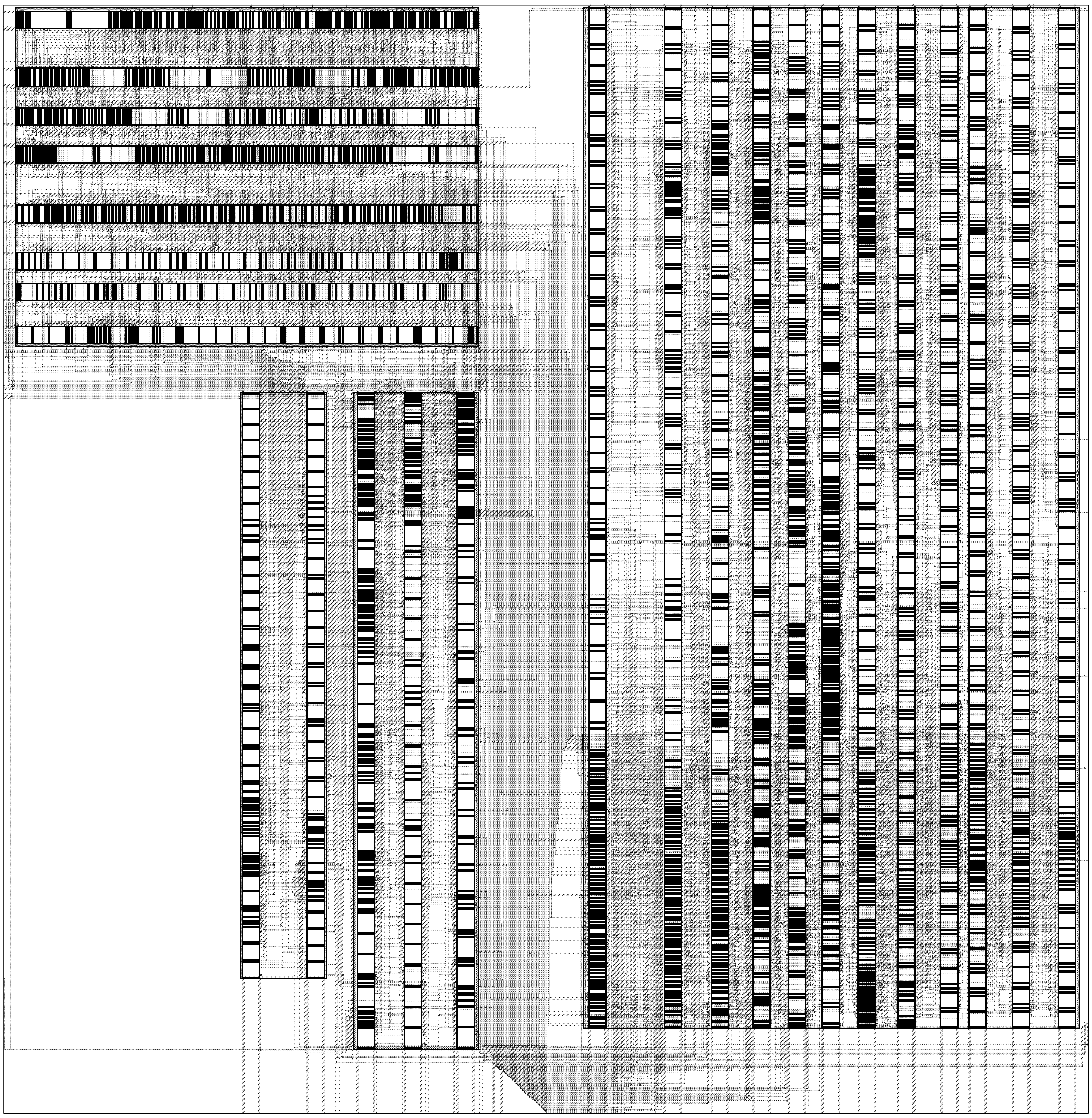

Figure D.10: Layout of Internal Processor.

D.5 Dynamic Huffman Coding

The dynamic Huffman coding simulation programs compress and decompress a file using dynamic Huffman coding with a zero order model of the 256 bytes plus an end-of-file symbol.

First we implemented the traditional algorithm in C (ahuf.c). It uses a tree whose nodes are doubly linked with pointers. The same algorithm was later used in the 'dcompr' program.

The program hufsim.c simulates the cell array algorithm for dynamic Huffman coding as described in section A.3.1 (dynamic Huffman algorithm in hardware)[2]. Its internal workings are similar to lzsim.c:

- The 'cells' array contains structures with all the per-cell signals and state variables.
- The global signals and state variables are simulated by C variables.

Both programs yielded exactly the same compressed data.

```
/* hufsim.c Dynamic Huffman coder simulation. */
#include <stdio.h>
#include<string.h>

#define SETSIZE 257 /* The alphabet size. */

/*#define DEBUG*/

FILE *infile,*outfile;
char* inname,*outname;

unsigned char inbyte,outbyte;
int incount=8,outcount=0;
/* The functions getbit() putbit() and flushbyte() are used for
   bit-oriented I/O to the compressed file. */

/* getbit() -- get next bit of compressed file. */
getbit() /* return 0,1 or EOF */
{
 int c;
 if(incount==8) {
   c=fgetc(infile);
   if(c==EOF)return EOF;
   comprsize++;
   inbyte=c;
   incount=0;
```

[2]This program simulates a less advanced version of the hardware array in which the cells who mark themselves for A-reception or B-reception remove their B and A code prefixes. For this operation and for adding the other code prefix later, they had to perform expensive shift operations. In the advanced version of the array, the cells record size of the B and A code prefixes in the MASK register and can later do with a 1-position shift for adding the other code prefix.

```
 }
 c=inbyte&1;inbyte>>=1;incount++;
 return c;
}

/* putbit -- add bit to compressed file. */
putbit(int b)
{
 if(outcount==8) {
  fputc(outbyte,outfile);
  comprsize++;
  outcount=0;
 }
 outbyte=(outbyte>>1)+(0x80& -b);
 outcount++;
}

/* flushbyte -- fill last byte so that it will be written to the file */
flushbyte()
{
 int i;
 for(i=outcount;i<=8;i++)putbit(0);
}

/* openfiles() opens the input and output files.
   During compression:
    infile contains uncompressed data, is read with getc(),
    outfile contains compressed data, is written with
                         putbit() and flushbyte().
   During decompression:
    infile contains compressed data, is read with getbit(),
    outfile contains uncompressed data, is written with putc() */

openfiles()
{
 if((infile=fopen(inname,"rb"))==NULL) {
  fprintf(stderr,"Cannot open input file\n");
  exit(2);
 }
 if((outfile=fopen(outname,"wb"))==NULL) {
  fprintf(stderr,"Cannot open output file\n");
  exit(2);
 }
}

unsigned short codebus,charbus,leafbus,successbus,countbus,maskbus;
unsigned short codebuf;

int cnmin;

refill()
```

```
{
 unsigned short mb=maskbus;
 while(mb!=0xffff) {
  codebuf=(codebuf<<1)+getbit();
  mb=(mb<<1)+1;
 }
}

sendout()
{
 unsigned short mb=maskbus;
 while(mb!=0xffff) {
  putbit(codebuf>>15);
  codebuf<<=1;
  mb=(mb<<1)+1;
 }
}

char* bincode(unsigned short c,unsigned short m)
{
 int i=0;
 static char buf[17];
 while(m!=0xffff) {
  buf[i++]=(c&0x8000)?'1':'0';
  m=(m<<1)+1;c<<=1;
 }
 buf[i]=0;
 return buf;
}

typedef struct
  { unsigned short code,mask,count,charr,leaf;
    char markinc,toolow,marka,markb,markar,markbr;} celltype;

celltype cells[SETSIZE*2-1];

typedef enum {INIT,ENCODE,DECODE,TESTA,TESTB,EX0,EX1A,EX1B,EX2} funcode;

do_cell(int cn,funcode fc)
{
 celltype *cp;
 cp=cells+cn;
 switch(fc) {
  case INIT:
    cp->charr=charbus;
    cp->count=countbus;
    cp->code=codebus;
    cp->mask=maskbus;
    cp->leaf=leafbus;
    cp->markinc=0;
    #ifdef DEBUG
```

```
    if(leafbus)
     printf("Setting cell #%03d to char %d, code %s\n",cn,charbus,
                  bincode(codebus,maskbus));
    else
     printf("Setting internal cell #%03d to code %s\n",cn,
                          bincode(codebus,maskbus));
   #endif
   break;
  case ENCODE:
   if(cp->leaf && charbus==cp->charr) {
    codebus=cp->code;
    maskbus=cp->mask;
    #ifdef DEBUG
     printf("Cell #%03d: encoding char %d to %s\n",cn,charbus,
                bincode(codebus,maskbus));
    #endif
   }
   if(cp->markinc) {
    cp->count++;
    #ifdef DEBUG
     printf("Incrementing cell#%03d to %d\n",cn,cp->count);
    #endif
   }
   cp->markinc=cp->toolow=0;
   break;
  case DECODE:
   if(cp->leaf && (cp->code|cp->mask)==(codebus|cp->mask) ) {
    maskbus=cp->mask;
    charbus=cp->charr;
    #ifdef DEBUG
     printf("Cell #%03d: decoding char %d to %s\n",cn,charbus,
                bincode(codebus,maskbus));
    #endif
   }
   if(cp->markinc) {
    cp->count++;
    #ifdef DEBUG
     printf("Incrementing cell#%03d to %d\n",cn,cp->count);
    #endif
   }
   cp->markinc=cp->toolow=0;
   break;
  case TESTA:
   cp->marka=cp->markb=cp->markar=cp->markbr=0;
   if((cp->code|cp->mask)==(codebus|cp->mask) &&
       !cp->toolow && (cn!=2*SETSIZE-2&&cp->count==(cp+1)->count)) {
    maskbus&=cp->mask;
    successbus=1;
   }
   break;
  case TESTB:
```

```
  if((cp->code|cp->mask)==(codebus|cp->mask)&&!cp->toolow) {
   if(!successbus||(cp->mask==(cp->mask&maskbus)))cp->markinc=1;
   if(cn!=2*SETSIZE-2&&cp->count==(cp+1)->count && maskbus==cp->mask) {
      cp->marka=1;
      #ifdef DEBUG
       printf("Marking cell #%03d as A cell, code %s\n",cn,
                        bincode(cp->code,cp->mask));
      #endif
      cp->markinc=0;
      charbus=cp->charr;
      countbus=cp->count;
      leafbus=cp->leaf;
   }
  }
  if(successbus&&!cp->marka&&(cp->code|maskbus)==(codebus|maskbus)
                  && cp->mask==(maskbus&cp->mask)) {
     unsigned short r=maskbus;
     cp->markbr=1;
     #ifdef DEBUG
      printf("Marking cell #%03d for B reception, code %s\n",cn,
                      bincode(cp->code,cp->mask));
     #endif
     while(r!=0xffff){r=(r<<1)+1;cp->code<<=1;cp->mask=(cp->mask<<1)+1;}
  }
  break;
case EX0:
  if(cn!=2*SETSIZE-2&&cp->count==countbus && (cp+1)->count > countbus) {
   cp->markb=1;
   cnmin=cn+1;
   #ifdef DEBUG
    printf("Marking cell #%03d as B cell, code %s\n",cn,
                    bincode(cp->code,cp->mask));
   #endif
   cp->markinc=1;
  }
  break;
case EX1A:
  if(cp->markb) {
   codebus=cp->code;
   maskbus=cp->mask;
   leafbus=cp->leaf;
   charbus=cp->charr;
  }
  break;
case EX1B:
  if(cp->markbr||cp->mask&maskbus!=maskbus)cp->toolow=1;
  if(cp->markbr) {
    unsigned short r=maskbus;
    while(r!=0xffff){r=(r<<1)+1;cp->code>>=1;
            if(!cp->mask)successbus=1;cp->mask>>=1;}
    cp->code|=codebus& maskbus;
```

```
    }
    if(cp->marka) {
     cp->charr=charbus;
     cp->leaf=leafbus;
    }
    if(!cp->markbr && !cp->markb && (cp->code|maskbus)==(codebus|maskbus) &&
                     cp->mask==(maskbus&cp->mask)) {
     unsigned short r=maskbus;
     cp->markar=1;
     #ifdef DEBUG
      printf("Marking cell #%03d for A reception, code %s\n",cn,
                      bincode(cp->code,cp->mask));
     #endif
     while(r!=0xffff){r=(r<<1)+1;cp->code<<=1;cp->mask=(cp->mask<<1)+1;}
    }
    break;
  case EX2:
    if(cp->markar) {
      unsigned short r=maskbus;
      while(r!=0xffff){r=(r<<1)+1;cp->code>>=1;
              if(!cp->mask)successbus=1;cp->mask>>=1;}
      cp->code|=codebus& maskbus;
    }
    if(cp->markb) {
     cp->charr=charbus;
     cp->leaf=leafbus;
    }
    break;
 }
}

all_cells(funcode fc)
{
 int i;
 for(i=0;i<SETSIZE*2-1;i++) do_cell(i,fc);
}

inittree()
{
 int i,j,incr;
 maskbus=0x007f;
 codebus=0xff00;
 leafbus=1;
 countbus=1;
 incr=0x0080;
 charbus=0;
 do_cell(0,INIT);
 codebus+=incr;
 charbus=1;
 do_cell(1,INIT);
 codebus+=incr;
```

```
 incr=0x0100;
 maskbus=0x00ff;
 for(i=2;i<SETSIZE;i++) {
  charbus=i;
  do_cell(i,INIT);
  codebus=codebus+incr;
 }
 j=0;
 leafbus=0;
 for(i=SETSIZE;i<2*SETSIZE-1;i++) {
  countbus=cells[j].count+cells[j+1].count;
  codebus=cells[j].code;
  maskbus=(cells[j].mask<<1)+1;
  j+=2;
  do_cell(i,INIT);
 }
}

update()
{
 unsigned short amask,acode,bmask,bcode,aleaf,achar;
 if(cells[2*SETSIZE-2].count==32766)return inittree();
 successbus=0;
 #ifdef DEBUG
  printf("Starting test cycle\n");
 #endif
 maskbus=0xffff;
 all_cells(TESTA);
 all_cells(TESTB);
 while(successbus) {
  acode=codebus;
  amask=maskbus;
  aleaf=leafbus;
  achar=charbus;
  #ifdef DEBUG
   printf("Starting exchange cycle\n");
  #endif
  all_cells(EX0);
  successbus=0;
  all_cells(EX1A);
  all_cells(EX1B);
  if(successbus)return inittree();
  bmask=maskbus;
  bcode=codebus;
  charbus=achar;
  maskbus=amask;
  codebus=acode;
  leafbus=aleaf;
  all_cells(EX2);
  if(successbus)return inittree();
  codebus=bcode;
```

```
   maskbus=0xffff;
   #ifdef DEBUG
    printf("Starting test cycle\n");
   #endif
   all_cells(TESTA);
   all_cells(TESTB);
  }
}

int do_decode()
{
 int c;
 codebus=codebuf;
 all_cells(DECODE);
 c=charbus;
 refill();
 update();
 return c;
}

do_encode(int c)
{
 charbus=c;
 all_cells(ENCODE);
 codebuf=codebus;
 sendout();
 update();
}

compress()
{
 int c;
 openfiles();
 inittree();
 while((c=getc(infile))!=EOF) {
   do_encode(c);
 }
 do_encode(256);
 flushbyte();
 fclose(infile);
 fclose(outfile);
}

decompress()
{
 int c;
 openfiles();
 inittree();
 maskbus=0;
 refill();
 while((c=do_decode())!=256) {
```

```
  putc(c,outfile);
 }
 fclose(infile);
 fclose(outfile);
}

usage()
{
 fprintf(stderr,"Usage: ahuf [c|d] infile outfile\n");
 exit(1);
}

main(int argc,char **argv)
{
 if(argc!=4)usage();
 inname=argv[2];
 outname=argv[3];
 if(strcmp(argv[1],"c")==0)compress();
 else if(strcmp(argv[1],"d")==0)decompress();
 else usage();
 exit(0);
}
```

