



Pilkington Library

Author/Filing Title NUNEZ YANEZ

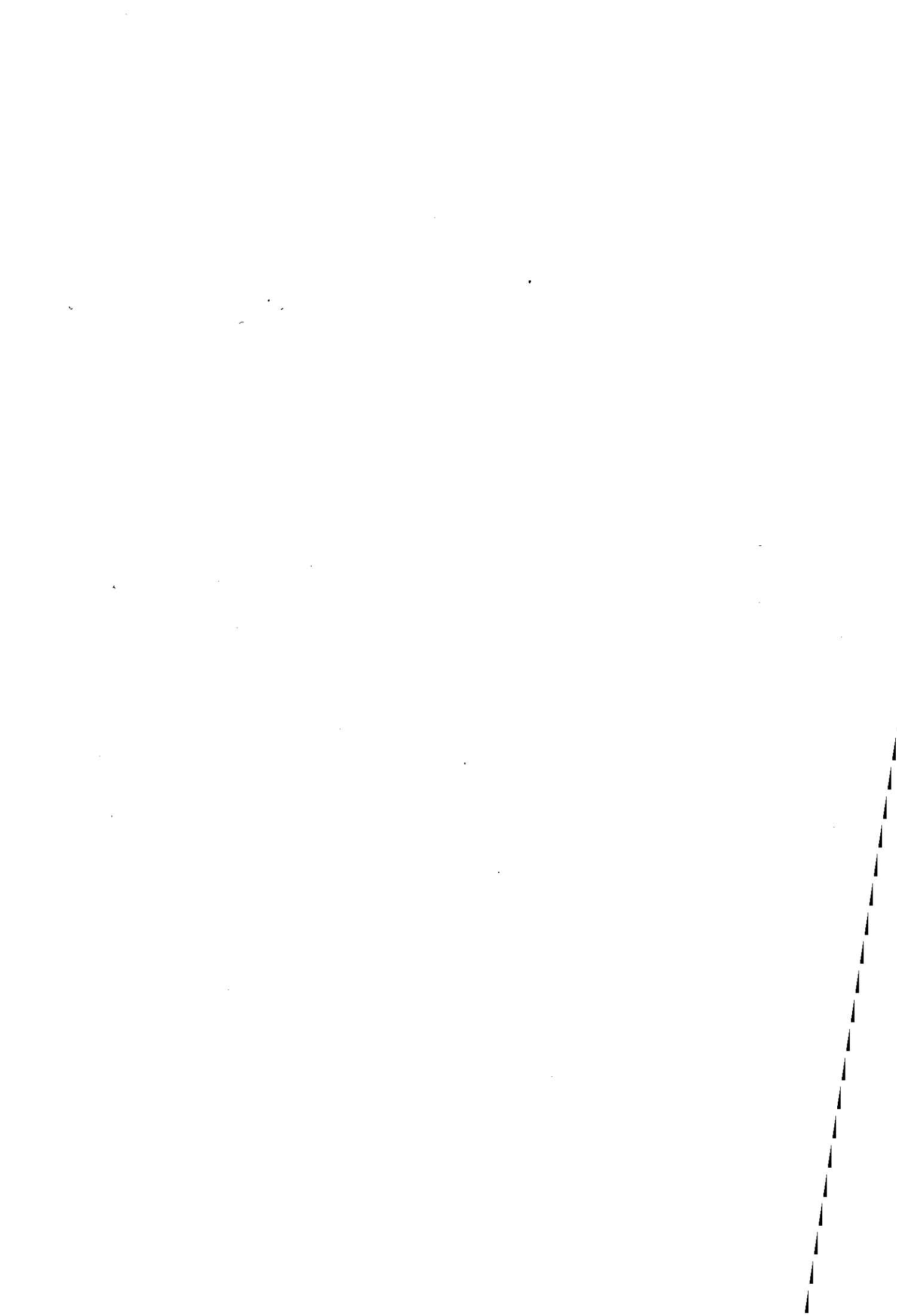
Vol. No. Class Mark T

**Please note that fines are charged on ALL
overdue items.**

FOR REFERENCE ONLY

0402452410





GBIT/SECOND

LOSSLESS DATA COMPRESSION

HARDWARE

By

José Luis Núñez Yáñez


A Doctoral Thesis

Submitted in partial fulfilment of the requirements for the award of

Doctor of Philosophy of Loughborough University

June 2001

© by José Luis Núñez Yáñez 2001

 Loughborough University Library
Date Feb. 02
Class
Acc No. 040245241

Abstract

This thesis investigates how to improve the performance of lossless data compression hardware as a tool to reduce the cost per bit stored in a computer system or transmitted over a communication network.

Lossless data compression allows the exact reconstruction of the original data after decompression. Its deployment in some high-bandwidth applications has been hampered due to performance limitations in the compressing hardware that needs to match the performance of the original system to avoid becoming a bottleneck. Advancing the area of lossless data compression hardware, hence, offers a valid motivation with the potential of doubling the performance of the system that incorporates it with minimum investment.

This work starts by presenting an analysis of current compression methods with the objective of identifying the factors that limit performance and also the factors that increase it. The X-Match method is selected as a promising technique because it offers a level of parallelism not present in other methods combined with low latency. The algorithm analysis focuses on improving its compression ratio typically halving the original uncompressed size. The hardware development phase designs a high-performance architecture that is then implemented in silicon using a non-volatile reprogrammable ProASIC FPGA as our prototyping technology. The device is fully tested at speed to verify its high-performance characteristics achieving over 1 Gbit/second throughput with a 33 MHz clock frequency and latency of only 5 cycles. The compression/decompression engine is then extended to a full-duplex architecture that can handle compressed and uncompressed data streams simultaneously and uses a simple coprocessor-style interface. The full-duplex device offers a combined compression and decompression performance of 3.2 Gbit/second in Xilinx Virtex or Altera Apex FPGA's technologies but its complexity in terms of logic elements is comparable to the half-duplex architecture because the decompression architecture is based on RAM memory readily available in modern FPGA's. This work concludes comparing our device with other high-performance architectures and showing that our chip, named X-MatchPRO, offers unprecedented levels of throughput in a hardware implementation of a general application lossless data compression algorithm. It, therefore, enables the usage of data compression in areas that were traditionally out of reach in previous research.

Acknowledgements

Firstly, I would like to start by thanking my supervisor Professor Simon Jones for offering me the opportunity of registering for a PhD degree while working as a research assistant. It is clear that without his help and guidance this endeavour will not have come to a good end.

Secondly, I would like to acknowledge all the support obtained from Dr Stephen Bateman and his group at Actel/Gatefield corporation. Thanks to them for offering me the chance of spending a very fruitful time at Fremont (California) where I was enlighten to the challenges of hardware testing.

Thirdly, to all the people of the System Design Group at Loughborough University. Especially, to Claudia Feregrino and René Cumplido with whom I have shared the same working environment for the last 3 and a half years.

Fourthly, thanks to my old friend Oscar Fernandez who made the uncertainties of the first year much more manageable.

Finally, to my dad Luis Núñez for always having faith on me and to my fiancé Nicoleta Capet for transforming the last year into a very promising one.

Table of Contents

<i>1 Introduction</i>	<i>1</i>
<i>1.1. Research aim</i>	<i>1</i>
<i>1.2. Basics on data compression</i>	<i>1</i>
<i>1.3 Effects of compressing data</i>	<i>2</i>
<i>1.4 Current applications of lossless data compression technology in communication networks and storage systems</i>	<i>4</i>
<i>1.5 Advances in communication/storage technology generate a motivation for new compression methods</i>	<i>6</i>
<i>1.6 Objectives to be achieved in this research</i>	<i>9</i>
<i>1.7 Thesis structure and method</i>	<i>10</i>
<i>2 Lossless Data Compression Review</i>	<i>12</i>
<i>2.1 Objectives of chapter</i>	<i>12</i>
<i>2.2 Data compression basic definitions</i>	<i>12</i>
<i>2.3 Elements of a lossless data compression system</i>	<i>14</i>
<i>2.3.1 Compression (Modelling, Coding, Packing)</i>	<i>14</i>
<i>2.3.2 Decompression (Unpacking, Decoding, Modelling)</i>	<i>15</i>
<i>2.4 Statistical and Dictionary-based lossless data compression methods</i>	<i>16</i>
<i>2.4.1 Statistical methods</i>	<i>18</i>

2.4.1.1	<i>Statistical modelling</i>	18
2.4.1.1.1	<i>Finite-context modelling</i>	19
2.4.1.1.2	<i>Finite-state modelling</i>	21
2.4.1.2	<i>Statistical coding</i>	21
2.4.1.2.1	<i>Whole-bit coding</i>	22
2.4.1.2.1.1	<i>Shannon-Fano coding</i>	22
2.4.1.2.1.2	<i>Huffman coding</i>	23
2.4.1.2.1.3	<i>Golomb, Rice, Elias, Fiala coding</i>	24
2.4.1.2.2	<i>Fractional-bit coding</i>	25
2.4.1.2.2.1	<i>Full-precision arithmetic coding</i>	26
2.4.1.2.2.2	<i>Low-precision arithmetic coding</i>	28
2.4.2	<i>Dictionary-based methods</i>	30
2.4.2.1	<i>Dictionary-based modelling</i>	30
2.4.2.1.1	<i>Lempel-Ziv 77 (LZ77, LZ1) modelling</i>	31
2.4.2.1.2	<i>Lempel-Ziv 78 (LZ78, LZ2) modelling</i>	32
2.4.2.1.3	<i>BSTW modelling</i>	33
2.4.2.2	<i>Dictionary-based coding</i>	34
2.4.2.2.1	<i>Phased binary coding</i>	35
2.4.2.2.2	<i>Run length coding</i>	35
2.4.3	<i>Other methods</i>	36

<i>2.5 Lossless data compression hardware</i>	37
<i>2.5.1 Statistical hardware</i>	37
<i>2.5.1.1 Binary arithmetic hardware</i>	38
<i>2.5.1.2 Multi-alphabet arithmetic hardware</i>	42
<i>2.5.1.3 Tree-based hardware</i>	46
<i>2.5.2 Dictionary-based hardware</i>	48
<i>2.5.2.1 LZ1 Hardware</i>	48
<i>2.5.2.2 LZ2 Hardware</i>	53
<i>2.5.2.3 Other dictionary-style hardware</i>	55
<i>2.5.3 Other hardware</i>	55
<i>2.6 Summary</i>	56
3 The X-Match method	61
<i>3.1 Objectives of Chapter</i>	61
<i>3.2 Features of the X-Match lossless data compression method</i>	62
<i>3.2.1 Introduction</i>	62
<i>3.2.2 X-Match algorithm description</i>	63
<i>3.2.3 X-Match hardware analysis</i>	65
<i>3.2.3.1 Compressor architecture</i>	65
<i>3.2.3.2 Decompressor architecture</i>	66
<i>3.2.3.3 Hardware performance</i>	67
<i>3.3 Mapping of research objectives to X-Match</i>	68
<i>3.3.1 How can we produce a faster X-Match?</i>	68
<i>3.3.2 How can we produce a better compressing X-Match?</i>	69

3.3.3 <i>How can we prove the feasibility of our solutions?</i>	69
3.4 <i>Conclusions</i>	70
4 <i>Experimental framework</i>	71
4.1 <i>Objectives of Chapter</i>	71
4.2 <i>Data set selection</i>	71
4.3 <i>Hardware selection</i>	75
4.4 <i>Software selection</i>	76
4.5 <i>Technology selection</i>	77
4.6 <i>Measurement definitions</i>	78
4.7 <i>Conclusions</i>	79
5 <i>Focus on compression efficiency</i>	80
5.1 <i>Objectives of Chapter</i>	80
5.2 <i>X-Match compression efficiency</i>	81
5.3 <i>Dictionary-based approach</i>	82
5.3.1 <i>Introduction</i>	82
5.3.2 <i>The dictionary-based model</i>	82
5.3.3 <i>The dictionary-based coder/decoder</i>	83
5.3.3.1 <i>Introduction</i>	83
5.3.3.2 <i>Match location coding techniques</i>	83
5.3.3.3 <i>Run-length coding techniques</i>	87
5.3.3.4 <i>Conclusions</i>	94
5.4 <i>Compression performance comparison</i>	96
5.4.1 <i>Canterbury data set compression</i>	

<i>performance comparison</i>	96
5.4.2 <i>Disc data set compression</i>	
<i>performance comparison</i>	97
5.4.3 <i>Memory data set compression</i>	
<i>performance comparison</i>	97
5.5 <i>Conclusions</i>	98
6 <i>Focus on compression throughput</i>	99
6.1 <i>Objectives of Chapter</i>	99
6.2 <i>Introduction</i>	99
6.3 <i>Model architecture</i>	100
6.3.1 <i>Out of Date Adaptation (ODA) description</i>	104
6.4 <i>Coder/Decoder architecture</i>	108
6.4.1 <i>Run Length Internal (RLI) description</i>	110
6.5 <i>Packer/Unpacker architecture</i>	114
6.6 <i>Compression/Decompression core throughput evaluation</i>	118
6.6.1 <i>Serial test methodology</i>	118
6.6.2 <i>Parallel test methodology</i>	120
6.7 <i>Conclusions</i>	124
7 <i>X-MatchPRO lossless data compression technology</i>	125
7.1 <i>Objectives of Chapter</i>	125
7.2 <i>Full-duplex processing</i>	125
7.3 <i>Width adaptation logic</i>	131
7.4 <i>Full-duplex X-MatchPRO architecture</i>	133

<i>7.5 X-MatchPRO operation</i>	<i>136</i>
<i>7.5.1 X-MatchPRO interface</i>	<i>136</i>
<i>7.5.2 Register bank description</i>	<i>138</i>
<i>7.6 X-MatchPRO threshold value</i>	<i>139</i>
<i>7.7 X-MatchPRO latency</i>	<i>140</i>
<i>7.8 X-MatchPRO operational modes</i>	<i>141</i>
<i>7.8.1 Compression mode</i>	<i>141</i>
<i>7.8.2 Decompression mode</i>	<i>142</i>
<i>7.8.3 X-MatchPRO special conditions</i>	<i>143</i>
<i>7.8.3.1 Buffer coding overflow</i>	<i>143</i>
<i>7.8.3.2 Buffer coding underflow</i>	<i>144</i>
<i>7.8.3.3. Decoding buffer overflow</i>	<i>144</i>
<i>7.8.3.4 Decoding buffer underflow</i>	<i>144</i>
<i>7.9 FPGA-based X-MatchPRO: complexity and performance</i>	<i>145</i>
<i>7.10 Conclusions</i>	<i>150</i>
8 Conclusions	151
<i>8.1 Objectives of Chapter</i>	<i>151</i>
<i>8.2 Summary of the objectives and the research flow</i>	<i>151</i>
<i>8.3 Summary of the X-Match compression method</i>	<i>152</i>
<i>8.4 Main contributions achieved in this research:</i>	
<i>The X-MatchPRO hardware</i>	<i>153</i>
<i>8.5 Other contributions achieved in this research</i>	<i>156</i>
<i>8.6 Measurement of success</i>	<i>158</i>

<i>8.7 Limitations of research</i>	<i>158</i>
<i>8.8 Future work</i>	<i>159</i>
<i>8.9 Summary</i>	<i>160</i>
<i>References</i>	<i>161</i>
<i>Publications & Patent Applications</i>	<i>173</i>

Chapter 1

Introduction

1.1 Research aim

This thesis aims to understand how to improve lossless data compression hardware as a means of boosting the performance of high-speed storage systems and communication networks.

1.2 Basics on data compression

Data compression in a digital system is a process that comprises the removal of redundancy and/or information present in a block of data with the objective of obtaining a reduction in the number of bits that must be transmitted or stored [Bell90], [Lelewer87]. This process can be done in a lossless or lossy way.

Lossless compression allows the reconstruction of the original data after decompression since all the information remains in the compressed block and only redundancy is discarded. Lossy methods on the other hand allow only partial reconstruction since these methods not only remove redundancy but also information. The objective of a lossy compression algorithm is then to remove only information that is of little interest for the intended application. Lossy compression is useful for digital data types that are an approximation to data analogue in nature such as images or voice. Lossless compression can be used with any data type since it is completely reversible and it must be used in data types such as textual or executable data where all the bits are critical. Lossy compression can achieve much higher compression ratios than lossless precisely because there is not a requirement to maintain all the information content of the data source. It is usually possible to define a quality factor that determines the

compression ratio and the fidelity with which the compressed data represents the original data. Lossy compression methods such as the popular image compressors JPEG/MPEG [Wallace91], [MPEG-2] exploit the fact that information can be selectively eliminated from the image during compression as long as the viewer does not perceive the degradation after decompression. The basics of lossless and lossy compression are quite different and many lossy algorithms include a lossless version to be used with those data types such as medical or military image information that cannot accept any quality lost. Lossy compression can achieve typical ratios of 20:1 with good quality. This in contrast with lossless compression where something between 2/3:1 is the standard.

This thesis is wholly concerned with lossless compression methods so by compression we will mean lossless compression unless the contrary is stated. It will also imply compression/decompression since any useful compressor system has a corresponding decompressor.

1.3 Effects of compressing data

Compressing a block of data has 2 main positive effects when applied to computer systems that have to manipulate large amounts of digital information.

- Compression improves throughput in communication applications by increasing the bandwidth available in the transmission links hence the same equipment can achieve a significant increase in the transfer rate. Alternatively, simpler lower bandwidth equipment can replace the high bandwidth one to maintain the transfer rate while using a more economical solution.
- Compression increases the capacity of the physical media in storage applications hence more data can be kept in the same device. It also increases the speed of information storage and retrieval since the time required to access uncompressed data from storage can be significant higher than that of compressed data using fewer bits.

Although data compression has a lot of potential to improve the performance of a digital system it could actually have a negative impact if it is not deployed properly. A number of issues must be taken into account when introducing compression in a data pipe.

- Compression must be done avoiding compressing data already compressed or compressing encrypted data. These 2 events can result in data expansion and degrade the performance of the system unless a detection mechanism is included.
- The compression method must be able to outperform the throughput of the original system. Otherwise compression becomes a bottleneck and the data pipe becomes empty waiting for the compressor to process data. If the compressor can only match the performance of the original system then throughput will be the same but an economical advantage can be obtained with fewer or slower transmission links.
- The uncompressed system throughput (UST), the compressor throughput (CT) and the expected compression ratio (ECR) must be balanced ($CT = UST/ECR$) to obtain optimal performance [Hi/fn97]. For example if a data pipe supports 10 Mbytes/s and the compressor is expected to halve the data traffic its throughput should be 20 Mbytes/s to avoid bubbles where the data pipe becomes empty of any useful content. The effective throughput of the original data pipe plus compression is then 20 Mbytes/s and other components attached to it will forward data to the data pipe at this ratio. If the instantaneous compression ratio is worst than the predicted compression ratio a mechanism must be used to prevent the data pipe from overflowing. Throughput will degrade accordingly but an improvement will be noticeable as long as data expansion is avoided. If the instantaneous compression ratio is better than the expected compression ratio bubbles will appear in the data pipe but the effective throughput will still remain at 20 Mbytes/s.
- Compression produces a variable length output depending on how much redundancy is present in the input data. A more complex management method is required to store and retrieve this data because it is not possible to have an exact knowledge of the capacity of the compressed media.
- Full-duplex technology can carry data in both directions simultaneously. If only software compression mapped to a general-purpose processor or half-duplex hardware compression are available provision must be made to compress part of the time and decompress the rest increasing the throughput requirements of the compression method.
- The increase in latency resulting of applying a compression algorithm could prevent any benefit and data could take longer to arrive to the destination point if the transmission

time includes latency plus transmission and a mechanism is not available to do both operations concurrently masking latency with transmission time.

1.4 Current applications of data compression technology in communication networks and storage systems

It has been widely accepted that the performance of a storage system or a communication network can be improved by a typical factor between 2/3 by the use of lossless data compression [Hi/fn97], [Cyclades00], [Jung98], [Mogul97], [Mitel00]. Indeed nowadays data compression is widely used in communication devices such as routers, bridges and modems to increase the bandwidth of networks such as LAN, WAN and wireless [AlliedTelesyn00], [Intel00], [Cisco00], [Dickson00]. Storage systems such as file servers, solid state storage, hard disk drives, tape drives use data compression not only to increase capacity but also to increase the available bandwidth to move data in and out of the device [VanDuine00], [Cressman94]. Compression is also useful in other applications that benefit from a reduction in the amount of data that must be stored or moved such as printers, copiers and scanners.

The use of data compression methods has thrived thanks to the exponential growth in bandwidth and storage requirements combined with the need to keep costs within a budget. It seems that, although technology advances are constantly increasing the bandwidth and capacity of transmission and storage media, the applications that run on them always find ways to use all the resources available and create a need for more. The consequence is that sometimes the technology is not available or the cost of its implementation is uneconomical. Compression is an effective way to alleviate this problem. Figure 1.1 obtained from [Cyclades00] uses an example to illustrate the cost benefits of data compression applied to a wide area network (WAN).

Line Speed	Approximate Cost	Effective Throughput	Cost per Kbps
56 Kbps	\$125/ month	112 Kbps (with data compression)	\$1.1
128 Kbps	\$325/month	128 Kbps (no data compression)	\$2.5

Figure 1.1. Savings introduced by compression in a wide area network.

Figure 1.1 shows that a similar bandwidth can be obtained with a lower speed line halving the costs of the line rental.

A typical network configuration includes a group of high-speed LAN's interconnected using a low-speed WAN network. The expensive WAN can easily be a bottleneck because it concentrates the data traffic exchange among the LAN networks. Modern routers use compression to optimise WAN efficiency but there is not a unique standard dealing with the compression method to use. This means that some form of negotiation based on a compression control protocol must be established between receiver and transmitter to agree which compression method to use. Nevertheless, the LZS [Hi/fn96], [Hi/fn99] algorithm a LZ1 (Lempel-Ziv-1) [Ziv77] derivative from Hi/fn has emerged as the preferable method in many cases because of its high throughput and good compression ratios. Popular router manufactures such as Cisco [Cisco00] and Intel [Intel00] support LZS compression. LZS has been accepted as standards ANSI x3.241-1994 (American National Standards Institute), QIC-122 (Quarter Inch Cartridge), IETF RFC1974 (Internet Engineering Task Force), FRF.9 (Frame Relay Forum) [Hi/fn96] among others for storage and communication applications.

Compression is routinely used in modems thanks to the v.42bis standard proposed by the CCITT (Comite Consultatif International Telephonique et Telegraphique) [Thomborson92], [Acorn92]. The v.42bis standard uses a variant of the LZW [Welch84] compression algorithm also used in the UNIX utility '*Compress*' and itself a derivative of the LZ2 (Lempel-Ziv-2) [Ziv78] algorithm to increase data throughput. It is meant to be implemented in modem hardware but it is also possible to include it in the software that interfaces to a non-compressing modem. The algorithm defines a way to monitor compression efficiency and switch to transparent mode when data expands.

Something that WAN and modem compression have in common is that the speed requirements are quite low. The V90 [Intel01] standard for modems defines a throughput of 56 Kbytes/s while typical WAN throughputs such as T1 WAN [Tanenbaum96] are in the order of 1.5 Mbytes/s . This means that in many cases compression can be supported in software running in the same CPU that handles the rest of the functions present in the communication protocol. If this is not enough a coprocessor processing in the order of Mbits/s will suffice.

Another data compression method that has achieved commercial success is the ALDC algorithm, another LZ1 derivative developed by IBM [IBM94], [Cheng95], and also available from AHA (Advanced Hardware Architectures) [AHA97]. It has been accepted as standards ISO/IEC 15200 (International Organisation for Standardisation/ International Electrotechnical Commission), ECMA-222 (European Computer Manufacture Association), ANSI x3.280-

1996, QIC-154 [Craft98]. The IBM AS/400 family of high performance server's [VanDuine00] able to handle terabytes of data distributed over a number of hard disks and tape drives use ALDC compression integrated in the storage controller. This is quite a unique solution because the use of compression in hard disk devices remains something of an ad-hoc technique. The dedicated compression chip brings an important capacity gain factor 2 to 4 and minimizes any performance impact if compared with a software-based solution. The overhead of the compression process is higher than that of the decompression process so data compression is better used with read intensive applications such as databases.

The most popular way to introduce compression in a hard disk in user transparent mode is controlled by the operating system and based in software such as those present in MS-DOS DoubleSpace and Stacker or Windows DriveSpace. This compression technology has generated some controversy on its reliability in the past [Halfill94]. Popular compression utilities like WinZIP, ARJ, PkZIP are file compressors not designed to work in a blocked mode which is needed to allow fast random access to the uncompressed data. They are especially useful for backup purposes where speed is not an important issue. Their main inconvenience is that they are user initiated and too slow to be applied in real-time environments.

Compression is commonly present in tape drive technology such as QIC (Quarter Inch Cartridge), DAT (Digital Audio Tape) and DLT (Digital Linear Tape) with the main objective of increasing data capacity. Tape drives concentrate on offering high data capacity for backup purposes and not for on-line access. Speeds of 6 Mbytes/s with a compressed capacity of 80 Gbytes are offered in the high-performance DLT8000 [Quantum99] products. The DCLZ [AHA96] algorithm, a LZ-2 derivative developed by Hewlett/Packer [Bianchi89] has been accepted as standards QIC-130, ECMA-151, ANSI-X3.223, ISO/IEC-11558 [AHA95]. This method seems to be the preferred choice for tape compression [Cressman94], [Seagate97]. AHA (Advanced Hardware Architectures) acquired DCLZ technology from Hewlett/Packer and it currently offers several devices with throughputs around 20 Mbytes/s [AHA97b].

1.5 Advances in communication/storage technology generate a motivation for new compression methods

Recent advances in networking technology and the significant requirements for bandwidth and data capacity generated by applications such as real-time video conferencing, 3D animation modelling, Internet telephony, virtual reality, video on demand, etc have made

some storage/communications equipment to operate at speeds in excess of 1 Gbit/s. Optical communications are a good example of the sort of systems where Gbit/s throughputs are reached. Gigabit networking [Vandalore95] has been made possible thanks to fibre optic signalling equipment able to transmit at a bandwidth of several Gigabit/s over long distances with low error rates. Storage equipment has benefit from technology such as RAID (Redundant Array of Inexpensive Disks) [Storage00] to achieve over 1 Gbit/s bandwidth performance.

There are 3 popular networking technologies working at speeds in the order of Gbit/s:

Gigabit Ethernet (IEEE 802.3z): Gigabit Ethernet [GEA97] specifies the data link layer (layer 2) of the OSI (Open System Interconnection) [Tanenbaum96] protocol model and it has been the most widely-used high-bandwidth LAN networking technology for the past few years. It has been endorsed by major companies in the field such as Cisco systems and 3Com and also by legions of start-ups. Since Ethernet (IEEE 802.3 at 10 Mbits/s) and FastEthernet (IEEE 802.3u at 100 Mbits/s) are the most popular LAN technologies a gigabit/s version offers a smooth upgrade path since it is cost effective and it does not require new specific training. It uses the same IEEE 802.3 frame format and flow control methods which means that it is simple to connect a LAN using Gigabit Ethernet as the backbone to a number of servers/terminals internally using Ethernet devices running at lower speeds. There is also an effort to include specifications for MAN (Metropolitan Area Network) and WAN (Wide Area Networks) in future versions of high-speed Ethernet [Caruso99a]. The requirement to keep compatibility with older technologies has created some performance problems such as failing to deliver true QoS (Quality of Service) required by some applications like video on demand. Although work has been undertaken in providing QoS at higher layers than the link layer with the use of network protocols such as RSVP (Resource Reservation Protocol) it remains a best effort protocol. This has prevented Gigabit Ethernet from offering a complete solution to the bandwidth problem.

ATM (Asynchronous Transfer Mode): ATM [Pivotal97] is also a link layer protocol like Ethernet. It was introduced earlier than Gigabit Ethernet to be used in LAN's as well as WAN's in those applications demanding a lot of bandwidth. It initially offered 155 Mbits/s in ATM OC-3 with a path up to ATM OC-128 offering 6.4 Gbits/s. ATM was thought to be the perfect solution to the bandwidth problem but that did not happen. In ATM it is possible to guarantee QoS, very important in applications such as video on demand, but it is more expensive and the migration path is more complex than using Ethernet. ATM uses fixed length cells of 53 bytes enabling extremely fast hardware-based switching in direct contrast

with Ethernet where packet length varies from 64 to 1514 bytes. The ATM protocol includes standards that provide LAN emulation of networks such as Ethernet/Token Ring so it is possible for an application to communicate to an Ethernet network unaware of using ATM. In many cases ATM offers a best solution if it is used as a backbone of a WAN joining together different LAN's where Ethernet technology is at its best.

Fibre Channel: Fibre channel [Burton95] defines a complete multi-layered stack of functional levels from the physical layer to the upper-level application interfaces. It can run at speeds up to 1062 Mbits/s. It seems to be the preferred solution to attached storage devices to a host computer forming Storage Area Networks (SAN). Its use as a gigabit networking technology is not as popular [Mace98]. Storage Area Networks are formed by a series of storage nodes and server nodes sharing a common pool of data that can be physically separately up to 10 Km using Fiber channel based on fibre optic cables or 30 meters over copper wires. The storage nodes can be external rack-mounted RAID subsystems formed by a number of SCSI drives to offer capacities of terabits of data. SCSI ultra-2 disk drives run up to 80 Mbytes/s while more recent SCSI ultra-3 technology offers a throughput of 160 Mbytes/s well over 1 Gbit/s. Recent RAID storage solutions are offering throughputs over 200 Mbytes/s [Storage00]. RAID technology [Deil99] is a method of combining several hard drives in a single unit offering a higher level of fault tolerance and throughput. Fault tolerance is achieved by writing the same block of data to a pair of disk. Improved performance is achieved by distributing data evenly across the disks to equalise disk accesses. If multiple disks in a RAID subsystem are being accessed simultaneously performance improves proportionally. The RAID controller portrays the multiple disks as a single unit to the application.