Appendix E

Detailed Description of Device.

E.1 Data Sheet of High-Speed Data Compressor and Decompressor

E.1.1 Purpose

The VLSI device compresses and decompresses data at a rate of up to 100 MBit/s. It employs sliding window coding followed by Huffman coding, which is one of the best algorithms for the lossless compression of computer files.

Due to its speed it is very suitable for real-time compression of data that must be written to disk or tape or that is transmitted over a network.

It compresses and decompresses blocks of data from memory to memory, using the DMA facilities that are present in most modern microcomputers. It can therefore operate with minimal interaction of the CPU.

E.1.2 Interfacing

E.1.2.1 Hardware Interfacing

The device is connected to the system bus of a computer system. It can be used with a variety of bus architectures. The device reads its input data from memory and writes its output data to memory using two independent DMA channels.

The DMA channels can be 8, 16 or 32 bits wide. The device will only read or write a single block of data before it expects a continuation command from the CPU. It will send as many DMA requests on the appropriate channel as needed to read or write a whole block. It relies on the DMA controller to address memory at increasing addresses. After it has finished either reading or writing a block, it sends an interrupt request to the CPU indicating in its status register which operation was finished. The CPU then establishes another region of memory where the compression device may read or write, it reprograms the DMA controller, it puts a different value into the input block size or output block size if necessary and it sends a continuation command to the device.

The device uses 2 static RAM chips, 32k×8 bits each, for intermediate storage. This RAM has a separate interface and it is not connected to the system bus.

The device has the following external connections:

- **system data bus** bidirectional, 32 pins.
- **system address bus** input, 4 pins, to select I/O register.
- **ioread, iowrite** inputs, 2 pins.
- **chip select** input, 1 pin.
- **ready** output, 1 pin.
- **dma req 0 and 1** outputs, 2 pins.
- **dma ack 0 and 1** inputs, 2 pins.
- **interrupt** output, 1 pin.
- **RAM data** bidirectional, 16 pins.
- **RAM address** output, 15 pins.
- **RAM read, write and chip select** outputs, 3 pins.
- **RAM ready** input, 1 pin.
- **power, ground** (2 of each) 4 pins.
- **clock, reset** 2 pins.

E.1.2.2 Device Registers

The CPU in the computer communicates with the device through sixteen 8-bit I/O addresses. Table E.1 shows their functions. Some registers are read-write, the other ones are read-only.

The 'input block size', 'output block size', 'byte read' and 'bytes written' registers are 16-bit counters, divided over 2 8-bit registers. They can hold values in the range 1…65536. The value 65536 is represented by 0 in both registers. They always indicate a size in bytes.

The registers `parameter register number' and `parameter register content' provide access to a set of 16 compression parameter registers.

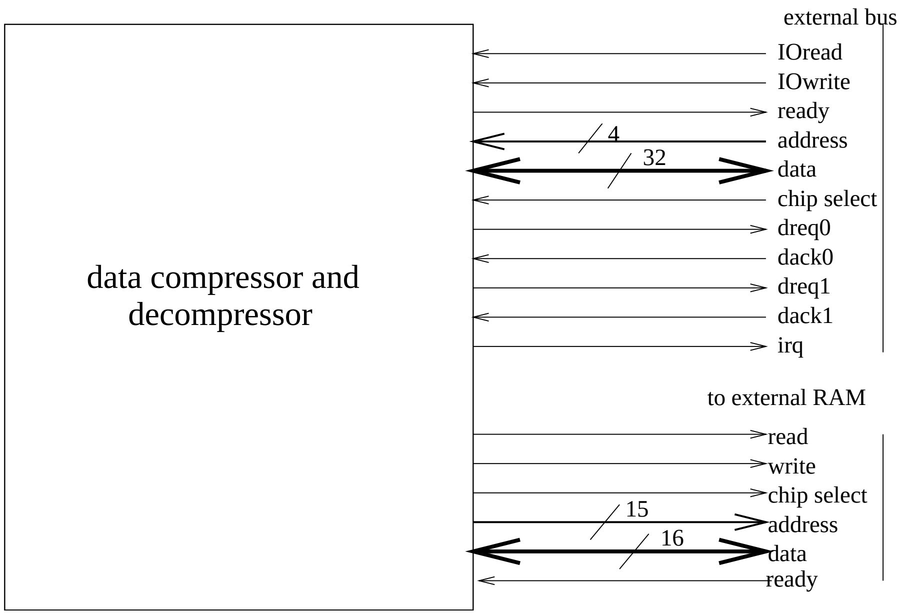

Figure E.1: External Connections of Compressor

E.1.3 Compression Algorithms

The device compresses data using sliding window compression with a window size of up to 8192 bytes and subsequent Huffman compression of the generated addresses, lengths and literals. Two Huffman codes are employed, one for the set of lengths and literals and one for addresses. Block static Huffman coding is supported. The device is capable of compressing and decompressing data in the same format as used by the `gzip` compression program.

C programs are available that can compress and decompress data in any of the formats supported by the device.

The various compression formats are selected by putting the device in programming mode and writing values to the parameter registers (see table E.6). If two registers hold one 16-bit value, then the least significant part is contained in the lowest numbered register. The device leaves the programming mode when the end-of-input command is given. It responds with an interrupt and sets bit 7 of the status register (abort) in case of a parameter error.

The data is compressed in two steps:

- A sliding window coder keeps the last N processed input bytes in a buffer. It matches substrings of incoming bytes against the buffer. If a substring matches, it is converted to the address and length of the substring. Single characters (that are not part of a matching substring) are converted to literals. The output of the sliding window coder consists of a sequence of length/address pairs and literals.