Other Gbit/s networking technologies include serial HiPPI: (High Performance Parallel Interface) [Djumin97] that operates within the physical and link layers at speeds of 1.2 Gbit/s over distances up to 10 Km. It offers Gbit/s performance and high reliability at the physical layer whilst other protocols relay in higher layers for data lost detection. Also there has been recent interest in mapping directly IP-over-SONET [Trillium97] to avoid the overhead incurred by the mapping IP-over-ATM and then over SONET. While IP (Internet Protocol) is the typical network protocol in layer 3 in the OSI model SONET (Synchronous Optical Network) is a physical layer protocol (layer 1). These efforts have given way to future trends towards 10 Gbit/s standards. 10-Gigabit/s Ethernet [Caruso99b] and 10-Gigabit/s SONET are 2 examples of where high-speed networking seems to be heading. These technologies should be available within the next few years.

Data compression is not currently being used to its full advantage in these systems due to performance limitations encountered in the data compression hardware, although if properly deployed it could double the performance and capacity of storage/communication systems with minimum investment. Storage Area Networks (SAN) using fibre channel to communicate high-capacity and high-speed disk arrays or a Gigabit Ethernet backbone connecting a group of Ethernet LAN's running at lower speeds are current examples where present compression technologies fail to deliver the required performance for successful integration. To realise fully the benefits of data compression in these areas requires a compression technology that matches the throughput of the original system.

1.6 Objectives to be achieved in this research

The overall aim of this research is to improve the speed and compression of lossless data compression hardware. In order to achieve these aims we can identify the research objectives and then we can map them into the thesis structure.

1. The first work to be undertaken is the identification of the factors that improve/limit current lossless data compression hardware. A survey on current compression technology will provide us with common limitations that hamper performance and also the features that boost it.
2. We will then develop solutions that will try to avoid the common bottlenecks found in current technology and improve the factors that define the efficiency of a compression method namely:
 - The speed at which the compression/decompression processes are executed.
 - The average compression ratio that the method can achieve on typical data.
3. Once we have identified a set of solutions that we believe achieve the aim of improving compression technology we will demonstrate the feasibility of these solutions by developing a practical hardware architecture and mapping it into available silicon. The final output and the core to evaluate how well we have achieved our initial aim will be the performance figures obtained by this hardware device.

1.7 Thesis structure and method

The research objectives can be mapped into the thesis structure as follows.

Chapter 1 is an introduction whose objectives are as follows: Firstly to brief the reader on the basic concepts on lossless and lossy compression methods. Secondly to establish the motivation of this work based on the applications and current state of compression technology. Finally to propose a set of objectives and the methodology to be followed to achieve them.

Chapter 2 is concerned with a background revision and systematic classification on previous research efforts. This review will show the features that limit and boost compression performance and will help us to identify a suitable way to progress further.

Chapter 3 involves the selection of a research vehicle to base our experimentation. We will use the information provided in chapter 2 to justify the selection of a method that shows high performance characteristics.

Chapter 4 selects a common development framework to base the experimentation. The selection identifies the data sets and compression methods to be used for the compression ratio and compression speed figures and justifies their selection. The process aims to select state-of-the-art methods so a meaningful comparison can be done between them and our own method. The technology to be used for the hardware implementation is also chosen.

Chapter 5 focuses on improving the compression efficiency defined as the average compression ratio output/input on the typical data sets selected in chapter 4. A set of solutions and their implications on complexity and speed will be described. We will select some of these solutions to progress further based on 3 interrelated parameters: compression, speed, and complexity. Since our overall goal is to identify a feasible architecture and to demonstrate it in hardware it is important that complexity does not exceed that of currently or soon available hardware.

Chapter 6 focuses on improving the compression speed defined as a function of 2 variables: throughput and latency. Throughput is defined as the constant and data independent uncompressed data rate and it is measure in bits/second. Latency is defined as the time it elapses since the last input symbol enters the device until the devices is ready to start a new

operation and it is measured in cycles. Again the 3 factors must be taken into account since it is usually possible to increase throughput by reducing compressing efficiency. The importance of each factor is dependent on the application but it is possible to guide the process by a selection of figures: typical lossless compression that halves the original uncompressed data, throughput over 1 Gbit/s, latency around 10's of cycles and complexity in the order of 100's of thousands of gates. Finally, the proposed core architecture is mapped to our silicon test bench and tested to prove their benefits.

Chapter 7 extends the compression engine developed in chapter 6 to a full self-contained lossless data compressor coprocessor and maps it into the technologies selected in Chapter 4. A final comparison is made between the features of our device against other high performance lossless data compressor chips.

Chapter 8 concludes this thesis evaluating how well we have achieved the objectives initially proposed. It also shows the limitations of the current work and identifies a path where future work could be undertaken.

Chapter 2

Lossless Data Compression Review

2.1 Objectives of Chapter

This chapter presents a review on the area of lossless data compression. The objective is to analyse current lossless data compression methods and then to select a set of interesting concepts for further research in the following chapters.

Firstly, we will introduce some basic concepts on data compression and assess the main components present in a lossless data compression system, then continue with an investigation on recent advances in software and hardware data compression and finally conclude highlighting the features common to high performance lossless compression methods.

2.2 Data compression basic definitions

Lossless data compression is possible because some of the bits that form a symbol contain redundancy. It is possible then to devise a mechanism to eliminate the redundant bits and still maintain the complete meaning of such a symbol. The amount of information in bits of a symbol a is given by the expression:

$$\text{number_of_bits} = -\log_2(\text{probability}(a)) \quad [2.1]$$

where $\text{probability}(a)$ is the probability of occurrence of symbol a . This for example means that if the probability of occurrence of a symbol a is 1 then the information content of that

symbol is 0 and 0 bits are needed to code it because in essence no other symbol can happen in the receiver end. On the other hand if the probability of occurrence of a symbol b is 0 then from equation [2.1] infinite bits would be needed to code it and in essence the coding operation can not take place. This will happen with an alphabet with infinite number of symbols that cannot be coded. Using equation [2.1] the minimum number of bits needed to represent a symbol c with probability 0.9 is 0.152. If the symbol is a single bit an optimal coder will be able to remove 0.848 bits and the decoder in the receiving end will still be able to know if a 0 or 1 was transmitted. The information content of a block of data that uses an alphabet of size n can be obtained weighting the information content of each symbol with its probability of occurrence producing the expression:

$$H = -\sum_{i=1}^n (\text{probability}(a_i) * \log_2(\text{probability}(a_i))) \quad [2.2]$$

These 2 expressions [2.1] and [2.2] are due to Shannon [Shannon48]. H is known as the entropy or information content of a data source and forms the basis of the information theory due to the same author. It represents the minimum number of bits needed on average to code an input symbol using a given probability distribution and a lower bound to measure the efficiency of any coding method. The equations made a clear distinction between model and coder. The model is a collection of data that identifies where the redundancy is located in a message while the coder is a mechanism to exploit this information to reduce the number of bits needed to represent the original message. Equation [2.2] establishes that lossless compression is possible because some symbols or groups of symbols have a higher chance of occurrence (probability) than others. As a direct consequence true random data is impossible to compress because it contains no redundancy and all the symbols have the same probability $p=1/\text{alphabet_size}$ producing a flat probability distribution with a value of H that equals $\log_2(\text{alphabet_size})$. A useful definition to measure the efficiency of a compression method is the compression ratio (CR) of equation [2.3] where compressed output and uncompressed input are measured in number of bits. Compression is obtained whenever the CR is in the range of (0,1). This measurement will be used in the rest of this work.

$$CR = \text{Compressed Output/Uncompressed Input} \quad [2.3]$$

2.3 Elements of a lossless data compression system

In any lossless compression system it is possible to identify 3 components with different functionality. These are: the model, the coder and the packer. The same 3 elements are present in the decompressor but their function is now opposite in 2 of them (the unpacker and the decoder) whilst the model is used in the same way.

The separation between model and coder is particularly useful to classify the 2 main families of lossless data compression methods: dictionary-based methods and statistical methods. It reflects the fact that once we have decided which modelling technique to use for our data, the coding method is not fixed and a wide range of techniques remain available to choose from. Although some coding methods map better than others depending on the chosen model, many different combinations are possible.

These 3 components must be applied in the right order as shown in Figure 2.1.

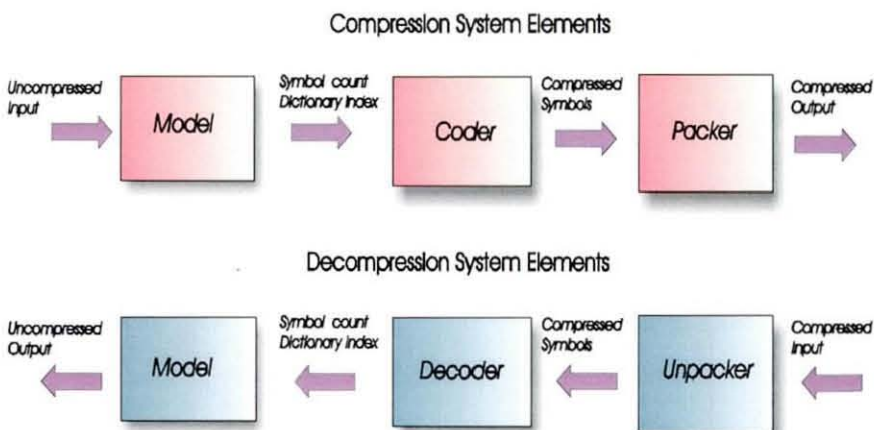


Figure 2.1. Elements of a Lossless Data Compression System

2.3.1 Compression (Modelling, Coding, Packing)

- The function of the model during the compression process is to identify where the redundancy is located in the data source and to signal repetitive data sequences to the coder. The model uses past experience obtained from processing the input data source to guide these 2 tasks. The model performs the same function in compression and decompression and it must be maintained in synchrony matching all the compression states during decompression to ensure proper decoding.

- The function of the coder is to assign a number of bits to each event notified by the model. A non-trivial coder will use the information passed by the model to code more common data using fewer bits than to code less common data and therefore to increase the compression efficiency of the method. A trivial coder will assign the same number of bits to each event.
- Finally the packer is used to group the variable or fixed length codes output of the coder in fixed length units depending on the word width of the system before they are output as compressed data.

Models can be adaptive, semi-adaptive or static:

- In adaptive models the adaptation or learning process takes place concurrently to the compression process. The model dynamically changes the information it stores depending on the properties of the data source. After receiving a symbol an adaptive model obtains the information that describes it using its internal history and passes this information to the coder. It then performs an adaptation function modifying its internal history to reflect the symbol just seen.
- Semi-adaptive models use a two-pass approach where in the first pass the model adapts and in the second pass compression takes place with a static model providing the information to the coder.
- Static models use the same information to process any data source. Its usefulness is limited because for example a general model obtained from compressing text might offer a very inaccurate representation of an image file.

Adaptive models are usually preferred because they offer superior performance. They avoid the overhead of having to process the data source twice and/or the need to transmit model information to the decoder. This work is mainly concerned with adaptive models.

2.3.2 Decompression (Unpacking, Decoding, Modelling)

- The unpacker function is to break the compressed input data stream into units where the boundaries correspond to compressed symbols. The unpacker needs information about the compressed length of the previously uncompressed symbol that must be provided by

the decoder before it can disregard the bits used in the previous decoding step and shift in new compressed data for a new cycle. This property of the decoding process creates a feedback loop between coder and unpacker and it means that it is quite difficult to pipeline these 2 stages. The job of the packer/unpacker in some dictionary-based techniques that obtain compression by replacing variable-length groups of symbols with fixed-length codes can, however, become trivial. This variable-to-fixed way of operation means that the boundaries between compressed symbols are fixed so the previously mentioned feedback loop does not exist.

- The decoder transforms the compressed data into indices or pointers to tables where the uncompressed data can be found in the model. These pointers could be addresses to dictionary locations in dictionary-based methods or arithmetic values in the range between 0 and 1 in statistical methods.
- The model uses the index information provided by the decoder to obtain the corresponding uncompressed data and output it. The uncompressed data could be a group of symbols in a dictionary-based method or a single symbol in a statistical method. The model also uses the uncompressed data to perform the same adaptation function as the compression model to keep in synchrony and maintain correct operation.

2.4 Statistical and dictionary-based lossless data compression methods

Statistical methods show a more clear separation between model and coder. Statistical models are based on assigning a value to symbols depending on their probability using the rule: the higher the value the higher the probability. The accuracy in which this frequency assignment reflects reality determines the efficiency of the model. The model passes this frequency information in form of symbol count and total count to the coder. The coder objective is to use few bits to code symbols with high probability and vice versa. Compression is obtained if the symbols that get assigned shorter codewords prove to be most popular in the input data source. Again adaptive models are preferred because they offer superior performance and since they start with an empty state they do not need to transmit the model as part of the compressed data. These methods are also called symbol wise methods because they process each input symbol independently in contrast with dictionary methods that group symbols together. Statistical methods tend to use a form of a dictionary to hold the active subset of the working alphabet and this concept should not be confused with dictionary-based modelling.

The dictionary used in a statistical method has frequency counts associated to its locations and this is not true in dictionary-based methods.

Dictionary methods try to replace a group of symbols by a dictionary location code or dictionary address that points to a dictionary position that stores the same group of symbols. Compression is obtained as long as the location code uses fewer bits than the group of symbols it replaces. It is characteristic in these methods to give the modelling stage an extra importance whilst the coding stage is simplified. They are simpler than statistical methods and tend to run faster with good compression ratios. For this reason dictionary compression remains as the most popular both in hardware and software although the best compression ratios are found in the area of statistical compression [Bell89]. The information pass to the coder by the model is a dictionary location plus information relating to the match length. This information can be sent to the bit packer directly without further processing by the coder or the coder can try to assign shorter codes to those index/length combinations that prove to be more popular.

Hybrids are also quite popular with combinations mainly between dictionary models with statistical coders. Figure 2.2 shows a classification of modelling and coding techniques for lossless data compression and examples in each category.