  The value of N can be chosen up to 8192. The minimum substring length is 3 and the maximum substring length may can be set to a value not higher than 258.

A3	A2	A1	A0	description	type
0	0	0	0	Command register	read-write
0	0	0	1	Interface mode	read-write
0	0	1	0	Status register	read-only
0	0	1	1	Interface error code	read-only
0	1	0	0	Input block size LO	read-write
0	1	0	1	Input block size HI	read-write
0	1	1	0	Bytes read LO	read-only
0	1	1	1	Bytes read HI	read-only
1	0	0	0	Output block size LO	read-write
1	0	0	1	Output block size HI	read-write
1	0	1	0	Bytes written LO	read-only
1	0	1	1	Bytes written HI	read-only
1	1	0	0	Parameter register number	read-write
1	1	0	1	Parameter register content	read-write.
1	1	1	0	reserved	—
1	1	1	1	Compression error code	read-only

Table E.1: I/O Registers.

Sliding window coding can be eliminated completely, in which case only Huffman coding is performed.

- A Huffman coder converts the lengths, literals and the addresses to a compressed byte stream using one or two Huffman codes. The compressed data includes descriptions of the Huffman codes as well.

  The following situations can occur:

  – No Huffman coding is selected. In this case a fixed encoding is used for lengths, literals and addresses.

  – Static Huffman coding is selected with one set of codes per file. In this case statistics are collected of the first block of the sliding window coder output (the size of this block is selectable) and a set of Huffman codes is created, which is used throughout the compression session. The description of the code is transferred once at the start of the compressed file.

  – Static Huffman coding is selected with different Huffman codes for each block. In this case each block of the sliding window coder output is collected in a buffer, statistics of the block are collected, a set of Huffman codes is collected, which is used for that block only. Descriptions of the codes are transferred along with each compressed block.

  The first method has a short time interval between the arrival of uncompressed data and the output of compressed data and it allows the use of `end-of-section'

bits	val	description
7		Abort operation.
6–5	0 0	No input-related command.
	0 1	Continue reading next input block.
	1 0	End of input section.
	1 1	End of input.
4–3	0 0	No output-related command.
	0 1	Continue writing next output block.
	1 0	Acknowledge end of output section condition.
	1 1	Reserved.
2–0	0 0 0	No further command.
	0 0 1	Reset device
	0 1 0	Run self-test
	0 1 1	Enter programming mode.
	1 0 0	Start compression.
	1 0 1	Start decompression.

Table E.2: Command Register

commands for interactive applications.

E.1.4 Compression and Decompression Sessions.

A compression session is started with the `start compression' command. It lasts until an end-of-input or abort command is given or until an error occurs. A decompression session is started with the `start decompression' command. It lasts until an end-of-input or abort command is given or until an error occurs or until the end of the compressed data is encountered. When the end-of-input command is given, the session does not end immediately. Compression or decompression is first finished, the output is written and the end-of-output status is given.

The end-of-section commands serve two purposes. Only one of them can be enabled at a time.

- **Interactive applications** In many interactive applications the data stream consists of short blocks with long time intervals in between. If the end of input section command is given at the end of each block, the compressor does not wait for more input before it generates the corresponding output (and then it indicates the end-of-output-section status). In this way the compressed data can be further processed before more input is available. Huffman coding cannot be used.

- **Multi-file sessions** When Huffman coding is active, there is a long time interval between input of the uncompressed data and output of the compressed data. For long sessions this has little impact on throughput. Therefore it is possible to compress more than one file per session. The end-of-input-section command indicates

bits	val	description
7		Operation is aborted.
6–5	0 0	Input in progress.
	0 1	Waiting for next input block.
	1 0	Input section is ended.
	1 1	Input is ended.
4–3	0 0	Output in progress.
	0 1	Waiting for next output block.
	1 0	Output section is ended.
	1 1	Output is ended.
2–0	0 0 0	Ready to accept new command.
	0 0 1	Initialization sequence in progress.
	0 1 0	Self-test in progress.
	0 1 1	Core programming in progress.
	1 0 0	Compression in progress.
	1 0 1	Decompression in progress.

Table E.3: Status Register

bits	val	description
7		Generate IRQ signal when appropriate.
6–5	0 0	8 bit DMA.
	0 1	16-bit DMA.
	1 0	32-bit DMA.
4–1		Reserved, set to 0.
0	0	Continue same output block after end of output section.
	1	Start new output block after end of output section.

Table E.4: Interface Mode

the end of a file. When the corresponding compressed data is written, the end of output section status is indicated. Each file is decompressible independently of other files. Compressed files can be automatically padded out to a certain multiple of bytes to facilitate their storage on disk sectors.

E.1.5 Performance

The device is able to process data at a sustained rate of 100 megabits per second. This is 12.5 million 8-bit bytes per second. This is the rate of the uncompressed data, either when decompressing or when compressing. The clock frequency is 50MHz.

The achieved compression ratio depends on the type of data being compressed. For a large set of files (texts and binaries) the average ratio is 2.9 bits/byte.

bits	val	description
7		Reserved.
6		DMA request timed out.
5		Aborted by Abort command.
4		Internal error in interface.
3		Inappropriate command given.
2		Error detected in compressed data.
1–0		Reserved.

Table E.5: Interface Error Code

reg	bits	description
0–1		sliding window size, maximum 8192 bytes.
2–3		maximum substring length, maximum 258 bytes. A value of 0 indicates bypass sliding window compression.
4	7	All bytes in window are set to zero before compression. Matching against those zero bytes is permitted.
5–6		Number of address/length pairs and literals per block, maximum 8192
7	7	Use Huffman code for lengths and literals.
	6	Use Huffman code for addresses.
	5	Enable `end-of-section' command for interactive applications.
	4	0 Use one set of Huffman codes per file, 1 use one set of Huffman codes per block.
	3	Enable multi-file sessions.
8–9		Padding count of compressed data. Should be a power of 2, minimum 1, maximum 65536.

Table E.6: Functions of Parameter Registers.

E.2 Hardware Interfacing.

This section describes the interfaces used in the data compression and decompression device. The external bus connects the device to the computer system it is built in. The bus interface module is the only module that accesses it. The data channels transfer the data from the bus interface through the various compressing modules back to the bus interface. Those channels carry data items at a very high speed. The data items are uncompressed bytes, partially compressed data consisting of lengths, literals and addresses and finally compressed bytes. These channels have flow control capabilities so they provide a way of synchronization. All modules are connected to the internal bus, of which the embedded processor is the only bus master. All modules can send interrupt signals to the processor, which can then read out their status registers and set their control registers. This provides another important mechanism of synchronization. The internal bus also carries all the traffic between the processor and the RAMs it is connected to. Swappable RAMs are a way to exchange large quantities of data at once between the processor and other modules. The code buffer needs so much storage that it cannot be located on the chip itself. Therefore it interfaces to an external RAM.

Figure E.2 shows the architecture diagram with the modules, the channels and the RAMs connected to the internal bus.

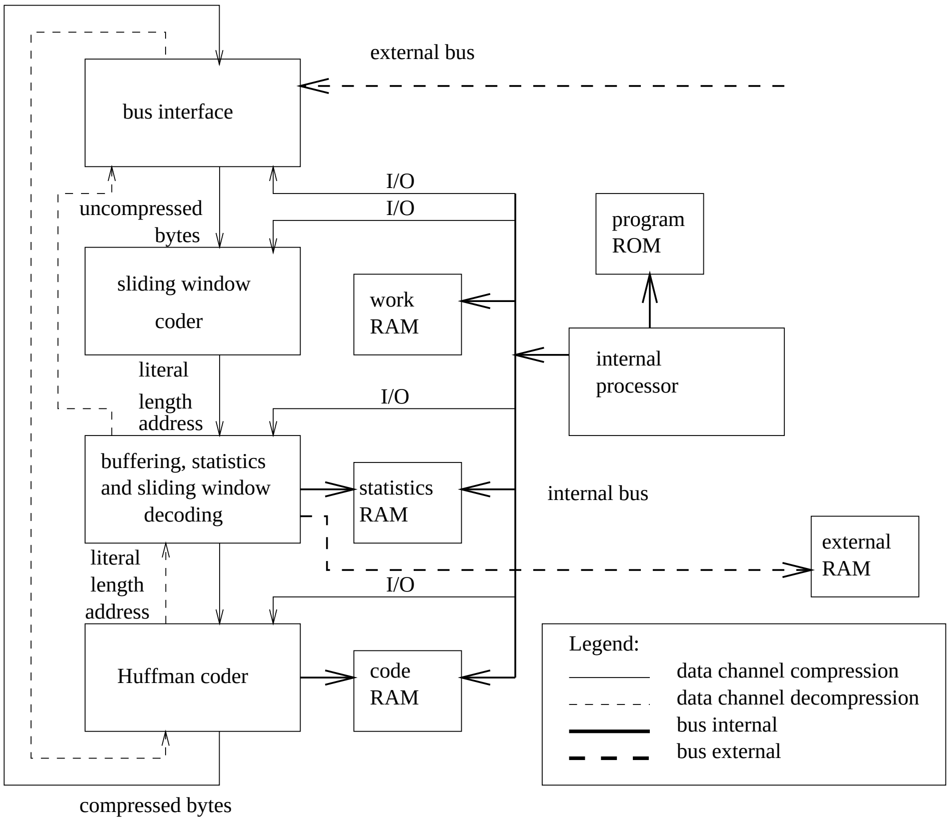

Figure E.2: Architecture Diagram with Internal Bus.

Only the external bus and the external RAM connection are visible outside the chip. All other buses and channels are strictly internal.

E.2.1 External Computer Bus

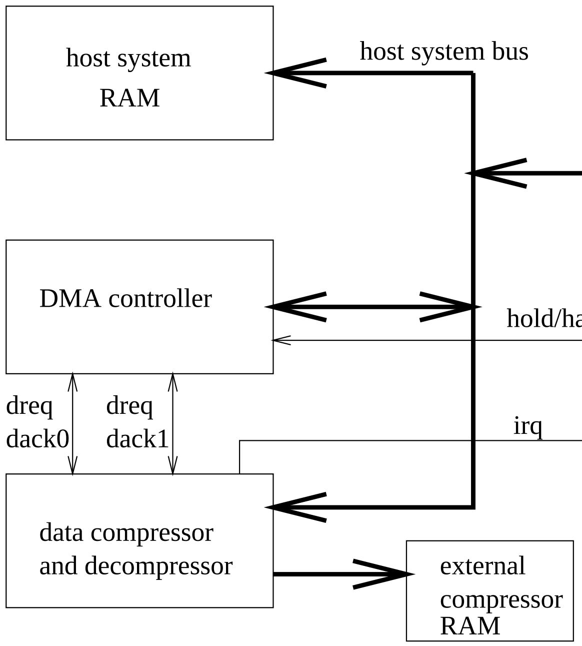

Figure E.3: The Data Compression and Decompression Device in the System.

The external bus connects the compression device to the system bus of the computer system that the device is built in. Figure E.3 shows that system. The device is used as a peripheral. The CPU sends commands to the device and reads its status through I/O registers. The device uses the DMA controller to read data from the RAM and to write the processed data back to RAM. Whenever reading or writing of a block of data is finished, the CPU must reprogram the DMA controller. If an error occurs or if all operations are finished, the CPU must be notified too. The device sends an interrupt whenever the attention of the CPU is needed. Signals of the external bus are are inputs to the device, unless indicated othterwise by 'rev'.

First there is the standard external bus connection. It carries the following signals:

- **data bus** (32 bits, bidirectional) I/O register reads and writes run over this bus as well as DMA transfers. The bus is used when the chip select line is low or when a DMA transfer is in progress.
- **address bus** (4 bits) These lines from the system address bus determine which I/O register is read or written.

- **chip select** An active-low signal that indicates that an I/O register is read or written. This should only be active when the system indicates that it performs an I/O read or write and if the address bus carries an address in the range assigned to the chip. This input should be controlled by an address decoder. One of the signals 'io read' and 'io write' must be simultaneously active for the I/O transfer to occur.

- **io read** This active-low signal is active whenever an I/O register is read by the CPU. It is also active whenever a memory writing DMA transfer occurs. Data are put on the external data bus during the time this signal is active.

- **io write** This active-low signal is active whenever an I/O register is written by the CPU. It is also active whenever a memory reading DMA transfer occurs. The data is being latched synchronously to the clock until 'io write' goes inactive. The last data latched is the data that will be used.

- **ready** (rev, open collector) This active-low signal acknowledges the 'io read' and 'io write' signals, but only if they have to do with this device, either DMA actions or register I/O. The signal goes inactive as soon as the 'io read' or 'io write' goes inactive.

The data bus contains bits numbered from 0 to 31. The data stream is a logical stream of bytes. The first byte is in bits $0 \ldots 7$ and the last byte is in bits $0 \ldots 31$. IF the DMA transfer width is 16 bits, only bits $0 \ldots 15$ are used and if the DMA transfer width is 8 bits, only bits $0 \ldots 7$ are used. I/O register reads and writes are always on bits $0 \ldots 7$ of the data bus. If the chip is connected to a system that uses a big-endian byte order, it is best to wire the data bus connections byte-swapped to the system data bus. If only 8 or 16-bit DMA transfers are possible in the system, the unused lines may be left unconnected.

Next there are the DMA lines. The following signals are present:

- **dma req 0** DMA request signal for DMA channel 0.

- **dma ack 0** (rev) active-low DMA acknowledge signal for DMA channel 0.

- **dma req 1** DMA request signal for DMA channel 1.

- **dma ack 1** (rev) active-low DMA acknowledge signal for DMA channel 1.

DMA channel 0 is used for memory reading and DMA channel 1 is used for memory writing.

Whenever the bus interface is ready to perform a DMA transfer, it sets the appropriate DREQ signal active and waits until the DMA controller sets the appropriate DACK signal active. If it is a DMA transfer to memory, 'io read' must also be active. The bus interface will put the data on the data bus during the time that the this signal is active. If it is a transfer from memory, 'io write' must be active. The bus interface will read the contents of the external data bus and latch them when this signal goes high (inactive) again. The bus interface drops DREQ as soon as it receives DACK. It will make the 'ready' signal active as with ordinary I/O reads and writes.

Finally there is the external interrupt line. It carries the following signal:

- **int req** This is the interrupt request signal, which goes active each time the attention of the CPU is needed. The signal goes inactive again as soon as the CPU reads the status register. The polarity of this signal can be selected by a bit in a device register.

E.2.2 Internal Processor Bus

The internal processor bus connects the internal processor to the other modules and its working RAM. The other modules are accessed as I/O devices by the processor.

The bus carries the following signals:

- **data out** This is a 16 bit wide bus driven by the processor. It carries all data written to the RAM and the I/O devices.
- **data in** This is a 16 bit wide bus driven by other devices. It carries all data read from the RAM and the I/O devices.
- **address** This is a 12 bit wide bus driven by the processor. It carries the address of the RAM location or the I/O device that is accessed.
- **read** This is one line driven by the processor. If set high, the RAM (or the addressed I/O port) must place a value on the `data in` bus at the next clock edge.
- **write** This is one line driven by the processor. If set high, the RAM (or the addressed I/O port) must accept the data that appears simultaneously on the `data out` bus.
- **interrupt** This is an input line for the processor. If it is high at the active clock edge, the processor interrupts its normal program flow.

The ROM is part of the processor itself and the connections to it are not described here.

The internal interface is similar to a traditional processor bus, except that it has separate data lines for reading and writing and that it is a synchronous design in which all bus cycles take exactly one clock cycle. If an address is placed on the address bus along with the read signal, the addressed data (read from RAM) appears on the `data in` bus during the next clock cycle. If an address is placed on the address bus along with the write signal, the data to be written appears simultaneously on the `data out` bus and it must be accepted by the next clock edge.

We chose separate data buses for input and output instead of a bidirectional bus because on-chip tristate logic is relatively costly and because on-chips RAMs have separate data buses as well. The on-chip RAMs of most silicon compiler vendors work exactly as the internal processor bus does.

An address decoding circuit will decode the address into the signals necessary to select the various pieces of RAM and I/O devices. This decoder is primarily a combinatorial circuit. In the designed system a 4 kiloword address range is expected, divided into the following regions:

- 2 kilowords of RAM that is only accessible by this processor.
- 896 words of RAM that is swappable between the processor and the Huffman encoder/decoder (see section E.2.3).
- 320 words of RAM that is swappable between the processor and the statistics collecting module (see section E.2.3).
- A 16 byte 'parameter RAM', which is shared between the processor and the bus interface of the device. It is located inside the bus interface module (and therefore not shown in the main architecture diagram).
- Various control, status and other ports of the other modules. These are the I/O addresses, preferably mapped to one consecutive range. It is expected that less than 64 addresses are needed.

The logical 'or' of all interrupt sources is connected to the interrupt input of the processor. The processor will only honor interrupts when not already handling an interrupt. The interrupt source will clear its interrupt signal as soon as the relevant input port is read by the internal processor.

E.2.3 Swappable RAMs

There are two swappable RAMs. Each of them consists of two identical banks. These RAMs are:

- 2 banks of statistics RAM (each bank 320 words of 13 bits), shared between the buffer/statistics module and the internal processor.
- 2 banks of code RAM (each bank 896 words of 16 bits), shared between the Huffman coder and the internal processor.

One of the banks is connected to the internal processor bus and the other bank is connected to the other module. The interface to the other module is similar to the external processor bus, but it is a separate bus that contains an address different from the address on the internal processor bus. The data and address bus width on the interface to the other module may also differ.

At any given time the internal processor can access one of the banks in a swappable RAM. The processor can read the contents that were written there by the other module (before the banks were swapped) or it can set up the data in the RAM for use by the other module (after the next swap). The other module has exclusive access to a separate

RAM bank of the same size. The connections to the two banks can be swapped between the internal processor and the other module. The bank that was accessible to the other module, becomes accessible to the internal processor and vice versa.

Swapping is controlled by a single I/O port writable by the internal processor.

E.2.4 External RAM

The external RAM interface connects the buffer/statistics module to an external RAM. This external RAM is privately used by this module as a buffer memory.

The external RAM interface carries the following signals:

- Data (16 bits bidirectional)
- Address (15 bits)
- Read (active-low)
- Write (active-low)
- Chip Select (active-low) This is active when either Read or Write is active.
- Ready (input)[1]

This is an off-chip connection that can directly drive two $32k \times 8$ static (cache) RAM chips.

E.2.5 Data Channels

The data channels carry data items that are processed at a very high rate. Each of the data channels runs between two modules and is unidirectional. In most cases there are pairs of unidirectional channels, one for each direction. A channel is only used either when compressing or when decompressing. If a channel is not in use, the sending module makes the 'data available' signal 0 and the receiving module makes the 'ready' signal 0, to prevent spurious transfers. The 'end-of-section' and 'end-of-input' signals are also propagated over these channels.

A synchronous protocol is used in which one data item can be sent per clock cycle. The receiving end indicates that it is ready to receive and the transmitting end indicates that it has data available. If both these signals are high, then the data is latched into a register at the receiving end at the next active edge of the clock. If the receiver is not ready to receive more data, it sets the ready signal low and the transmitter must keep the same data available until the receiver is ready to consume it. Flow control is implemented this way and the receiver can wait an indefinite time before it sets its

[1] For a single static RAM device it would be unnecessary to use a ready line. It would be more convenient to use an adjustable timer.

ready signal high. To prevent deadlock, the transmitting module must continue all its tasks that do not depend on the completion of that transmission. For timing details see figure E.4.

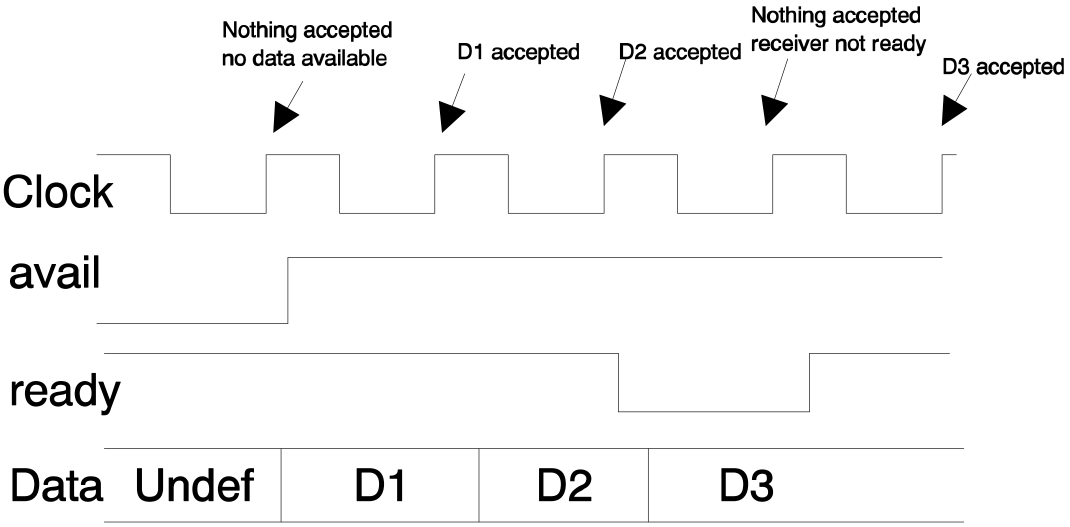

Figure E.4: Timing of One-Phase Channel.