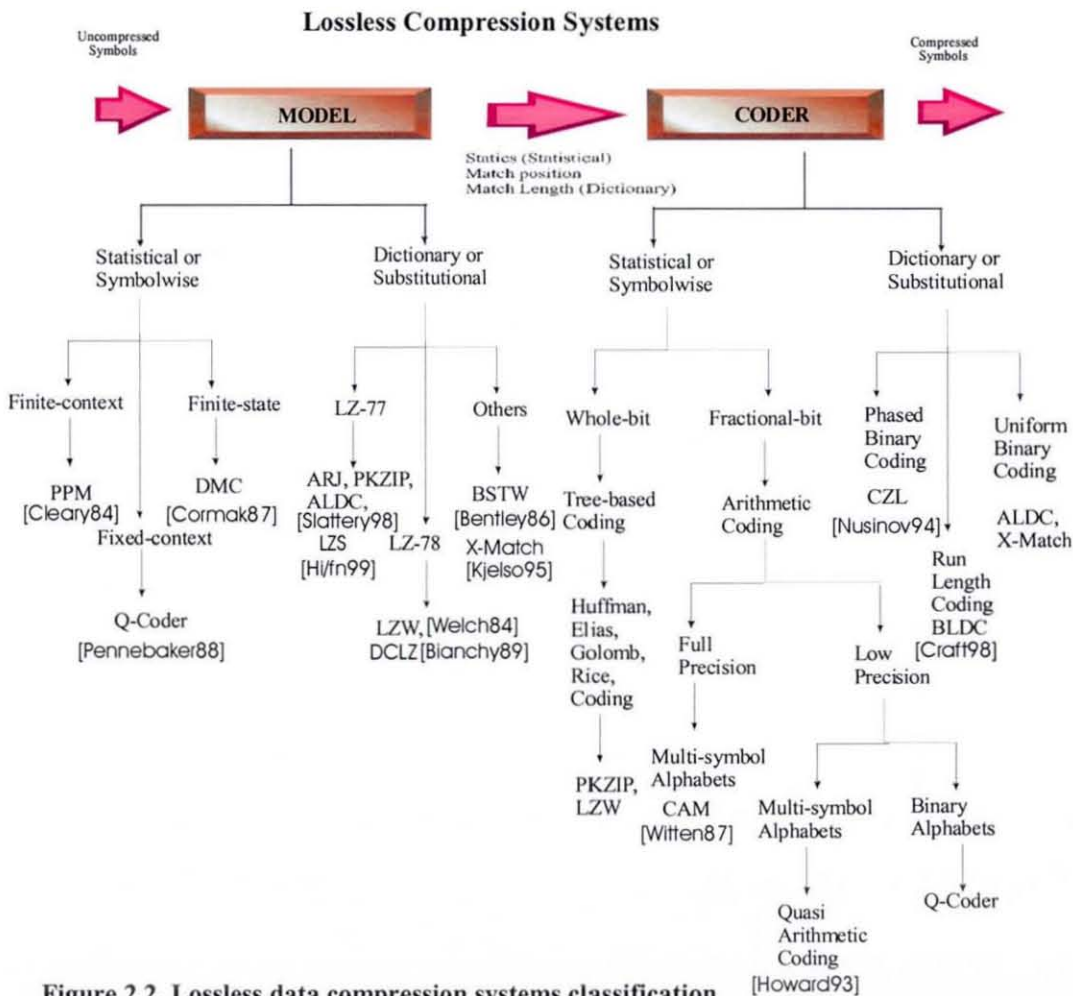


Figure 2.2. Lossless data compression systems classification

2.4.1 Statistical methods

2.4.1.1 Statistical modelling

Statistical models are based in doing predictions on the expected next symbol using the statistical information gather during the processing of previous data. Simple statistical modelling is based on assigning a count higher than 0 to any possible symbol in the alphabet and then to increase these counts according to the symbols being processed. This modelling strategy is usually called a context-free statistical model. In this simple model it is important to start the model with a count higher than 0 for any possible input symbol to avoid the *zero frequency problem* [Cleary95a], [Witten91]. The *zero frequency problem* occurs when the coder tries to code a symbol with a count of 0 because the equation that drives the coder – $\log_2(\text{probability})$ fails if $\text{probability} = 0$.

The value of the probability for a symbol ‘a’ is given by:

$$\text{probability}(a) = \frac{\text{symbol_count_of_a}}{\text{total_symbol_count}} \quad [2.4]$$

The information that the coder requires from the model is the probability of the symbol ‘a’. It then becomes the responsibility of the coder to use this information efficiently to obtain compression. If the probability information provided by the model is inaccurate the coder will fail to compress the symbol and it might even expand it (use more bits than in its original representation) thus showing the importance of good modelling.

This simple context-free modelling technique does not use the concept of dictionary because all the input symbols are present in the system from the start. The concept of a dictionary in a statistical method appears when not all the possible input symbols are assigned a frequency count higher than 0 and an escape mechanism is enabled to avoid the *zero frequency problem*. Statistical models use dictionaries when the alphabet is too large to be handled simultaneously (for example if system granularity is words instead of bytes) or if a context-based technique is being used. In these cases a dictionary is used to hold the alphabet subset that is active at that moment. The dictionary locations in a statistical model have frequency counts associated with them and this feature avoids confusion with dictionary-based modelling that will be discussed in section 2.4.2.

Context-free modelling offers modest compression ratios because the probabilities tend to have low and similar values with values approaching 1 for a symbol being rare since all the

other symbols in the alphabet must also be accommodated. Probability values approaching 1 can be obtained by exploiting the concept of context-based modelling. Context-based techniques exploit the fact that a prediction can be made with much more certainty by observing the symbols that have just preceded the current symbol. They will be analysed in the following sections.

2.4.1.1.1 Finite-context modelling

A real breakthrough in statistical modelling came with the introduction of context-based prediction and context-blending techniques in [Cleary84] with the PPM (Prediction by Partial Matching) algorithm.

PPM methods extract the redundancy present in a block of data using a variable-order context-based statistical model. A key concept in PPM is model order. The order of the model defines the maximum number of symbols that can be used to predict the next symbol. The symbols that are used to predict the next symbol are called context. In other words, a context is formed by symbols and the maximum number of them defines the order of the model. For example a first order model working with English text will find that the probability of 'h' following a 't' is much higher than the probability of an 'h' on its own. Then after activating context 't' because a 't' has been received the system will predict that an 'h' will follow with a 95% probability. If a 'h' does follow much greater compression will be achieved. Assuming a 256 symbols alphabet an optimal coder will assign to symbol 'h' only $-\log_2(0.95) = 0.07$ bits which is a big reduction over the original 8 bits. Of course, if the prediction fails and for example not symbol 'h' but symbol 'w' follows more bits will be needed to code a symbol with low probability and indeed no data compression but data expansion could take place. Any symbol predicted with probability lower than $1/256 = 0.004$ (0.4 %) will expand when coded because $-\log_2(1/256) = 8$ bits.

The PPM methodology assumes that the higher the order the more precise the prediction would be and fewer the bits needed to code it. For example, let's imagine an extreme case where the order of a model was as high as all the letters contained in a book except the last one. A prediction on the last letter using this context would be made with almost 100% certainty and would not create almost any output since no 2 books are the same but the last letter. The only uncertainty will be left to spelling errors. This system is unfeasible but illustrates the idea of prediction with high orders.

PPM methods do not assign a count higher than 0 to all the possible input symbols, only to those that have been seen after a particular context. To avoid the *zero frequency problem* described in section 2.4.1.1 an escape technique is used so the system can vary its order falling from a higher order to a lower order if a valid prediction is not possible in the first one. This variable-order feature is enabled by the escape mechanism that effectively blends together all the different orders present in the system. When a particular order fails to make a prediction because the item being predicted is new to that context the escape mechanism is activated. The model tries to use the next lower order and so on until the item is successfully predicted or the 0th order, where the context is empty, is used. The 0th order has to be implemented in a way that any possible input symbol has a count higher than 0. The context-free model described in section 2.4.1.1 corresponds to a 0th order model. Depending on the implementation an order -1 where all the possible symbols have the same fixed probability could be defined. In this case 0th order is allowed to fail to make a valid prediction an escape to order -1.

The size of the alphabet is typically byte-based to exploit the fact that most data exhibits a byte granularity. Binary alphabets are also popular due to its simplicity mainly in hardware implementations. PPM word-based compression has also been analysed by [Moffat89] with a word defined as maximal sequence of alphabetic characters and a non-word as maximal sequence of non-alphabetic characters keeping statistics separately for both distributions. His results show an important compression benefit when replacing a 0th order model with a 1st order model. Higher-order modelling shows no advantage for word-based compression.

Several variations from the original PPMA and PPMB methods described in [Cleary84] have appeared modifying how the escape probabilities are calculated to improve how the orders blend together. This research has produced methods such as PPMC [Moffat90] and PPMD [Howard93a] each of them offering some improvement over the previous one.

A lot of research has been done in choosing an optimal maximum context length. The classical approach based on byte alphabets uses an upper bound with a context length of 4 or 5 symbols while showing that further extensions of context length damage compression due to an excessive use of the escaping mechanism. However a more recent approach named PPM* [Cleary95b] uses unbounded context lengths to achieve superior performance. Unbounded-length contexts are formed by all the symbols that have been seen in the input stream and used efficient data structures to maintain complexity under reasonable limits. They also exploit the use of deterministic contexts or contexts that make a single prediction. Other refinements aimed at improving compression is the inclusion of a local order estimation

(LOE) technique that makes a prediction based on the input stream characteristics on which context length should be used for the next symbol.

These techniques have been utilised in PPMZ [Bloom98]. PPMZ uses LOE to choose a context length between 12 and 0 when a deterministic unbounded-length context has failed to make a prediction. PPMZ is considered to be one of the best data compressors available in the literature but it achieves this by imposing high demands on memory resources and CPU performance. Execution is measured around 1 symbol every 20K CPU cycles while the demand on memory resources is around 30 times the size of the file being compressed.

2.4.1.1.2 Finite-state modelling

Finite-state modelling is based on a state transition graph formed by nodes representing states and edges leaving and entering the nodes representing transition probabilities between the states. Finite-state models can construct the finite-context models of the previous section with ease. For example a single node can represent a simple byte-based 0th order context-free model with 256 transitions leaving and entering the node. Each edge would be associated with the probability of a byte occurring. A byte-based 1st order finite-context model would have a finite-state equivalent model formed by 256 nodes each of them with 256 transitions leaving the node and entering the same node and the other 255 nodes. Finite-state modelling can also built more complex structures to reflect data behaviour that can not be adequately represented with finite-context modelling. The draw back with finite-state adaptive models is that their construction and maintenance is more difficult with techniques based on heuristics instead of mathematical analysis [Bell89]. Adaptive model construction is usually based on starting with a simple model with a single node and then duplicate or clone the node based on parameters related to node usage. If a transition to a particular node from different nodes proves to be popular it is duplicated to capture which states contribute the most. The more popular implementations of finite-state modelling are based on binary alphabets where each node or state has only 2 possible next states [Cormak87]. This simplifies the managing of the model and also suits arithmetic coding since binary coders are much simpler to implement.

2.4.1.2 Statistical coding

The function of a statistical coder is to use the frequency information provided by the model to produce a minimal number of bits an obtain compression. A good coder will output a number of bits close but never fewer than $-\log_2(\text{probability})$ for a given model since this

quantity defines the information content or entropy of the model and it is the optimal code length.

The spectrum of statistical coding techniques expands from the fast but sub-optimal prefix coders to the slow but optimal arithmetic coders with a range of coding techniques located somewhere in between trading speed for coding efficiency.

The prefix coders or whole-bit coders which are derived from the well known Huffman [Huffman51] codes are sub-optimal because they only produce an optimal output when the probability distribution of the input symbol matches exactly $1/(2^x)$ where x is an integer and positive number.

The arithmetic coders belong to the class of fractional-bit coders and are known as being optimal because their output can be arbitrarily close to information content of the model by controlling their precision.

2.4.1.2.1 Whole-bit coding

Whole-bit coding assigns an integer number of bits bigger than 0 to each coding event so the codes assigned to each input symbol are independent and disjoint from each other. This technique is also called prefix-free coding because a valid codeword can never be the prefix of other valid codeword. This means that the coder immediately knows when all the bits corresponding to a codeword have been received and therefore knows where the next codeword starts. If the prefix-free property is not respected the code can not be decoded without errors. Uniform binary coding (UBC) where each symbol in the alphabet is assigned a codeword length $\log_2(\text{alphabet_size})$ bits is the trivial form of prefix-free coding. UBC can not obtain compression in a statistical method because it assumes that all the symbols have the same invariable probability of occurrence $p=1/\text{alphabet_size}$. UBC is useful in dictionary-based methods when it is used as a dictionary address and the dictionary data width is larger than $\log_2(\text{dictionary_length})$. The prefix-free property considerably simplifies coding and decoding operations and enables fast parallel implementations.

2.4.1.2.1.1 Shannon-Fano coding

Shannon-Fano Coding is considered to be the first well-known modern method for efficiently coding a group of symbols [Shannon48]. It uses the probability of each symbol to assign more bits to symbols with low probability and fewer bits to symbols with high probability. The

construction method, however, can not guarantee producing a whole-bit optimal code and 3 years after its invention it was quickly superseded by the more efficient Huffman codes.

2.4.1.2.1.2 Huffman coding

Huffman coding was presented in [Huffman51] and since then it has enjoyed a widespread popularity. It is a whole-bit optimal code meaning that it can never be improved on by other whole-bit coder. Although its performance in many cases is close to that of Shannon-Fano coding it can never be worse and it is usually better.

To construct a Huffman code for an alphabet formed by n symbols we need to build a tree knowing the probability distribution of these n symbols in our data source. Firstly, we list these symbols in decreasing (or increasing) order of probability forming the leaves of our future Huffman tree. Secondly, we repeatedly select the 2 leaves with smallest probabilities forming a sub-tree whose probability is the sum of the 2 leaves. Finally, we continue this process with the sub-trees until only one tree remains. The Huffman code for a symbol n_i is obtained traversing the tree from the root to the leaf assigned to that symbol and adding a bit 1 or 0 to the code depending if we go left or right at every branch of the tree. A Huffman tree for an alphabet of 6 symbols is illustrated in Figure 2.3. The tree is constructed using the symbol probability distribution $P = \{2/41, 3/41, 5/41, 7/41, 11/41, 13/41\}$ for our alphabet $r = \{f, e, d, c, b, a\}$. For example to code the message 'aaba' the output of the Huffman coder would be '00000100'. Since our example alphabet has 6 symbols a uniform binary code would need at least 3 bits per symbol. Then a total of 12 bits would be needed to code the 4 symbols. The output of the Huffman coder is 8 bits so we have a reduction of 4 bits.

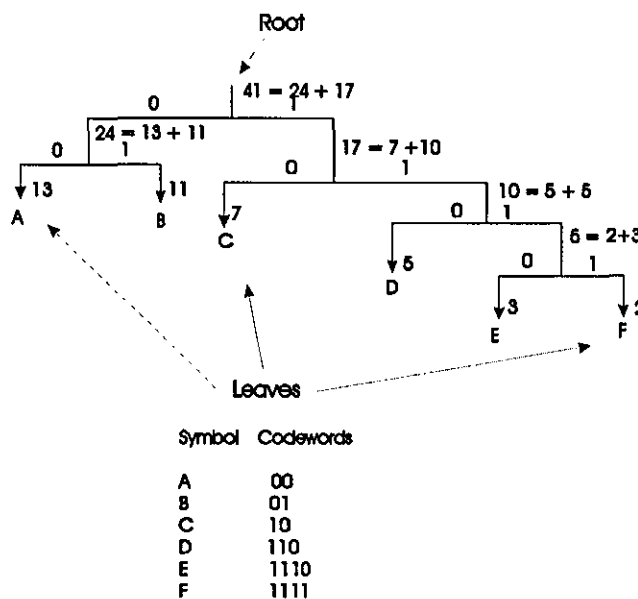


Figure 2.3. Huffman Tree Example

Following this procedure and using a fixed model a fixed Huffman tree is quite trivial to construct. More challenging is the adaptive model case when we want to dynamically adapt the coding tree to changes in the model induced by variations in the statistical properties of the input data source. Small variations in the model could force important changes in the tree structure resulting in a very time-consuming process of reconstructing the tree after each input symbol. Dynamic Huffman Coding has been subject of study in [Vitter87], [Knuth85] where the tree updating procedure is done by traversing the tree from the leaf to the root in constant time proportional to the encoding length. These methods require in the order of $n+r+H$ time to encode a file of n symbols with an alphabet of size r . H is the number of bits produced. This means that if H is much bigger than n and H is much bigger than r then $n+r+H \cong H$ and every bit is output in 1 cycle so in each coding cycle is possible to obtain one bit of output including the updating process of the tree. This measure of throughput depends on the number of compressed bits produced and therefore on the instantaneous compression ratio. This is an undesirable characteristic because it is not possible to guarantee a constant data rate in the uncompressed port.