Figure E.5 shows a typical implementation of such a channel. The sending module contains a flipflop for 'data available' and a register for the data to be sent. The receiving module contains a flipflop for 'ready for data'. If both the handshake signals are high, the receiver receives the data at the next active clock edge, typically by latching it into a register of its own. The receiver must either reset the flipflop for the ready signal or it must make sure it can receive more data at the next clock edge. The transmitter must either reset the data available signal or it must latch new data into its register for the next clock cycle. For this scheme to work with a maximum throughput of one data item per clock cycle, a module must have at least one buffer in which it can queue a data item that it has promised to receive but that it can not readily consume. This buffer is otherwise bypassed.

The byte channels carry uncompressed or compressed bytes. They contain the following signals:

- **data** (8 bits) This is the data to be processed by the receiver.
- **avail** This indicates that the data is valid.
- **ready** (in reverse direction) This indicates that the receiver is ready to receive data or the 'end of section' or 'end of data' events.
- **end of section** This indicates that a section of the data is ended. Such a section may be a command entered by a user or a data block generated by an application. It may take a long time for the next section to arrive and hence complete processing of the section is desired before the next section arrives.

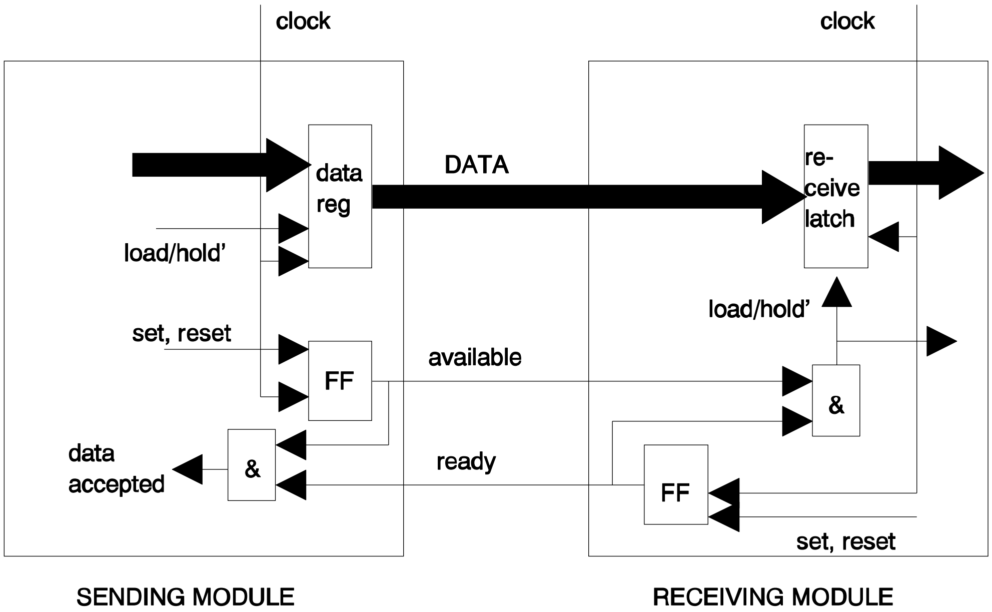

Figure E.5: Implementation of One-Phase Channel.

- **end of data** This indicates the end of the data stream.

At most one of the avail, 'end of section' or 'end of data' signals may be active at any time. Only when the avail signal is high, the data signals are valid. If the receiver cannot accept more data, it will set the ready signal low as long as the data cannot be accepted.

The length, literal and address channels carry partially compressed data as produced by the sliding window coder. They also carry the end-of-section and end signals. Their protocol is very similar to that of the byte channels. They carry the following signals:

- data (12–16 bits, depending on the maximum address size) Normally a length literal item is carried in the 10 least significant bits of data. Values below 256 denote literals, values above 257 denote lengths. The values 256 and 257 denote end-of-data and end-of-section respectively. When a length is transferred, the next item will be an address.
- available
- ready (in reverse direction)

E.3 Module Specifications.

E.3.1 Bus Interface

The bus interface has the following tasks:

- Communicating with the external CPU through command and status registers.

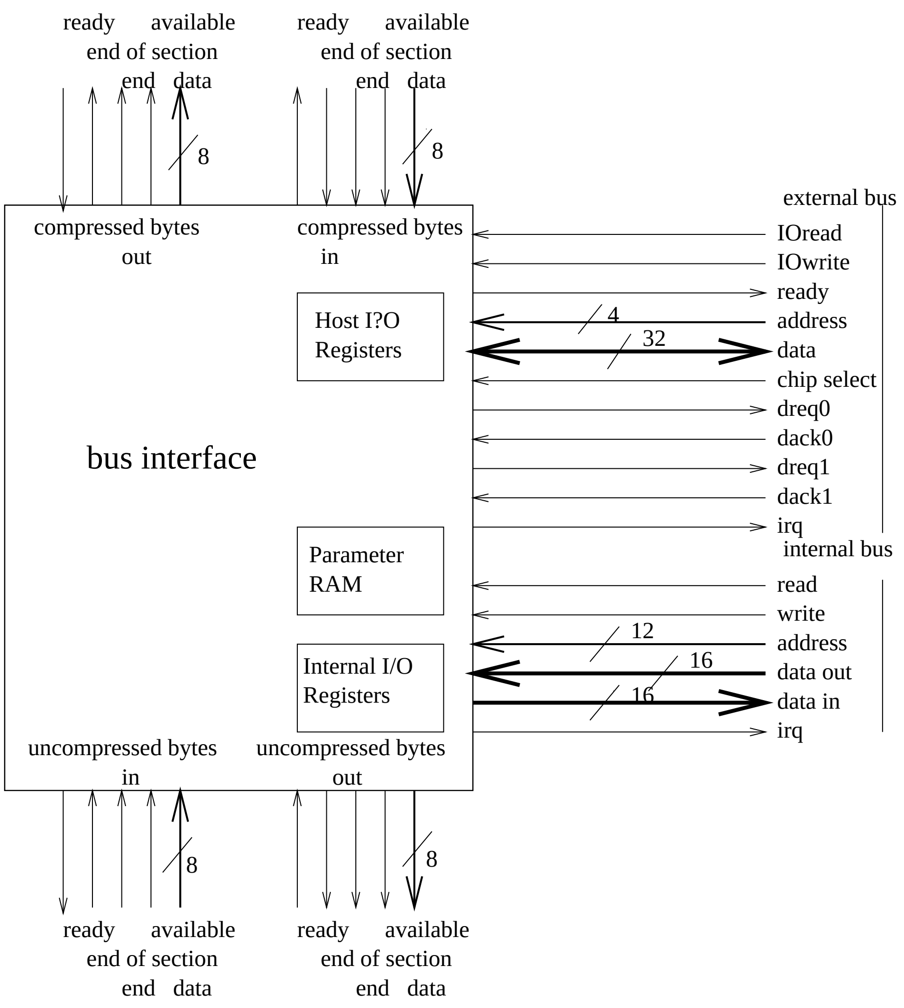

Figure E.6: Diagram of Bus Interface

- Generating external Interrupts.
- Data transfer between core and system RAM through DMA

These tasks are performed autonomously, independent of the internal processor. Unlike the other modules, the bus interface is not programmed or initialized by the internal processor. The whole communications protocol with the host system, involving DMA, interrupts, byte counters, command and status registers is invisible to the core and the internal processor.

The bus interface has the following channels:

- The external bus, that is described in a previous section.
- The compressed bytes channels, two byte channels in either direction. These are connected to the Huffman coding module.
- The uncompressed bytes channels, two byte channels in either direction. One runs from the bus interface to the sliding window coding module, the other one runs from the buffering and statistics module (sliding window decoder) to the bus interface.

The bus interface contains the following storage items:

- 16 8-bit I/O registers, accessible by the host system as I/O ports, but not by the internal processor. These are various command and status register and byte count registers.
- 16 8-bit parameter registers, indirectly accessible by the host system through 'parameter register number' and 'parameter register content' registers. When the 'set and check compression parameters' command is active, these registers are accessible as RAM on the internal processor bus.

The following I/O ports are made available on the internal processor bus:

- **Core Status** (output port, 8 bits). This is the status code that the internal processor sends to the bus interface when it aborts an operation due to an error. This status code is made available to the host system through one of the I/O registers of the bus interface.
- **Bus Status** (input port, 8 bits). Clears internal interrupt line when read. This port indicates to the internal processor the following things:
  - whether a command is available.
  - the number of the command, if available.
  - whether the command was aborted by the host system or the bus interface.

- **Bus Control** (output port, 8 bits). This port indicates to the bus interface the following things:
  - Whether the internal processor aborts the operation.
  - Whether the internal processor ends the last command.
  - The direction of the 'compressed bytes' and 'uncompressed bytes' channels.
  - Whether the parameter register are accessible on the internal bus or by the bus interface.

When the host system writes any of the 'reset', 'self test', 'compress' or 'decompress' commands to the command register, the bus interface issues an internal interrupt and makes the command available on the 'bus status' register. When the host system aborts a command in progress or when the bus interface module encounters an error, it issues an interrupt and indicates through the bus status register that there is an abort. If the internal processor reads the bus status register, the interrupt is cleared.

When the host system writes the 'enter programming mode' command to the command register, the bus interface lets the host system set the 16 parameter registers. When the programming mode is left, the bus interface issues an internal interrupt and makes the 'set and check compression parameters' command available on the 'bus status' register. As long as this command is active, the internal processor may access the 16 parameter registers.

There are three ways in which a compress or decompress command can be terminated:

- The bus interface issues an abort. It interrupts the processor that stops all other modules and indicates through the bus control port that it terminates the last command.
- The end of the data appears. The sliding window coder, the Huffman coder or the buffering and statistics module detect this condition and they interrupt the processor. The processor then indicates through the bus control port that it terminates the last command.
- One of the modules (including the internal processor) detects an error. The internal processor stops all other modules and then it indicates through the bus control port that it aborts the operation and that it terminates the last command.

The other commands are terminated by the internal processor through the 'bus control' port.

E.3.2 Embedded Processor

The embedded processor implements all functions of overall control, the tree generation and Huffman tree transmission and reception. These functions are:

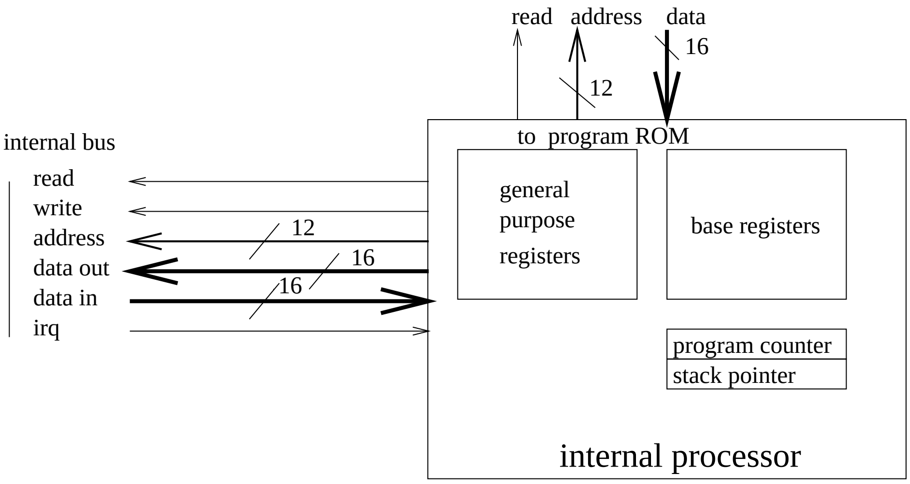

Figure E.7: Diagram of Processor

- Initialization and control of all other modules except bus interface.
- Starting and terminating the commands as given by the bus interface.
- Computing optimal Huffman codes from statistics (statistics obtained through a swappable RAM).
- Programming codes into the Huffman coding Module (through a swappable RAM).
- Encoding and decoding of static Huffman code descriptions that are sent along with the compressed data.

The internal processor is the only bus master on the internal processor bus. The program is contained in a ROM, not connected to that bus. The ROM contains 4096 16-bit instructions. 2048 16-bit words of RAM are the working storage of the internal processor and they are connected to the internal bus. Also connected to the internal bus are one bank of the swappable statistics and code RAMs.

The processor is optimized for random access to arrays, which occurs frequently in Huffman related algorithms. It has 8 16-bit general purpose registers (also used as index) and 16 array base registers. All operations are 16 bit.

E.3.3 Sliding Window Coder

The sliding window coding module compresses the uncompressed byte stream into literals and substring descriptors, which consist of lengths and addresses. When decompressing it is unused.

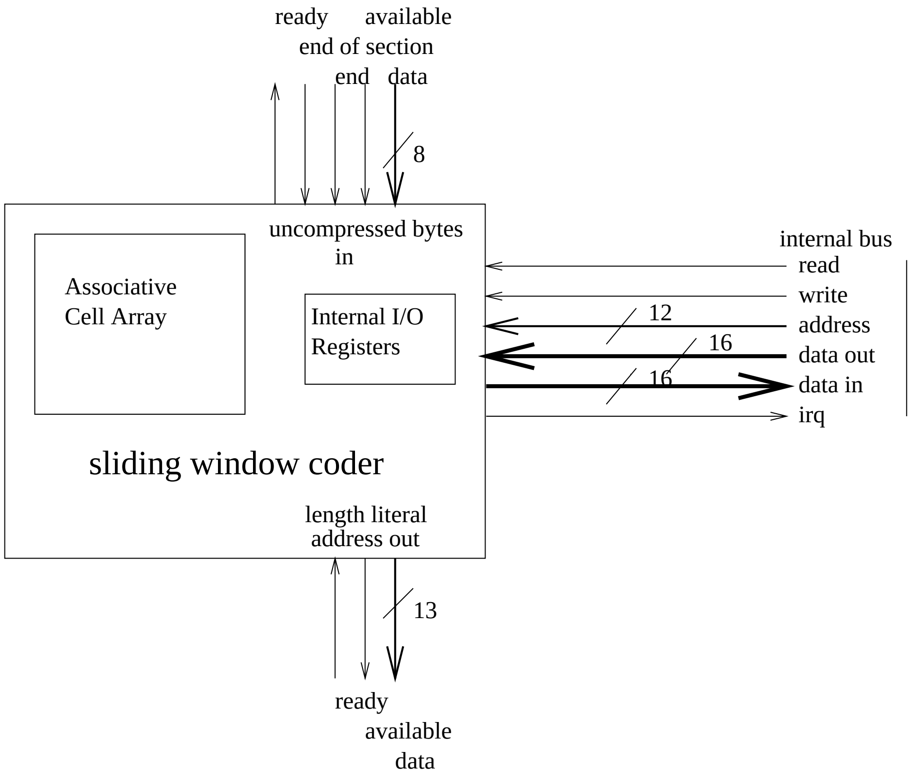

Figure E.8: Diagram of Sliding Window Coder

The module is connected to two channels:

- The 'uncompressed bytes' channel, a byte channel from the bus interface.
- A length, literal and address channel to the buffer/statistics module.

The module has 8192 8-bit bytes of storage for the sliding window plus 2 extra flag bits for each byte in the window. This is 8192×10 bits together.

The following I/O ports are available on the internal bus:

- **Window Command** (8 bit output port), The internal processor can give the following commands:
  - start compressing.
  - abort operation.
  - continue operation.
- **Window Status** (8 bit input port), indicates to the internal processor the following things:
  - whether end-of-section occurred.
  - whether end-of-data occurred.

Reading this port clears the interrupt signal.

- **Window parameters** (8 bit output port) The internal processor can set the following parameter:
  - whether matching against uninitialized zero part is allowed.
- **Window Length** (9 bit output port) to set maximum string length (1 ... 258).
- **Window Size** (14 bit output port), to set window size (1 ... 8192).

Whenever the module processes the end-of-section or end-of-data marker that is present on the data channels, it stops all operations and sends an interrupt to the internal processor. The processor is supposed to give the 'continue operation' command on the 'window command' port when end-of-section was encountered.

E.3.4 Buffering, Statistics and Sliding Window Decoding Module

The buffering, statistics and sliding window decoding module has the following functions:

- buffering blocks of lengths, literals and addresses for static Huffman coding.
- collecting frequency statistics.
- providing simple buffering when fixed coding is used.
- performing sliding window decompression.

The module has the following channels:

- An uncompressed byte channel to the bus interface.
- A length, literal and address channel that is connected from the sliding window coder to this module. Its address width is 13 bits (12 for 4kB sliding window).
- A pair of length, literal and address channels in either direction that are connected to the Huffman coder. The channel from the Huffman coder has an address width of 16 bits (15 if maximum window size is 32kB). The other one has an address width of 13 (12) bits.
- An interface to one of the banks of the statistics RAM.
- An interface to the external buffer RAM.

The module has access to one of the banks of 320 13-bit words of the statistics RAM. Further it has access to the 32k$\times$16 external RAM. During compression it can maintain at least 3 buffers, the first being read, the second being in storage and the third being written.

The module has the following ports on the internal bus:

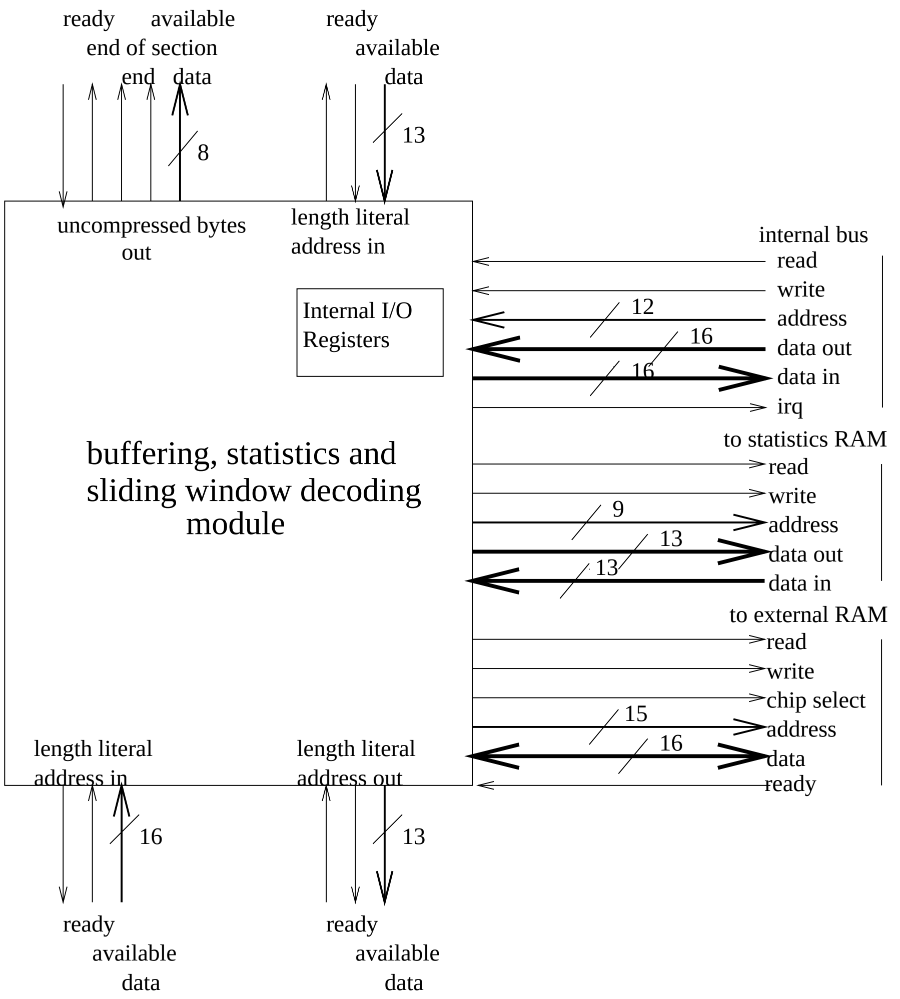

Figure E.9: Diagram of Buffering and Statistics Module.

- **Buffer Command** (8 bit output port) The internal processor can give the following commands:
  - start compression.
  - start decompression.
  - stream compression (act as a simple buffer while compressing).
  - abort.
  - continue reading (new block from sliding window coder) or continue decompressing.
  - continue writing (new block to Huffman coder).
- **Buffer status** (8 bit input port) It indicates the following events:
  - block completely read into buffer.
  - block all written from buffer.
  - block last read has end-of-data.
  - block last read has end-of-section.

  Reading this register clears the internal interrupt signal.
- **Buffer size** (14 bit output port) maximum size of a block in buffer (number of lengths or literals, 1. . . 8192).
- **Read size** (14 bit input port) size of block last read (number of lengths or literals, 1. . . 8192).

When reading a block from the sliding window coder, statistics are updated for each item read. For each length or literal, one of 288 statistics RAM locations is incremented. The length class is determined if the item is a length. Whenever a length is read, the class code of the address is determined and one of 32 locations corresponding to the class code is also incremented. When the block is all read, the processor swaps the statistics RAM for a bank that contains all zeros and fresh statistics can be computed for the next block.

Whenever a block is all read or written, the module generates an interrupt and stops the reading or writing operation until the corresponding continue command is given. A block is completely read in the following cases:

- If it contains as much lengths or literals as indicated by the buffer size.
- If the buffer memory is full.
- If an end-of-section or end-of-data marker is found on the data channel from the sliding window coder.

The internal processor finds the size of the block last read in the 'read size' port.

When in stream compression mode no interrupts are given. End-of-section or end-of-data are simply passed along. The module can be switched to stream compression mode from the standard block compression mode. In this case all data buffered thus far is written first and no data is lost.