Huffman coding is an optimal-code when the probabilities produced by the model for n symbols are given by $p(n_i)=1/2^x$ where x is an integer number bigger than 0. In this case the minimum possible number of bits to code a symbol n_i is $-\log_2(1/2^x) = x$. This quantity is an integer number bigger than zero that a Huffman code can output. The problem arises when a good model produces probabilities for a symbol close to 1 that would need a fraction of a bit to be coded (x is closed to 0). A Huffman code must output at least 1 bit and always an integer number of bits as its codeword. The coder in this situation outputs redundancy with the worst case being of 1 extra bit per symbol.

2.4.1.2.1.3 Golomb, Rice, Elias, Fiala Coding

Golomb, Rice, Elias and Fiala coding can be considered variations in the Huffman coding theme since it is possible to construct a Huffman tree for them. They offer less compression than Huffman codes but their simplicity and speed makes them attractive as an alternative to uniform binary coding. Golomb codes [Golomb66] are built by arranging the symbols of the alphabet in descending probability order and assigning positive integers to them. Golomb codes are based on the use of a coding parameter m that changes the shape of the code. Smaller values of m should be used for more skewed probability distributions because very few bits are assigned to more probable symbol but many more to less probable symbols. To encode an integer n using the Golomb code with parameter m we obtain n/m and output this as an integer unary code. Then we obtain $n \bmod m$ and output this value using a binary code.

For example if $n = 10$ and $m = 4$ then $n/m = 2 = '11'$ and $n \bmod m = 2 = '010'$ so the code is $'11010'$. We could have not used $'10'$ to represent 2 because then the resulting code $'1110'$ would have not respected the prefix free property. This means that the binary code needs to be adjusted to avoid extending the unary code. Rice coding [Rice83] is a subset of Golomb coding because only parameters m that are power of 2 are allowed ($m=2^k$). Rice coding is specially suitable for hardware implementation because $n/(2^k)$ can be calculated by shifting and $n \bmod 2^k$ by setting to 0 all the bits in n but the less significant k bits. The following Table 2.1 hows an example of Golomb and Rice codes.

Elias codes [Elias75] are similar to Golomb and Rice codes but they do not use a parameter m so they offer little flexibility and compression performance is limited. Elias describes 2 codes γ and δ . In code γ an integer n is coded as a unary code for $1+\log_2(n)$ bits followed by a code of $\log_2(n)$ in length coding $n-2^{\log_2(n)}$ in binary. The δ replaces the suffix unary code by a γ code.

The Fiala codes [Fiala89] are known as $[start, step, stop]$ codes because they use these 3 parameters to construct many different possible codes. Symbol n is coded as n 1's followed by a 0 and then followed by a field of size $start+n*step$. If this value is equal to the stop value then the preceeding 0 can be omitted. The example in Figure 2.4 corresponds to a Fiala code with the following parameters $[0,1,5]$.

Position	Rice Parameter	K=0	K=1		K=2	γ Elias code	[0,1,5] Fiala code
	Golomb Parameter	M=1	M=2	M=3	M=4		
0		0	00	00	000	10	0
1		10	01	010	001	110	100
2		110	100	011	010	1111	101
3		1110	101	100	011	11100	11000
4		11110	1100	1010	1000	11101	11001

Table 2.1. Prefix-Free codes example

These codes need the symbols in the alphabet to be organised in decreasing order of probability so fewer bits are assigned to the most probable symbol.

2.4.1.2.1 Fractional-bit coding

Fractional bit coders have the ability of mapping a symbol to a fraction of a bit. This means that if the probability of a symbol is close to 1 very little output is needed to code this symbol.

On the other hand a whole-bit coder needs at least 1 bit and always an integer number of them. Fractional-bit coders are usually called arithmetic coders [Witten87], [Langdon84] because they required arithmetic operations such as divisions and multiplications to code a symbol. Arithmetic coders produced a single and unique codeword for the whole message being processed and therefore lack the direct correspondence between bits in the codeword and symbols in the message. They are not prefix-free codes like the whole-bit coders described in the previous section. They are optimal in the sense that they produce an output as close to the entropy of the model as desired by controlling their precision. This optimality comes with the price of higher complexity. The dependencies that appear between the coding/decoding of a symbol and the coding/decoding of next symbol make a parallel implementation a difficult problem. Fast approximations to arithmetic coders using low precision multiplication-free arithmetic speed-up the process at the expense of compression. Current research aims to solve the problem with the lack of parallel execution of the coding and decoding processes.

2.4.1.2.2.1 Full-precision arithmetic coding

Full precision arithmetic coding replaces a stream of input symbols with a single output number less than 1 and greater than or equal to 0 using exact precision multiplications and divisions. A general encoding algorithm to accomplish this follows :

Set low_old to 0.0

Set high_old to 1.0

While there are still input symbols do

Get input symbol

range_old = high_old - low_old

*high_new = low_old + range_old * Pcum_i*

*low_new = low_old + range_old * Pcum_{i-1}*

End of While

Output low

The next graphical example of Figure 2.4 shows the result of processing the message 'aaba' with an alphabet $r = \{f, e, d, c, b, a\}$ using the same probability distribution $P = \{2/41, 3/41, 5/41, 7/41, 11/41, 13/41\} = \{0.05, 0.07, 0.122, 0.170, 0.261, 0.317\}$ as in Figure 2.3 for the Huffman coder. The $Pcum$ are the cumulative probabilities of the symbols $Pcum = \{0.05, 0.13, 0.252, 0.422, 0.683, 0.999\}$. The example shows how the subinterval $[0,1)$ is subdivided in sections proportional to the probability of the symbol that they represent.

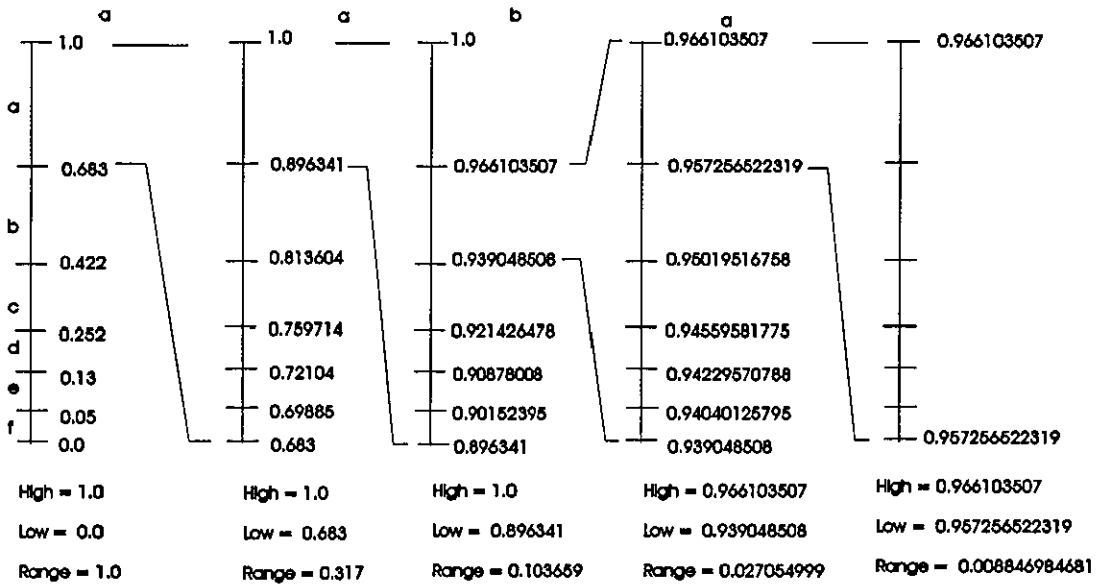


Figure 2.4. Arithmetic coding example

Then we can use any value between the last high and low to represent the string. If we chose 0.96 then we can represent the string in 7 bits obtaining 1 bit reduction if comparing with the Huffman code.

To decode the compressed stream we use an algorithm as follows:

Get encoded number

Do

Find symbol_i whose range straddles encoded number

Output the symbol

range = Pcum_i - Pcum_{i-1}

Subtract Pcum_{i-1} from encoded number

Divide encoded number by range

Until no more symbols

In the previous example the encoded number is 0.96 so we know that first symbol to be output is 'a' then we subtract symbol low value from encoded number to obtain 0.277. Then we divide by the range 0.317 and the resulting value is 0.873817. Then we know that the second symbol is another 'a'. We continue by subtracting symbol low value from encoded number to obtain 0.190817. Then we divide by the range 0.317. We now obtain 0.601946 so the next symbol is 'b'. The process continues until no more symbols are left to decode. To detect when to stop either a special termination character can be encoded (not done in this example)

or the length of the uncompressed string can be concatenated to the compressed message. If the uncompressed length is large enough the overhead is small.

The previous example shows a practical problem with arithmetic coding related to precision. If the message continues for a few more symbols we would have run out of bits to hold the required precision. It also seems to show that the whole message needs to be processed before the compressed codeword is known. These problems have been solved in practical implementations so only integer arithmetic is required and incremental transmission is possible.

A practical arithmetic coder is reported in [Witten87]. His implementation outputs a bit of codeword as soon as it is known and replaces the floating point arithmetic for integer arithmetic so the interval $[0,1)$ is replaced by $[0,N)$ with N being as large as 65536. The cumulative probabilities provided by the model are also stored using integer numbers in form of cumulative frequency counts so $Pcum$ of example 2.4 becomes $Fcum = \{ 2, 5, 10, 17, 28, 41 \}$ so the $Pcum_i$ is obtained dividing the $Fcum_i$ by $Fcum_n$ where n is the last symbol that stores the total, in the example symbol 'a' with $Fcum = 41$. When the low and high values are close together some more significant bits are equal. These bits are added to the output and then the interval is scaled up so it keeps large enough to assign some range to all the possible input symbols. This is necessary to avoid having underflow conditions. When low and high straddle 0.5 the next bit output is not known but a follow-on procedure is used to keep track of the number of cycles the mechanism is used. Operation continues until the interval falls above or below 0.5. If the interval is above 0.5 then a 1 is output together with a number of 0's as indicated by the follow-on mechanism. If the interval is below 0.5 a 0 is output together with a number of 1's as indicated by the follow-on mechanism. Other similar mechanism for incremental transmission and fixed precision arithmetic have been developed by [Guazzo80]. The IBM bit stuffing idea of [Pennebaker88] that consists in inserting zeros to block carry propagation fulfils the same function as the follow-on procedure described above.

2.4.1.2.2.2 Low-precision arithmetic coding

Low precision arithmetic coding aims to replace the slow multiplications and in some cases divisions necessary to implement the full precision algorithm for some simpler alternative. It comes in 2 main fashions. Techniques that replace the slow multiplications by shifts and adds and techniques that perform all the calculations ahead of time and store the results in look-up tables.

Quasi-arithmetic coding is based on performing all the calculations ahead of time and it is described using a binary alphabet in [Howard93b]. The number N in the interval $[0, N)$ generates a number of possible states in the coder that equals $3^N/16$. If N is small the number of states is small and it is possible to precompute all the possible state transitions and outputs and stored them in a table. If $N=4$ then the number of states is 3. Quasi-arithmetic coding shows that if compared with an exact arithmetic coder the number of extra bits output per input symbol is at most $5.771/N$. This means that a larger value of N improves the efficiency but also increases the complexity. In practical terms values between 32 and 128 are used. The proposed way to extend the binary coder to a multi-alphabet coder is to assign the symbols of the alphabet to the leaves of a binary tree. Then the coding of a symbol is decomposed in the coding of a binary decision at each level of the tree. A binary Quasi-arithmetic coder can be used in each level of the tree.

The method proposed in [Rissanen89] and used in the Q-coder [Pennebaker88] simplifies the multiplication and divisions operations by scaling the range and the total count of the model to the same interval $[0.75, 1.5)$. The implementation replaces storing the *high_new* value by storing the *range_new* so the original equations:

$$\begin{aligned} high_new &= low_old + range_old * Pcum_i & [2.5] \\ low_new &= low_old + range_old * Pcum_{i-1} \end{aligned}$$

become:

$$\begin{aligned} range_new &= range_old * (Pcum_i - Pcum_{i-1}) & [2.6] \\ low_new &= low_old + range_old * Pcum_{i-1} \end{aligned}$$

The algorithm then makes the approximation $range/Fcum_n \cong 1$ and since $Pcum_i - Pcum_{i-1} = (Fcum_i - Fcum_{i-1})/Fcum_n$ equation set [2.6] is simplified to:

$$\begin{aligned} range_new &= Fcum_i - Fcum_{i-1} & [2.7] \\ low_new &= low_old + Fcum_{i-1} \end{aligned}$$

Multiplications and divisions are not longer present in equation set [2.7].

The analysis in [Lei95] shows that the error of the Rissanen method depends on the value $(Fcum_n - Fcum_{n-1})/Fcum_n$. The error is larger when this value is smaller so the most probable symbol is placed in the last position to force $Fcum_n - Fcum_{n-1}$ to have a large value. The conclusion is that the degradation of the method is significant when the count of the most probable symbol is small. An extension of the method is proposed so the approximation

$range/Fcum_n \cong 1$ after scaling both quantities to the interval [1,2) is replaced by a number $b = \{0.5, 0.625, 0.75, 0.875, 1, 1.25, 1.5, 1.75, 2\} = \{0.100, 0.101, 0.110, 0.111, 1.000, 1.010, 1.100, 1.110, 1.111\}$. The multiplication times b can be done shifting and adding with the following values of $b = \{2^2, 2^2+2^0, 2^2+2^1, 2^3-2^0, 2^3, 2^3+2^1, 2^3+2^2, 2^4-2^1, 2^4-2^0\}$. This is particularly well suited to fast software and hardware implementations. The simplified equations are now:

$$\begin{aligned} range_new &= b*(Fcum_i - Fcum_{i-1}) & [2.8] \\ low_new &= low_old + b*Fcum_{i-1} \end{aligned}$$

This better approximation improves the method and the results show a degradation of 1.04% compared with a full precision implementation while Rissanen method increases the degradation up to 6.06%.

2.4.2 Dictionary-based methods

2.4.2.1 Dictionary-based modelling

Dictionary-based modelling is a concept easier to understand than statistical modelling. The model stores a collection of symbols expected in the input data source in the form of a dictionary. It then tries to replace occurrences of these symbols in the data being processed by indexes to the dictionary locations where the same data can be found. These methods try to group symbols together and replace them by a single index to improve compression. They are string oriented and not symbol oriented like the previous statistical methods. As long as the index size is smaller than the string size compression is obtained. The larger the dictionary size the higher the chance of finding the input symbol in it but also more bits are needed for the index.

Most of the dictionary modelling techniques have their roots in the work published in 1977 [Ziv77] and 1978 [Ziv78] by J. Ziv and A. Lempel. The LZ77 (LZ1) and LZ78 (LZ2) vary in the way the dictionary is built and maintained and how the indexes referenced the information stored in the model. They emerged as valid alternatives to classical statistical Huffman methods and generated plenty of research and variants on the LZ theme.

2.4.2.1.1 Lempel-Ziv 77 (LZ77, LZ1) modelling

An LZ77 dictionary is based on dynamically keeping a window of symbols seen previously in the input data source with a typical window length that varies between 512 and 16K bytes. The window slides over the data maintaining strings of symbols together forming phrases. A buffer is concatenated to the dictionary window that contains symbols that have not been processed yet. The buffer size is typically between 16 and 64 bytes. The buffer contents are compared against the dictionary contents to find the longest matching string. LZ77 compression is based on outputting three items: an index to the dictionary indicating where the match started, an offset indicating the length of the match and finally the first character in the input that did not find a match in the window. Figure 2.5 shows an example of how LZ77 works.

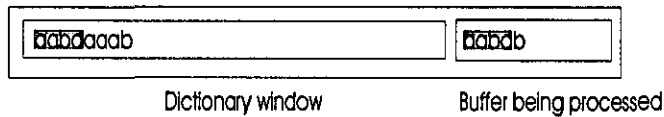


Figure 2.5. LZ77 Example

The dictionary matches the first 4 symbols of the buffer starting at position 0. The output of the dictionary model is $(0,4,b)$. Symbol b is the first symbol in the buffer that it is not found in the dictionary. This method presents 2 major inefficiencies. Firstly, if no match is found the output of the model is still 3 items. For example if symbol y does not exist in the dictionary the output is $(0,0,y)$. It is common to have many misses during the initial stages of compression so this effect would degrade compression significantly. Secondly, it always requires an extra character to be added to the output. This could be quite inefficient if this character could be made part of the next compressed token instead of explicitly adding it to the output.

These inefficiencies were dealt with in the LZSS implementation [Storer82]. This algorithm uses a single bit concatenated to every output token indicating a hit or a miss. If a miss the non-matching character is appended to this single match bit. If a hit the match location and match lengths are appended to the match bit. In this algorithm the output of the model is also the output of the coder since uniform binary coding (UBC) is used directly. LZS is a very popular hardware implementation similar to the LZSS version. LZS adds more complex coding techniques for the index and the match length to improve compression. We will discuss LZS in the hardware section.

Decoding is simpler than coding since time consuming searching is not necessary and the data provided in the compressed input can be used directly to fetch the uncompressed output from the dictionary.

The fast execution and simplicity of the LZ77 algorithm together with the good compression ratios obtained by their improved derivatives have made the LZ77 algorithm to become the most successful general application lossless data compressor. Popular software implementations such as PkZIP from PKWARE and ARJ from ARJ Software illustrate this fact.

2.4.2.1.2 Lempel-Ziv 78 (LZ78, LZ2) modelling

LZ78 creates dictionary entries formed by complete phrases by concatenating the first unmatched symbol to the previously matched phrase. The dictionary initial state is formed by only 1 phrase that is the empty string. The first symbol being processed is replaced by a pair formed by a reference to the empty string plus the symbol in explicit form. Then the symbol is added to the dictionary forming a new phrase. The output of the encoder is always formed by a reference to the longest matching string in the dictionary plus the symbol that stopped the match. This symbol is always added to the previously matched string forming a new phrase that it is then added to the dictionary. Figure 2.6 shows the dictionary state after processing 'aabaaaab' and then string 'aaba' is received. The longest matching string is 'aab' at location 4 and the new phrase 'aaba' is added at location 5. The output of the LZ78 algorithm is (4,a).

Dictionary	
Location	Contents
0	Empty
1	a
2	ab
3	aa
4	aab
5	

Input Data : aaba

Encoded Output : 4,a

Phrase number 5 : aaba

Figure 2.6. LZ78 example

The data structure that stores the dictionary can grow unboundedly and some means of controlling its size must be implemented to avoid using too many memory resources. When a predefined maximum dictionary size is reached, the dictionary can be frozen so adaptation stops or it can be reinitialised to an initial or an intermediate state to improve compression efficiency. Some other policies can also be used such as LRU (Least Recently Used) eliminating the dictionary entry that has not been used for the longest time.

A popular derivative of the LZ78 algorithm is the LZW [Welch84] variant that avoids the need to explicitly transmit the non-matching character by starting with an initial dictionary state where all the possible input symbols have already been included. An LZW derivative named LZC was implemented in the Unix utility *'compress'* with extra tuning of the coding process to improve compression performance. In this case the match location is not coded as a simple uniform binary code but as phased binary code to avoid adding extra bits to the output when only a few dictionary locations are valid.

2.4.2.1.3 BSTW modelling

The BSTW algorithm [Bentley86] follows a different approach to dictionary modelling when it is compared with LZ models. In general, LZ models try to assign a fixed-length code to a variable-length group of input symbols. On the other hand BSTW modelling tries to assign a variable-length code to a single input symbol defined as a word where the term word has a predefined meaning. [Bentley86] implementation defines a word as the longest sequences of alphanumeric and non-alphanumeric characters but other definitions are possible. BSTW keeps 2 distinct dictionaries for the 2 independent word streams maintained using a move-to-front (MTF) strategy. The MTF forces more popular words to appear closer to the top of the dictionary and this feature can be exploited to use fewer bits to code them. A prefix-free code such as a Huffman code can be used to achieve this effect with locations closer to the top of the dictionary being also closer to the root of the Huffman tree. New words are always added to the top of the dictionary and the oldest word located at the bottom of the dictionary is removed when it becomes full. The experimental results show that the larger the dictionary the better the compression. They also show that this simple single-pass MTF dictionary maintenance strategy plus fixed Huffman coding offers a similar performance to a two-pass Huffman scheme where the first pass is used to construct the Huffman tree and the second pass to produce compression.

Figure 2.7 shows an example of BSTW coding and adaptation.

Dictionary at time t		Dictionary at time $t+1$	
Location	Contents	Location	Contents
0	The	0	car
1	car	1	The
2	in	2	in
3	the	3	the
4	shop	4	shop
5		5	

Input Data at time t : car
 Encoded Output at time t : 1

Figure 2.7. BSTW example.

The MTF strategy is highly suitable for hardware implementation because the serial process of modifying the dictionary in software can be done in 1 cycle in hardware using a highly regular array of dictionary elements.

2.4.2.2 Dictionary-based coding

The function of the dictionary-based coder is to replace the uniform binary indexes produced by a dictionary model for other more efficient form of coding and therefore enhance compression. This form of coding tends to be much simpler than statistical coding because it does not handle probability information. Dictionary-based coding is in many cases trivial because the uniform binary codes that form the output of the model are used directly as the output of the system after being assembled in the bit packer. Uniform binary coding assigns a binary code of length $\log_2(\text{dictionary size})$ bits to each dictionary location and its decoding is trivial.

It is also frequent to use a statistical coder to code the output of a dictionary model creating a new form of hybrid. The idea is that the output of the dictionary model has biased statistical properties and some dictionary references are more frequent than others. This feature can be exploited by an statistical coder working as a back-end such as arithmetic or a Huffman coder to further process the output of the dictionary model and enhance compression [Moffat94].

Dictionary-based coders can be based on techniques like phase binary coding (PBC) or run length coding (RLC). Phased binary coding outputs a code whose length is dependent on how

many entries are valid in the dictionary and the dictionary grows by a single location at a time. Run length coding groups repetitive sequences of indexes output by the model. Other popular form of dictionary-based coding is to use a dictionary that grows in powers of 2. Then a uniform binary code can be adjusted to use only those bits needed to code the active section of the dictionary. Since misses prove to be very popular outputs from the model it is common to use a single bit prefix to the code to make a distinction between a match and a miss.

2.4.2.2.1 Phased Binary Coding

Phased binary coding is useful when not all the locations in the dictionary require a codeword to be assigned. This is a typical situation when the initial dictionary state is in an empty state and entries are added to the dictionary simultaneously to the input data being processed. In this case a more compact set of codewords can be used saving bits in the output. The basic phased binary coding algorithm follows:

If ($I < MAX - VALID$)

Code I using a binary code of $\lceil \log_2(VALID) \rceil - 1$ bits;

else

Code $I + MAX - VALID$ using a binary code of $\lceil \log_2(VALID) \rceil$ bits;

Where MAX is $2^{\lceil \log_2(VALID) \rceil}$ and $VALID$ is the number of dictionary locations valid in a particular instant. For example if $MAX = 128$ and $VALID = 127$ then if the dictionary location to be coded is $I < 128 - 127 = 1$ (location 0) only 6 bits are needed whilst 7 bits are needed for the rest of the locations. Phased binary coding tends to assign fewer bits to locations closer to location 0 so it is useful that the maintenance of the dictionary makes these locations more probable than those closer to the bottom. When the dictionary is full (in our example $VALID = 128$) there is no difference between using a phased binary code or a uniform binary code.

2.4.2.2.2 Run Length Coding

Run length coding is a simpler coding technique based on replacing repetitive sequences of the same symbol with a pair formed by a code indicating the repeating symbol plus a code indicating the length or number of repetitions that were seen. This method of coding is of limited usefulness for general compression. It can achieved, however, good results in some specific types of data where long runs of the same symbol are common such as repetitions of 0's in memory pages or fax pages. If the input to a simple run length coder is the string 'aaba'

the output will be $(a, 2, b, 1, a, 1)$. This example shows that the input to the run length coder is directly the input data so in this case no modelling is being performed on the data. It is also perfectly valid to include a dictionary-based model so the output of the model and input to the run length coder is a stream of dictionary location indexes. If the same dictionary location is reference more than twice effective run length coding can take place.

2.4.3 Other methods

The BWT (Burrows-Wheeler Transform) block-sorting algorithm described in [Nelson96] deserves special mention. This is a new modelling method based on a transformation function that converts a block of data using a sorting algorithm into a new block of data extremely well suited for data compression. The new block has exactly the same elements as the original block but the new organisation shows clumps of identical symbols grouped together. The transformation is reversible so the original block can be recovered. Compression is obtained by exploiting the increase in redundancy generated by the sorting algorithm using typically a Huffman coder or arithmetic coder preceded by a run length coder. The BWT algorithm combined with standard coding techniques produces a compression that rivals with that obtained by the finite-context modelling methods of section 2.4.1.1.1. The transformation function consists simply in shifting and sorting the input block of data so no complex arithmetic is involved. The main drawback of the method is that it needs to operate in a whole block of data simultaneously and therefore it does not support incremental reception or transmission. The blocks of data must be of at least 250 Kbytes to give good results. If the block sizes are reduced to a more manageable value of 4 Kbytes the sorting algorithm is inefficient and the extra bits of overhead needed in each block to ensure that block-un-sorting can be done degrade compression. In general the BWT modelling technique can be used as a front-end of a general compressor since it will improve the performance of the existing compressor by increasing the redundancy of the input data.

Other approach to efficient modelling of an input data source is the neural networks presented in [Jiang96b]. Two neural networks are described to perform lossless data compression.

The first one uses a single-layer of processing elements and it achieves a compression performance of 0.7. Each neuron in the system stores a data element fixed in length that corresponds to a possible input string. The input string is compared with this value and the output of the neuron goes to 0 if a match is found. A coding technique based on a Huffman code derivative such as those described in section 2.4.1.2.1.3 is used to assign fewer bits to neurons that are more successful in finding matches. The neurons are assigned an index that

increases from left to right and the network is maintain shifting neurons left after each input data is processed using the following method. Successful neurons are promoted to right side of the neural network. New input data is always added in this right side. Unsuccessful neurons are discarded when the reach the left side. The compression performance of the algorithm is low because matches have to be in full and partial matches that happen when part of the input string matches in one neuron and the other part in another neuron are not allowed.

To solve this problem a second example is developed that extends the initial model by adding a second layer of processing elements and improves compression typically to a value of 0.4. The second layer of neurons is design based on a 2-byte input string to detect partial matches of the MSB in one neuron and LSB in another neuron. It also extends the initial technique by coding runs of matches in multiple neurons using a single code. Although the technique has potential for a hardware implementation to exploit the massive parallelism present in neural networks this possibility is only indicated in the paper and no details are given on a hardware implementation. It is possible to identify the typical features of a high performance hardware system such as doing all the comparisons in a single cycle and using a simple adaptation mechanism.

2.5 Lossless Data Compressor Hardware

The same classification method as in software can be applied to the lossless data compressor hardware world with a separation between statistical hardware and dictionary-based hardware. There is an even clearer domination of dictionary-based methods over statistical methods in hardware. The reason is that statistical methods can not currently compete in speed and although the compression performance is theoretically superior this can be only be achieved with very high complexity. This complexity again degrades speed and makes them unfeasible for many applications. Although some implementations have been very successful, such as those based on binary alphabets from IBM, the simplifications that made them possible have limit their application to systems with low throughput requirements in the order of a few Mbits/s.

2.5.1 Statistical hardware

Statistical hardware is limited to simple 0th order modelling using multi-symbol alphabets that limits compression or simple high-order modelling using binary alphabets that limits speed. Coding is usually done with Huffman or arithmetic coding the later being preferred because of its compression efficiency.

2.5.1.1 Binary arithmetic Hardware

The Q-coder [Pennebaker88], [Arps88] from IBM is one of the best known examples in this category. It consists of a 7th order binary finite-context statistical model associated to a corresponding binary arithmetic coder. It is important to notice that this is a fixed-order model and not a variable-order model such as the PPM method of section 2.4.1.1. This means that always the same number of symbols (same context length) are used to predict a new symbol and a prediction can never fail. A variable-order model does not apply to a binary alphabet since a single probability value p defines both symbols 0 (p) and 1 ($1-p$) and predictions are always possible (no escaping). On the other hand PPM blends different orders together so if a prediction fails in a particular order the next lower order is used. No variable order models have been reported in hardware using either binary or multi-symbol alphabets to the best of our knowledge.