When in decompression mode, the module generates an interrupt whenever the end-of-section or end-of-data marker is encountered. It stops all operation and waits until the 'continue reading' command is given,

E.3.5 Huffman Coder

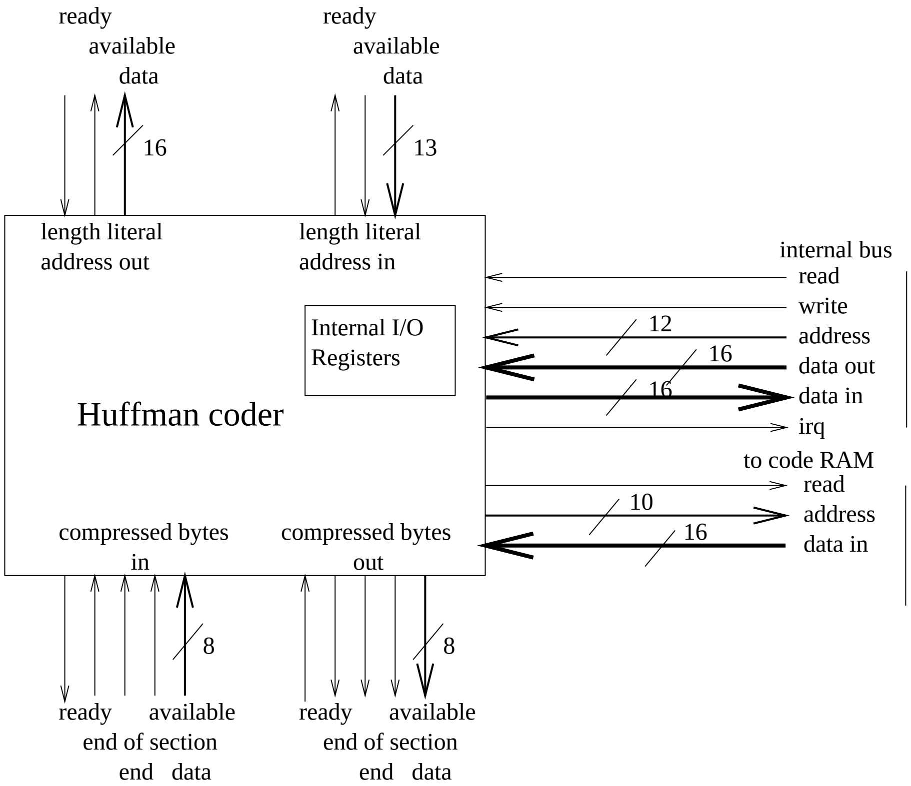

Figure E.10: Diagram of Huffman Coder

The Huffman coding module has the following functions:

- Static Huffman coding and decoding of lengths, literals and addresses.
- Converting variable length codes from and to bytes.
- Letting the internal processor send and receive variable length codes.

The module has the following channels:

- The 'compressed bytes' channels, a pair of byte channels in either direction to the bus interface.
- A pair of length, literal and address channels in either direction that are connected to the statistics and buffering module. The channel that runs from the other module to the Huffman coder has an address width of 13 (or 12) bits, the other one has an address width of 16 (or 15) bits.
- An interface to one of the banks of the code RAM. This interface is read-only. No 'write' or 'data out' signals are necessary.

The module has access to 896 16-bit words of coding RAM which are used as follows when compressing:

- 288 words indicate the code words for each length or literal.
- 288 words indicate the code word lengths for each length or literal.
- 32 words indicate the code words for each address class.
- 32 words indicate the code word lengths for each address class.

When decompressing the code RAM is used as follows:

- 512 words indicate the symbol (9 bits) and the code word length (4 bits) for all length or literal symbols with a code word less than 10 bits, addressed by codeword.
- 288 words indicate the symbols ordered by code word for all length/literal symbols with a code word longer than 9 bits.
- 6 words indicate the number of length or literal code words with a code word length of 10 through 15.
- 32 words indicate the symbol (5 bits) and the code word length (3 bits) for all address class symbols with a code word less than 7 bits. Each word contains two such entries.
- 32 words indicate the symbols ordered by code word for all address class symbols with a code word longer than 6 bits.
- 9 words indicate the number of address class code words with a code word length of 7 through 15.

Encoding is performed straightforwardly, decoding is performed using an algorithm that processes one code word at a time for code words less than a given number of bit and proceeds one bit a a time for longer code words. The algorithm uses a special ordering property of the Huffman trees used by the compressor. This algorithm is described in more detail in section A.2.4 (Huffman decoding). This algorithm should also be used for decoding the 'fixed' length/literal code.

The following I/O ports are present on the internal bus. The first three ports enable the internal processor to directly send and receive bit fields through the byte converter.

- **Bitfield Port** (16 bit bidirectional port), to send and receive variable length bit fields.
- **Bitfield Length** (4 bit output port), indicates the length of a bit field. Writing to it effects the transaction.
- **Bitfield Status** (8 bit input port, indicates whether byte converter is ready for next bit field transfer, or whether an end-of-section or end-of-data marker was present on the 'compressed bytes' channel.
- **Huf Control** (8 bit output port). The internal processor can give the following commands through this port:
  - start Huffman coding.
  - start Huffman decoding.
  - abort.
  - continue with next block.
  - continue until end of data.
  - realign bit field (next byte conversion is started on a byte boundary).
  - insert end-of-section marker.
  - insert end-of-data marker.
- **Huf Status** (8 bit input port), indicates the following events. Reading this register clears the interrupt signal.
  - end-of-data processed.
  - end-of-section code processed.
  - block completely processed.
- **Huf Length** (16 bit output port), indicates number of length/literal pairs to process in this block.
- **Bytes processed** (16 bit input port), indicates number of compressed bytes read or written. This port can be used to by the internal processor to pad the compressed data stream to a certain number of bytes.

Whenever a block containing 'Huf Length' lengths or literals is compressed or whenever the end-of-section or end-of-data marker is read, or whenever the end-of-section or end-of-block marker is decoded in the compressed data, an interrupt is generated and the operation is stopped until the internal processor sends the continue command.

E.4 Memory/IO Map of Internal Bus.

000h–7FFh RAM available to internal processor. Words are 16 bits wide.

800h–940h Swappable statistics RAM banks. One of the two banks is mapped into the processor address space, the other one is accessible by the statistics module. Words are 13 bits wide.

- **800h-91Fh** Counts for lengths and literals (0-287).
- **920h-93Fh** Counts for the address classes (0-31).

A80h–ABFh I/O ports.

- **A80h–A87h** reserved for bus interface.
  - **A80h** output, bus control port.
    - **bit 0** , parameter RAM addressability, 0 is by bus interface, 1 is by processor.
    - **bit 1** , byte channel directions, 0 is for compression, 1 is for decompression.
    - **bit 2** , 1 if the command-in-progress signal is to be reset (core terminates command).
    - **bit 3** , 1 if an abort-output signal must be given to the bus interface (core aborts command).
  - **A81h** output, core status port. 8 bits wide. Arbitrary 8-bit code sent by core when it detects an error.
  - **A82h** input, bus status port.
    - **bit 0–2** , indicates command given to core, 001 is reset, 010 is self test, 011 is set and check compression parameters, 1xx is any of four data transformations.
    - **bit 3** , 1 if command in progress.
    - **bit 4** , 1 if bus interface gives abort input signal.
- **A88h–A8Fh** reserved for sliding window coder and decoder.
  - **A88h** output, sliding window command port.
  - **A89h** input, sliding window status port.
  - **A8Ah** output, sliding window parameter port.
  - **A8Bh** output, sliding window length (1–258).
  - **A8Ch** output, sliding window size (1–8192).
- **A90h–A97h** reserved for buffering and statistics module.
  - **A90h** output, bufstat command port, 3 bits wide.
    - **001** Start compression (in block mode).

010 Start decompression (in stream mode).

011 Abort operation.

100 Continue reading next block.

101 Continue writing next block.

110 Go to stream mode.

A91h input, bufstat status port, 3 bits wide.

bit 0 last block read ended with 'end-of-data'.

bit 1 last block read ended with 'end-of-section'.

bit 2 block was read, waiting for continue command. (there is no status bit for 'block written', the Huffman coder takes care of that).

A92h output, block size port, 14 bits wide. This is the maximum size (in lengths plus literals) of a block that the module may read (1–8912).

A93h input, read size port, 14 bits wide. This is the size (in lengths plus literals) of the last block actually read.

A98h–A99h reserved for swappable RAM switching.

A98h output, Swappable RAM control port.

bit 0 controls statistics RAM bank mapping.

bit 1 controls code RAM bank mapping.

AA0h–AA7h reserved for Huffman coder and decoder.

AA0h bidirectional, bitfield port.

AA1h output, bitfield length port.

AA2h input, bitfield status port.

AA3h output, Huffman control port.

AA4h input, Huffman status port.

AA5h output, Huffman length (1–8192).

AA6h input, bytes processed.

AC0h–ACFh Parameter RAM, also accessible by bus interface module. Words are 8 bits wide.

C00h–F7Fh Swappable Huffman code RAM banks. One of the two banks is mapped into the processor address space, the other one is accessible by the Huffman coding and decoding module. Words are 16 bits wide.

When compressing:

C00h–D1Fh Code words (16 bit pattern) for each length class and literal (0..287).

D20h–D3Fh Code words (16 bit pattern) for each address class (0..31).

E00h–F1Fh Code word lengths for each length class and literal (0..287).

F20h–F3Fh Code word lengths for each address class (0..31).

When decompressing:

C00h-DFFh Items addressed by the first 9 length/literal code word bits.

- **bit 0–3** code word length, from 1 to 9, or value greater than 9 if the code word is longer than 9 bits. The value stored is 1 less than the actual length.
- **bit 4–16** symbol if the code word is shorter than 10 bits. For longer code words it is the first code word longer than 9 bits.

E00h-F1Fh Length/literal symbols ordered by code words for those symbols with code words longer than 9 bits.

F20h–F3Fh 64 8-bit items, stored 2 in a word, addressed by the first 6 bits of the address class code word. First item bits 0–7, second item bits 8–15

If the code word is shorter than 7 bits:

- **bit 0** 0
- **bit 1–3** code word length.
- **bit 4–7** symbol.

If code the word is longer than 6 bits:

- **bit 0** 1
- **bit 1–5** first code word with length greater than 6.

F40h-F5Fh Address class symbols ordered by code words for those symbols with a code word longer than 6 bits.

F60h-F68h Number of address class code words with a given length (7 through 15).

F70h-F75h Number of length or literal code words with a given length (10 through 15).

E.5 Instruction Set of the Embedded Processor.

E.5.1 Registers

The processor has the following registers:

- **index base registers**. There are up to 16 index base registers. They are added to an index register to obtain the address in indexed addressing mode. These registers are typically set to the start address of an array and remain constant during the execution of an algorithm. The index registers are used as the array index and vary far more frequently.