In the coding section the binary arithmetic coder uses the renormalization approximation introduced by Rissanen in [Rissanen89] and discussed in section 2.4.1.2.2.2 to avoid the complex multiplications. The range is divided between the 2 symbols LPS (Least Probable Symbol) and MPS (Most Probable Symbol). LPS is assigned to the lower part of the range A and MPS to the upper part of the range A . Renormalizations are used to expand the interval range A and to keep it large enough to accommodate both symbols using fixed precision arithmetic. Every renormalization produces an output bit. A is renormalized between 0.75 and 1.5 and approximated to 1 so no multiplications are needed as seen in section 2.4.1.2.2.2. In the modelling section the probability estimation process is adaptive and based on a state machine with 60 states k . A more probable symbol 1 uses 30 states and a more probable symbol 0 uses another 30 states. Each state k has associated a less probable symbol probability estimate $Qe(k)$ that would be used by the coder. When a LPS renormalization takes place the probability estimate Qe is increased since a LPS was just coded. After a MPS renormalization the estimate Qe is decreased what corresponds with an increase of the estimate of the MPS ($1-Qe$). This means that the renormalization process associated to the arithmetic coder and the state machine with different Qe values associated to the model are used to replace the explicit symbol counting mechanism used in other methods. An index value identifying a state and pointing to a Qe table position is kept for each different context whilst a single table stores all the probability estimate values Qe . Good compression results are presented using a 7th order model with 128 contexts based on the 7 neighbouring pels (bits) for facsimile compression. The Q-coder algorithm is also known as Adaptive Bilevel Image Compressor (ABIC). The simple multiplication-free arithmetic coder and simple dynamic probability adaptation enables a fast hardware implementation with high clock rates.

On the other hand the Q-coder has a symbol granularity of a single bit what means that at most 1 bit can be processed per clock cycle. This is a maximum performance throughput that is usually degraded by the characteristics of the data source. Experimentation [Kampf98] shows a worst case performance of 0.8 bit per clock cycle. The reason is that the arithmetic coding process can require multiple renormalizations (interval range expansions) per bit so throughput is not data independent. Moreover, the probability Q_e table is derived from processing black and white (bi-level) image files what should limit the compression efficiency if using the Q-coder as a general-purpose compressor. The QM-coder is a variation of the Q-coder used in the Joint Bi-level Image Experts Group (JBIG) compression algorithm [Arps88]. The QM-coder uses a different software optimised convention and allocates the MPS to the lower part of the range and LPS to the upper part of the range. The context in the QM-coder is formed by 10 bits generating 1024 different contexts. The worst case throughput performance is 0.73 bits per clock cycle. A recent VLSI implementation of the QM-coder and Q-coder has been done in [Slattery98a]. The device called the Qx-coder can implement both algorithms and clocks at 75 MHz with a throughput of approximately 64 Mbits/s using a CMOS 5S (0.35 μm) technology from IBM [Marks98].

[Jiang96a] describes a parallel binary arithmetic coder derived from the IBM Q-coder. The parallel implementation processes 4-bits in parallel using a tree of processing elements where each processing element corresponds to a modified Q-coder. The number of levels in the tree corresponds to the number of input bits being processed in parallel and in principle it only affects the latency. The equations that the processing elements have to implement are more complex than the Q-coder and include multiplications so the appealing multiplication-free feature of the Q-coder is lost. The author suggests performing the multiplications using a hierarchical structure of adders in each processing element to affect only latency and not speed but this should have a negative impact in complexity. Adaptation is also modified since it only happens every 4 input bits and not after every bit like in the Q-coder. This should have an impact on compression efficiency although the reported results show minimum differences in this aspect. Parallel decoding is possible because there are only $2^4 = 16$ possible input combinations. The pointer code can be used to directly perform a direct comparison with 16 values corresponding to all the possible combinations and the correct string can be found in parallel. The paper only gives compression results based on a C language implementation of the algorithm while hardware details are minimal.

[Kuang98] presents a 10^{th} order finite-context statistical model with associated binary arithmetic coder. A variant from the bit stuffing technique of IBM is presented to solve both the carry-over problems and the termination condition. A couple of bits are set to 0 after

receiving 16 consecutive bits set to 1. If there is a carry-over the second bit always blocks the propagation while the first bit is used to signal termination when set to 1. If the decoder receives 17 consecutive bits set to 1 then it knows operation must be terminated. The overhead of the method is 17 bits only at the end of the process plus the stuffing bits that are only needed in very few occasions. The other way to solve the termination condition is by adding an extra end of file (EOF) character that it is only coded once. This has the disadvantage that a third symbol is introduced in the system increasing the algorithm complexity. It is also possible to concatenate to the compressed block the original uncompressed size so the decoder stops once this uncompressed size is reached. This is the simplest solution although there could be a problem if the uncompressed size is not known at the beginning of the compression operation and incremental transmission is required.

The adaptive modelling unit is based on calculating a range of possible probabilities for symbol 0 and stored them in a table named $Prob_0$. Then other table Ad with 1024 entries corresponding to all the possible contexts generated in a 10th order model is used to address this table $Prob_0$ and obtain a conditional probability P_0 for the arithmetic coder. After receiving an input symbol the adaptation mechanism uses 2 offset tables to obtain a new address to the $Prob_0$ table. This address is stored in table Ad so the probability associated to the active context changes. This address will point to a position in the $Prob_0$ table with a higher value of P_0 if a 0 was received or the other way around otherwise. A simplified multiplier is used to perform the $P_0 * A$ operation where A is the range. The technique reduces the hardware complexity in half by discarding the half least significant bits result of the multiplication. It is important to notice that arithmetic coding can also involve a division to obtain a probability value if frequency counts and not probabilities are used in the model. This method like the Q-coder deals directly with pre-calculated probabilities stored in tables obviating the need for this operation. Again the presence of the renormalization loop in the arithmetic coder makes throughput data dependent.

The simulation results presented in the paper suggest that on average the renormalization loop uses only 0.5 cycles because in many cases is not needed. Adding this value to 8 cycles fixed time to for the rest of the chip operation produces an average value of 8.5 cycles to process a bit of input data. The clock rate is 25 MHz (using a 0.8 μm single-poly double-metal (SPDM) technology) and therefore the throughput is 25/8.5 approximately 3 Mbit/s. The renormalization loop could, however, take up to 7 cycles to complete. Then it could take up to 15 cycles to process a bit of input data for a worst case throughput of 1.67 Mbit/s. The compression ratio based on the experimental results shown in the paper using a combination of text, image and binary files seems to be in the order of 0.5.

The same research group presents a variation on the previous device in [Jou99]. A finite-context 10^{th} order statistical model associated to a binary arithmetic coder forms the device. The coder is essentially the same as in [Kuang98] but an additional tuning step is included in the model to improve compression efficiency. The model uses the same concept of having 2 tables: the first table is used to store the most probable 128 conditional probabilities for symbol 0 called $Prob_0$ that eliminates the need for a division, the second table Ad is used to store 1024 addresses to this $Prob_0$ table to reflect that the probability of the symbol 0 changes depending on which context is active. Offsets tables are used to calculate the new probability of symbol 0 after completing the current coding step. A probability-tuning step is introduced in this stage so depending on the characteristics of the data source being compressed a different pointer to $Prob_0$ is stored in Ad . A total of 5 different tuning steps are pre-calculated and the results stored in 5 different offsets tables. A fuzzy inference process based on the past behaviour of the processed data is used to select one of these 5 offsets tables to perform the adaptation in the model. The complexity of the model is higher than if it is compared with [Kuang98] but compression improves because the probability tuning step is used to reflect the characteristics of different data sources such as bi-level image data, colour image data, greyscale image data, binary data and text data. Compression results are shown on different data types and consistently outperform a 0^{th} order context-free multi-alphabet model plus arithmetic coder. An average compression ratio slightly better than 0.5 is achieved. The performance of the design is equivalent to the [Kuang98] since the same coder is used and the multiplication is again the limiting factor in speed.

A parallel architecture for arithmetic coding is presented in [Lee96]. The description does not include any references to the modelling stage only to a way of reorganising the basic arithmetic equations to incorporate parallel processing. The algorithm processes a number of N symbols in parallel but not in a single cycle because the dependency between 2 different group of symbols is present. The width of the symbol is not indicated but since the number of arithmetic operations (multiplications and additions) is considerable it will probably fit better a binary alphabet. The basic idea is to obtain the arithmetic equations state for low and range after processing N symbols and then reorganize them to replace N operations for $\log_2(N)$ so a tree-shape parallel architecture can perform them. The processing of N symbols must be completed before the next N symbols can access the architecture. Pipelining is not possible because the resulting state of low and range must be input together with the probability values of the N symbols. Hardware details are minimal.

2.5.1.2 Multi-alphabet arithmetic hardware

[Boo98] describes a 0th order context-free multi-alphabet statistical model associated to an arithmetic coder for coding of multilevel images. The alphabet size is 256 and frequency counts are dynamically maintained in the model for each of the symbols in the alphabet. We have already discussed that arithmetic coding needs cumulative frequency counts to properly identify the range the symbol uses in the $[0,1)$ interval. This means that a worst case in the adaptation process would need to increase all the cumulative frequencies stored in the model resulting in a large number of additions. To alleviate this problem the scheme implemented in the paper stores a subset of 16 cumulative frequencies named reference cumulative probabilities while the rest are stored in normal non-cumulative form. To calculate the cumulative frequency needed by the arithmetic coder for a symbol k it is necessary to use a reference cumulative frequency h plus a few frequency counts $h+1, h+2, \dots, k-1$ if $k < h+8$ or cumulative frequency $h+16$ minus a few frequency counts $h+15, h+14, \dots, k$. The worst case needs 9 additions or subtractions corresponding to 8 symbol probabilities and 1 reference probability. This technique simplifies the adaptation process, since now in the worst case only 16 reference cumulative frequencies need to be updated plus one symbol frequency, but it adds more operations to the calculation of the cumulative frequencies. The arithmetic coding process has been simplified by truncating the multiplier to a small number of most significant bits, which is a trade-off between complexity and compression efficiency. The architecture is evaluated using a $0.7\mu\text{m}$ CMOS standard-cell library [Peon97] and the non-pipelineable critical path is found to have a delay 26 ns in the interval updating formed by the multiplication and normalization process. Therefore the maximum clock frequency is reported in 39 MHz with a total area of 31 mm^2 .

The paper seems to deal with the concept of symbol probability and symbol frequency count indistinctly. However to obtain a symbol probability it is necessary to divide its frequency count by the total frequency count. The total frequency count is usually the cumulative frequency count of the last symbol. This division is usually done as a table look-up using a few more significant bits from both operands as address since divisions are even more undesirable than multiplications. This problem is not addressed properly in the proposed architecture that seems to use symbol frequency counts in the multiplication operation as if there were directly equivalent to symbol probabilities. The paper does not provide any results on compression performance. Compression performance should be limited since 0th order models perform poorly in most situations.

The same technique described in section 2.5.1.1 [Jiang96a] for a binary parallel coder is used in [Jiang94] to obtain a parallel implementation of a multi-alphabet arithmetic coder.

Although the arithmetic coder is associated with a 0th order model to evaluate the performance of the coder, the model implementation is not dealt with in hardware. The system processes 8 bytes at a time instead of 4 bits at a time using a hierarchical tree structure with 5 levels. Parallel decoding in this case is unfeasible because the number of possible input combinations to decode 8 symbols in parallel with an alphabet of 256 symbols is 256^8 . This means that the complexity of the parallel decoding hardware is too high. A sequential decoder is designed to work with the parallel coder. Most applications are read biased, which means that they tend to decompress more often than compress. The lack of parallel execution in the decoding process constitutes a major limitation in the implementation. There are 2 types of processing elements realising 2 different sets of equations in the tree structure. Both sets of equations involved multiplications and divisions. The proposed processing element has 1 level with 6 14-bit multipliers and 6 levels of adders for a total of 20 adders. It is obvious that the complexity of this PE is very high. These levels can be pipelined but since there are at least 5 levels of PE's in the whole coder the resulting latency would be very high degrading speed accordingly. One set of PE's involved the division by a fixed value of the initial range 16384 that can be readily implemented by shifting left the 14 least significant bits. The other equation involves the division by the total cumulative frequency count that always increases with each adaptation cycle and that in this implementation can have a maximum value of 8192. This division does not seem to be dealt with properly in the algorithm description. A typical solution is that, as in other implementations, the cumulative frequency count of the symbol and the total cumulative count are used to address a table and obtain the cumulative probability count resulting from dividing these 2 values. Hardware details are minimal. They are limited to a few block diagrams and not data is reported on complexity. The simulation results, which are based on a software model, seem to suggest that compression performance is not affected if comparing this parallel implementation with a sequential one. It is important to notice, however, that the adaptive model is updated every 8 symbols instead of every symbol as in classical sequential coders. The paper also acknowledges the need for higher order context-based modelling to improve compression. Unfortunately, the complexity of higher-order context-based modelling using multi-symbol alphabets has precluded any examples in hardware up to now.

The work presented by the same author in [Jiang95] is the implementation of a multi-alphabet arithmetic coder using a modification in the basic equations. These modifications consist in not using the values of *high* and *low* to define the state of the coder but instead using the values of *low* and *range* where *range* equals *high* - *low*. The paper claims that this change brings a simplification in the renormalisation process so that it can be controlled only when the value of *range* is less than half the initial range. This single condition test allows the

incremental generation of output bits and range expansion to avoid underflows and to allow coding to continue indefinitely using fixed-length registers. The criticism in [Moffat97] is based on several problems that seem not to be properly addressed in the original paper. A loss of coding efficiency is introduced by rounding errors and also the decoder is more complex since 2 values must be calculated. The paper gives no details on the model that should feed the arithmetic coder although the compression results are based on using a 0th order one. Hardware details are again minimal, limited to a few block diagrams that makes it difficult to draw any conclusions in terms of speed or complexity. The new algorithm equations are design to handle multiplications and divisions and no effort is made on simplifying these operations. These unresolved issues should limit a hardware realisation.

[Hsieh98] describes a multi-alphabet 0th order context-free model associated to a corresponding arithmetic coder for video compression. The modelling unit uses a limited past history model implemented as a first-in first-out buffer used to store a window of symbols from the input data source. This window buffer allows the modelling unit to pick up the local statistics of the data being compressed, thanks to the principle of locality of reference, thus increasing compression efficiency. A small buffer size improves the speed of adaptation but it could damage compression if not enough data is available to construct an accurate model. This limited-past history model overestimates the probability of a symbol by $1/(p+M)$ where p is the alphabet size and M is the buffer size. The overestimation is caused because all the possible input symbols in 0th order modelling must be assigned an initial count higher than 0 to prevent the coding from failing when a symbol is processed for the first time. The total frequency count is $p+M$. If the buffer size M is small this error is larger. To alleviate this problem a weighted limited-past history model is proposed so the overestimation value becomes $1/(p+M*W)$. If the weight W increases, the error decreases although a large value of W will also damage compression because the probability of a symbol not yet seen in the data source (the overestimation) could become very small. The best trade-off seems to be a weight of 16 and a buffer size of 112. The modelling unit also ensures that the denominator of the cumulative frequency is a power of 2 so that the division required to obtain the cumulative probability can be done by shifting. The algorithm uses the multiplication-free solution proposed by Rissanen [Rissanen89] to perform the arithmetic coding itself. The normalisation is done in a single step by counting the number of positions that the range and lower pointer should shift to maintain these values in the correct range [0.75, 1.5) used in Rissanen arithmetic coder [Rissanen89]. This technique avoids a data dependent throughput because it eliminates the need for a variable number of cycles to normalize the range. In the hardware implementation a similar technique to [Boo98] is used to store the frequency model using some frequency counts as base and others as variations from the base. In this case the

variations are not stored as the true frequency counts but as offsets from the base. This technique means that to obtain the true cumulative count of a symbol only 2 values a base and a offset must be added whilst in [Boo98] 9 values could be needed in a worst case. On the other hand the adaptation process must change in a worst case 16 bases and 16 offsets whilst [Boo98] only needs to change 16 bases and one true value. This adaptation process can be done in parallel using 32 counters but it increases hardware complexity.