- **direct base registers**. There are 2 direct base registers. They are added to a 6-bit direct address to obtain the address in direct addressing mode. These registers are typically set to a memory area for various variables and the I/O address area respectively. The direct addressing mode is then used to address either one of 64 variables or one of 64 I/O ports.

- **general purpose registers**. There are 8 general purpose registers. Arithmetic and logic operations always have a general purpose register as the destination. One of the source operands is the same register as the destination and the other source operand is either a general purpose register or an immediate value. General purpose registers also serve as index registers in indexed addressing mode. If the word 'register' is used in this text, its default meaning is one of the general purpose registers.

- **program counter**. The program counter addresses an address in ROM where the next instruction will be loaded from.

- **stack pointer**. The stack pointer addresses the top of stack in RAM. The stack will grow downward as registers or return addresses are pushed on it. The stack is used by the push/pop instructions, the jsr/rts instructions and by interrupts.

- **status flags**. The status flags indicate the result of the last compare instruction, whether the result of the last instruction was zero or whether the result of the last instruction was all bits set,

E.5.2 Instructions

The proposed processor must have the following instructions. The mnemonics of similar instructions on widely used processors are used.

- **add**. Add, binary operation.

- **and**. Logical bitwise and, binary operation.

- **cmp**. Compare, destination is not changed, only flags indicate whether destination was above, below or equal to the source operand, using unsigned binary arithmetic.

- **cpl**. Complement register (unary operation).

- **djnz**. (optional) Decrement a register and jump if it was not zero.

- **ja**. Jump if flags indicate Above.

- **jae**. Jump if flags indicate Above or Equal.

- **jb**. Jump if flags indicate Below.

- **jbe**. Jump if flags indicate Below or Equal.
- **je**. Jump if flags indicate Equal.
- **jmp**. Jump to ROM address.
- **jne**. Jump if flags indicate Not Equal.
- **jns**. Jump if All-set flag is not set.
- **jnz**. Jump if Zero flag is not set.
- **js**. Jump if All-set flag is set.
- **jsr**. Jump to subroutine at ROM address, push old program counter on stack.
- **jz**. Jump if Zero flag is set.
- **mhi**. Move immediate data to high 8 bits of destination register.
- **mov**. Move source operand to destination operand. Both can be registers, the source can be a memory or immediate operand, in which case the destination is a register. The destination can be a memory operand, in which case the source must be a register. If either of the operands is a non-general purpose register (base register, stack pointer), the other operand must be a general purpose register. As opposed to other instructions, mov occurs in many different instruction formats.
- **or**. Logical bitwise or (binary operation).
- **pop**. Pop a register from the stack (unary operation).
- **push**. Push a register onto the stack (unary operation).
- **rti**. Return from interrupt.
- **rts**. Return from subroutine.
- **set**. Set register to all ones (unary operation).
- **shl8**. Shift a general purpose register left 8 places (unary operation).
- **shr**. Shift a general purpose register right (unary operation). There is no shl, because an add of the register to itself already does this.
- **shr8**. Shift a general purpose register right 8 places (unary operation).
- **sub**. Subtract source from destination (binary operand).
- **xor**. Logical bitwise xor (binary operation).

The instructions fall into several categories.

- **Binary operations**. These are add, sub, 'and', 'or', xor, cmp, mov and mhi. There are 8 binary operations. mhi only occurs with immediate addressing. With the other addressing modes there are two different forms of mov, one to move the source operand to the destination register (load) and one to move the destination register to memory at the address of the source operand (store). With register to register operations, only seven binary operations are possible.

  Note: The architecture was changed to a load/store architecture. This means that the direct and indexed addressing modes are restricted to the move operations (load and store).

- **Unary operations**. These are cpl, pop, push, set and shr. These operations have one register as an operand. Up to 16 different unary operations can be specified. push and pop access memory and modify the stack pointer.

- **Jump operations**. These are djnz, ja, jae, jb, jbe, je, jmp, jne, jno, jnz, jo, jsr and jz. jmp and jsr have a 12 bit address that can be any ROM location. The other jump operations specify an 8-bit signed displacement with respect to the current program counter. The program counter is already incremented before the displacement is added to it.

- **Other operations**. These are rti, rts and other application specific instructions.

The binary operations have the following possible addressing modes:

- **register to register**. The source operand and the destination operand are both registers.

- **direct addressing**. The destination operand is a register and the source operand is in memory at an address that is the sum of a direct base register and a 6-bit address specified in the instruction. There are two directly addressable areas of 64 locations each. One of the areas will be used to store certain variables and the other one will be the I/O area. Only permitted with mov instruction.

- **indexed addressing** The destination operand is a register and the source operand is in memory at an address that is the sum of an index base register and a general purpose register (used as index register). This is the primary mode to access arrays. Only permitted with mov instruction.

- **immediate addressing**. The destination operand is a register and the source operand is an 8-bit value. The high bits of this value are 0. mov with an immediate operand sets the high bits of the destination to zero. mhi sets the high bits of the destination register to the immediate value and leaves the low bits unchanged. The proper way to set r1 to the value 1234h is therefore

```
mov r1,34h
mhi r1,12h
```

The logical operations with immediate addressing are often needed to select certain flag bits in I/O operations. Therefore we will limit the width of I/O ports with flag bits to 8 bits.

The encoding for the instruction set with each instruction fitting in 16 bits is shown in table E.7. The fields have the following meaning. For the encoding of these fields see E.8:

- **addr12**. 12-bit address in jmp or jsr instruction.
- **bd**. 1-bit direct base register specifier.
- **basei**. 4-bit index base register specifier.
- **bop**. 3-bit specifier for binary operation.
- **cond**. 4-bit specifier for jump condition.
- **direct6**. 6-bit direct address.
- **disp**. 8-bit signed displacement in jump conditions.
- **imm8**. 8-bit immediate value.
- **jop**. 1-bit specifier to select jmp to jsr.
- **regd**. 3-bit specifier for destination register.
- **regi**. 3-bit specifier for index register.
- **regs**. 3-bit specifier for source register.
- **uop**. 4-bit specifier for unary operation.

E.5.3 Application-Specific Instructions

For the specific application of Huffman tree computation the following special instructions are desired:

- **fill**. Fill a memory region with a certain value. The value is in the source register. The memory region is specified by an index base and an index register. The source register is stored in memory at the address that is the sum of the index base register and the index register. Then the index register is decremented by one. If the

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

 0  0  0 ---basei--- --regi-- --bop--- --regd-- Binary operation, indexed

 0  0  1 bd -----direct6----- --bop--- --regd-- Binary operation, direct

 0  1 ------------imm8------- --bop--- --regd-- Binary operation, immediate

 1  0  0 jop-------------paddr----------------- JMP. JSR

 1  0  1  0 ------------disp8------ ---cond---- Conditional jump

 1  0  1  1 ------------disp8------  x --regd-- DJNZ

 1  1  0  0  x  x  x  x  x ----uop---- --regd-- Unary operation

 1  1  0  1  x  x  x --regs-- --bop--- --regd-- Binary operation, register

 1  1  1 ---basei--- --regi--  0  0  0 --regs-- FILL

 1  1  1 ---basei---  0  0  1  0  0  1 --regs-- Mov to base register

 1  1  1 db  x  x  x  0  1  0  0  0  1 --regs-- Mov to direct base reg

 1  1  1  x  x  x  x  1  0  0  0  0  1 --regs-- Mov to SP

 1  1  1  x  x  x  x  x  x  x  1  1  0  x  x  x RTS

 1  1  1  x  x  x  x  x  x  x  1  0  0  x  x  x RTI
```

Table E.7: Encoding of the Instructions.

index register is decremented below zero (it then contains all 1 bits), the instruction terminates, else the instruction does not increment the program counter and repeats itself.

This way a region could be filled one location per clock cycle.

- **sel2**. Select second field (unary operation). Shift the destination register right four places and `and' with 15.
- **sel3**. Select third field (unary operation). Shift the destination register right eight places and `and' with 31.

E.5.4 Interrupts

When the machine is reset it starts execution at ROM address 0. When an interrupt occurs, the program counter and status bits are saved and execution proceeds at ROM address 1. The rti instruction restores the program counter and status bits that were saved by the interrupt. Only one interrupt is used. The interrupt handler would poll the various possible interrupt sources to see which event caused it. Between an interrupt and an rti, the interrupt is disabled.

As program addresses are 12 bit and there are four status bits (two for compare result, an all-set flag and a zero flag), it is possible to combine them in on 16-bit word and push that onto the stack.

```
Binary operations (bop field)
0 0 0 Mov
0 0 1 Mov (reverse direction), Mhi with immediate mode
0 1 0 Add
0 1 1 Sub
1 0 0 And
1 0 1 Or
1 1 0 Xor
1 1 1 Cmp

Unary operations (uop field)
0 0 0 0 Pop
0 0 0 1 Push
1 0 0 0 Sel2
1 0 0 1 Sel3
1 0 1 0 Cpl
1 0 1 1 Set
1 1 0 0 (reserved)
1 1 0 1 Shr
1 1 1 0 Shl8
1 1 1 1 Shr8

Jump operations (jop field)
0 Jmp
1 Jsr

Jump conditions (cond field)
0 0 0 0 Jbe (below or equal)  \
0 0 0 1 Ja  (above)            |
0 0 1 0 Jb  (below)            | Last compare
0 0 1 1 Jae (above or equal)   |
0 1 0 0 Je  (equal)            |
0 1 0 1 Jne (not equal)       /
0 1 1 0 Jz  (zero)            \  Zero flag
0 1 1 1 Jnz (not zero)        /
1 0 0 0 Js  (all set)         \  All-set flag
1 0 0 1 Jns (not all set)     /
```

Table E.8: Encoding of the bop. uop, jop and cond Subfields.

Curriculum Vitae

Lennart Benschop was born on February 1, 1967 in Gorinchem, The Netherlands. He attended the 'St. Oelbertgymnasium' in Oosterhout (NB) from 1979 to 1985.

In September 1985 he started to study Electrical Engineering at the Eindhoven University of Technology. He performed his graduation research in the area of information theory at the Faculty of Electrical Engineering under the supervision of prof.dr.ir. J.P.M. Schalkwijk. In February 1991 he received his master's degree in Information Technology from the Faculty of Electrical Engineering.

From 1991 to 1993 he worked as a research assistant at the Faculty of Electrical Engineering. He worked on a design methodology for information processing systems. This research led to the design of a data compression circuit, which later became the subject of his Ph.D. thesis. In September 1993 he received the Registered Technological Designer degree in Information and Communication Technology from the Institute of Continuing Education at the Eindhoven University of Technology.

From November 1993 he worked towards his Ph.D. on the design of a VLSI circuit for data compression under the supervision of prof.ir. M.P.J. Stevens.