The limited past history model means that in each cycle one symbol enters the buffer and one symbol leaves the buffer. Then 2 symbols one adding 1 count and other decreasing 1 count must adapt the cumulative frequency array adding cycles to the operation. In the decoding process only 16 comparators are needed to decode a symbol in parallel. This is an important reduction in complexity since a 256 symbol alphabet needs 256 comparators if a bank model is not used. During an initial cycle the pointer is compared with the bases and once the correct bank has been identified a second cycle is used to compare the pointer with the offsets inside the base. The scheme incurs again in a performance penalty since the decoding operation uses 2 cycles instead of 1. The results obtained after compressing a series of images suggest a compression performance of 0.5. The results show a clear advantage over the IBM Q-coder that only manages a best of 0.9 when processing the same set of images. This compares a 7th order context-based binary model with a 0th order context-free byte model and seems to indicate that the second one wins. The IBM Q-coder is targeted to bi-level images and its model is hardwired to this objective. A bi-level image has a symbol granularity of one bit and therefore it is very efficient to predict a whole symbol using 7 preceding symbols as the Q-coder does. If the symbol granularity is the byte, however, this rationality is lost and the system performance degrades. The compression ratio of the weighted limited past history model plus arithmetic coder is modest at 0.5. The reason is not in the implementation itself but the original concept of modelling a data source with a 0th order context-free model. Unfortunately no details are given in the hardware section on gate count or throughput but the complexity of the algorithm is considerable.

[Printz93] presents a non-adaptive 0th order context-free multi-alphabet model with an arithmetic coder. The modelling unit is fixed and is implemented as a look-up table that produces 2 values for each of the 256 possible inputs corresponding to its cumulative symbol probability and symbol probability. The system eliminates the need for a division because it handles probabilities directly. The modelling unit is therefore extremely simple and fast but since it is non-adaptive it will provide arbitrarily inaccurate statistics if it is used in a general compression application. The coder needs to perform 2 multiplications one for each of the 2

equations involved in updating the interval range A ($range_new$) and the code point C (low_new). See equation set [2.6].

The first equation that deals with the interval range is completely embedded in a look-up table that also calculates the multiplication factor for the second equation. A special index-based non-arithmetic representation of A reduces the size of the table. The second equation uses a special re-timed circuitry to obtain a cycle time equivalent to a single 2 bit adder. The design has been implemented in a special hardware prototyping board formed by a group of FPGA's plus memory. It clocks around 32 MHz and since 2 cycles are needed for each byte due to multiplexing and memory access time the throughput is 128 Mbits/s . The compression results are compared against the Q-coder using a combination of image files, text and binary data where it seems to offer some advantage. The design uses data tailored to the file that is going to be compressed to initially construct its probability model. The necessary decoder is not available in this publication although it is pointed out as future work together with adding adaptation capacity in the model. The author estimates a decoder speed 4 times slower than the encoder speed.

2.5.1.3 Tree-based Hardware

The chip described in [Mukherjee93] does not use arithmetic coding but tree-based codes and the byte as symbol granularity. Huffman coding is the most popular tree-based code. The code is static and it does not adapt to variations in the statistical properties of the data source, but, because it is not hardwired but mapped to a memory device, it can be changed to suit the application. For example a different code could be devised if the expected data is image data, text data or binary data. This is an advantage over a hardwired code but its performance is limited, for example, to finding a suitable code to process multiple images that could have very different statistical properties. If the switching process is done very often speed will be lower. Most of the paper is devoted of how to obtain an efficient mapping of the tree code to the memory device. All the formulation is based on a single tree code what means that the associated model is 0th order. If for example the model was 1st order 256 different tree codes will exist one for each possible context. The device complexity would be greatly increase with 256 memory devices plus more complex coding and decoding functions to multiplex among them. The context is expected to change after processing every input symbol so to use a single memory and to load it with the corresponding tree-code stored outside the device is not a sensible option.

The tree has the property that 2 bits are associated to each edge extending from parent node to child node so fewer interactions are needed to reach the leaf nodes starting at the root. The chip has been fabricated using a 2- μ m SCMOS technology with a clocking frequency of 83.3

MHz. The limiting factor is the memory access cycle time since several accesses are needed to code each input symbol.

A compression ratio of 0.5 is assumed so each byte of input generates 4 bits of output. Each tree edge generates 2 bits of output so around 2 memory accesses are required to generate the 4 output bits. This assumption is used to report a performance of 95.2 Mbits/s for compression and 60.6 Mbits/s for decompression. The throughput is highly dependent on the input data source since it would be halved if the compression ratio drops to 1.0 because around 4 memory cycles will be needed to process each input symbol. It is unlikely that a static Huffman coder plus a 0th order model achieved a compression ratio of 0.5 if used as general compressors.

An adaptive Huffman coder implementation in hardware is presented in [Liu95]. Adaptive Huffman coding involves modifying the Huffman tree after each symbol is coded or decoded. This could be a very time consuming task since each coding step could produce a very different Huffman tree. The adaptive algorithm uses a tree tuning strategy that does not rebuild the tree. It also uses a parallel technique to perform both tasks: generate the codeword and adapt the tree visiting the nodes only one time from leaf to root. This technique has the potential of halving the processing time if compared with a sequential approach. A scheme is devised to avoid interference between the code generation and tree tuning process. This interference could result in incorrect operation. The design is based on using CAM modules to store the information associated to each node and to speed up the tree adaptation process. The coding process can generate almost one bit of codeword per cycle. This measure of throughput is related to the output of the coding algorithm and not the raw input data as usual since it depends on the length of the codeword. If the input are bytes and a compression ratio of 0.5 is achieved then an input symbol would be processed in 4 cycles but this value would degrade to 8 cycles if the compression ratio is 1.0. A worst case expansion will affect throughput even more. No hardware details are given so it is not possible to know the clock frequency of the design. The decoding process is more complicated than the coding process because the dependencies present in the algorithm prevent it from using the same parallel technique. Therefore a sequential decoder is adopted. A frequency preset approach is proposed so only a few nodes need adjusting to tune the tree after it has been traversed to decode the codeword. The best case scenario processes one bit of input per cycle but this could degrade to 0.5 bits of input per cycle if all the nodes need adjusting. Moreover this figure again refers to input compressed data throughput and not as usual to the output uncompressed data throughput so it is data dependent. The output uncompressed data throughput is highly dependent on the data compressibility because even if a bit of input compressed data is decoded per cycle the output throughput will be determined by the

compression ratio. Worst compression ratio implies more cycles to decode a symbol and therefore a lower throughput. Finally, the model is a 0th order model which should produce a modest compression ratio when combined with a Huffman coder.

2.5.2 Dictionary-based hardware

Dictionary methods try to replace a symbol or group of symbols by a dictionary location code. The modelling stage is given extra importance while coding is simplified. Some dictionary-based techniques use simple uniform binary codes to process the information supplied by the model. Hardware dictionary-based compression is very popular and successful, achieving excellent throughput and competitive compression ratios.

2.5.2.1 LZ1 Hardware

LZ1 (LZ77) derivative devices have achieved significant commercial success. Chips implementing the ALDC algorithm (Adaptive Lossless Data Compressor) by IBM [Slattery98b] and LZS (LZ STAC) algorithm by Hifn [Hi/fn96] (previously STAC Electronics) illustrate this situation. The usage of these devices to improve system performance is well accepted. The fundamental reasons are that LZ1 derivatives achieve competitive compression with low complexity using multi-symbol alphabets. This in turn allows high throughputs in the order of Mbytes/s and not Mbits/s as with the previously discussed statistical approaches. Although the compression ratio of statistical methods is in theory superior to dictionary-based methods this is only true when using complex algorithms such as those described in section 2.4.1.1.1. These methods are unsuitable for fast hardware implementations.

The ALDC compression algorithm uses the same principles as the IBMLZ1 [Cheng95] chip. The model is based on a CAM used to store the history data. Several versions are available where the CAM varies in size from 512 bytes to 2048 bytes depending on the complexity and compression required. The dictionary is maintained as a circular buffer that keeps the sliding-window functionality typical of LZ1 algorithms. If 2 or more bytes are matched consecutively in the CAM a match is detected and the output is formed by 2 fields preceded by a single bit indicating a match condition. The first field is the match length that is coded using a logarithmic code derivative from a Huffman code. The maximum match length is 271 which is found to be the best value after extensive simulation results. The second field is the position in the CAM where the match starts and it is coded using simple uniform binary coding. If a match is not detected the symbol is added in literal form to the output preceded by a single bit

indicating a miss. This operational mode limits expansion to 12.5% when each 8-bit input symbol misses and becomes 9 bits in the output. The ALDC algorithm was implemented in a $0.8\ \mu\text{m}$ CMOS technology and clocks at 40 MHz to obtain a throughput of 320 Mbits/s with a complexity of 75 Kgates. This device is called ALDC1-40S [IBM94] and it is available as a hardcore from IBM microelectronics. IBM literature [Craft98] also reports that using a more recent technology such as IBM CMOS 5 (standard cell/gate array $0.35\ \mu\text{m}$, 6 levels of metal) the critical path located in the CAM searching operation can be further reduced to only 10 ns. The compression/decompression throughput is then 800 Mbits/s with a clock frequency of 100 MHz. A license version of the ALDC algorithm is also available from AHA in the AHA3521 chip [AHA97a]. This chip is implemented in a $0.5\ \mu\text{m}$ and clocks at 40 MHz for a 160 Mbits/s throughput because 2 cycles are used to process each byte. Some algorithm extensions to the ALDC method are also reported in [Craft98] to produce 2 variants named BLDC and cLDC algorithms. These extensions are based on using a front-end run-length coder pre-processor to feed the ALDC chip and improve compression without affecting speed.

IBM also introduces in [Franaszek96] a method for obtaining parallel LZ1 compression using cooperative dictionary construction. The idea is that the input data block is divided into a number of sub-blocks (typically 4) and these are processed in parallel using independent coders. The result is concatenated in a single block and a prefix area is added to indicate the decompressor how the single compressed file must be split to feed the independent decoders. To alleviate the problem of a decrease in compression efficiency the dictionary is shared among the coders and maintained in common. The effect is that more data is available as history data for each sub-block and compression improves. This simple concept can not provide the same level of compression as a single device solution because the history data available to compress a symbol n with a dictionary size m is not the m symbols that preceded symbol n but gaps exist in the history buffer corresponding to data assigned to the other coders and not yet processed. Also this solution precludes the use of incremental reception and transmission of data since the entire data block must be available before the compression operation can be initiated and all the compressed data must be available before the header can be added and transmission started. Therefore a higher latency is added with this technique.

The Hi/fn devices realize in hardware the LZS [Hi/fn96] algorithm developed by the same company. The LZS lossless data compression algorithm is a LZ1 derivative that uses a 2 Kbyte history buffer. The coding format of the LZS algorithm is similar to ALDC. A match is coded as 2 fields preceded by a single bit indicating a match condition. The first field is the offset or pointer to the buffer location where the match starts. The offset can be coded as a 7-

bit offset or 11-bit offset preceded by a single bit to differentiate between both. This is in contrast with ALDC where the offset was not coded. This coding scheme improves compression by assuming that short offsets to a maximum length of 128 are more common than offsets to the beginning of the buffer. The algorithm exploits the locality of reference effect that establishes that data that has just been seen is more probable to be seen again. The second field is the match length. This is coded using a prefix free code similar to a static Huffman code. A miss is coded by adding the literal to the output preceded by a single bit like in the ALDC algorithm. Expansion is limited to 12.5% when all the 8 bit literals are transformed into 9 bit codewords.

A RAM is used in some low-end LZS hardware products from Hi/fn to realise the history buffer. A tuneable feature is included so the amount of searching done in the buffer can be externally controlled trading throughput for compression. Compression throughput is limited when using RAM to 64 Mbits/s but since RAM tends to be plentiful it is easy to include multiple-history support. Multiple-history support means that different history buffers are maintained independently for different communication channels improving compression. The algorithm switches among them depending on which channel is active. The high-end products use a CAM to implement the history buffer. In this case searching is done exhaustively in a single cycle. A CAM-based device has been fabricated in a 0.5 μm CMOS technology and it is available from Hi/fn with the name 9610 [Hi/fn98b] Data Compressor Processor. It clocks at 50 MHz with a throughput of 400 Mbits/s. A more recent version named 9600 [Hi/fn99] has been fabricated in 0.35 μm CMOS technology with a maximum throughput of 640 Mbits/s when the internal logic is clocked at 80 MHz. This device includes also the novelty of being a full-duplex device so compression and decompression can be done simultaneously for a combined performance of 1.25 Gbits/s. All the other devices discussed in this section are half-duplex which means that the processor must compress part of the active time and decompress the rest. Full-duplex functionality is becoming a feature of data communication standards such as Gbit/s Ethernet. This network when running in full-duplex mode can carry each way 1 Gbit of data per second so full-duplex functionality in a single compression/decompression chip is a useful feature.

[Surk97] presents a PE-based (Processing Element) VLSI architecture for the LZ1 algorithm. Each PE compares the incoming input symbol with the symbol it stores in 1 cycle and shifts the symbol to its neighbour. The single dimension array of PE's behaves in a mode similar to a modified CAM-based design if the CAM cells are redesign to input data from their immediate neighbours. New data is input in the right most PE and the data located in the left most PE is eliminated from the history buffer. When a PE maintains its match signal active

for more than 1 cycle a string match is detected. The output identifies the PE position and the match length that equals the number of cycles the PE maintained its match signal active. There are a total of 1024 PE's and this value also defines the history buffer size. The maximum match length is limited to 16. This should affect compression since values around 256 are in general more appropriate for this process [Cheng95]. The miss problem is dealt in the traditional way of preceding all the codewords with a single bit. The basic symbol is 7-bits wide so the compressor as described is only suitable to compress ASCII coded text. The data input rate is constant and post-layout simulation indicates a performance of 700 Mbits/s using a 0.5 triple-metal CMOS technology and a 100 MHz clock.

[Jung98] presents another VLSI LZ1 implementation for optimisation of wireless local area networks. Much of the paper is devoted to analyse the effects of lossless data compression in wireless networks. The LZ1 algorithm codeword format has the classical 2 fields : offset plus match length but this time the codewords are of fixed length 16 bits which means that uniform binary coding is used for both values. Dictionary length is 512 so 9 bits are used for the match location while 7 bits are left for the match length. If a match is of length less than 3 then 2 bytes in literal form are added to the output. A single byte is added to 8 2-byte codewords to distinguish between compressed codewords and uncompressed literals. This technique limits expansion to 6.25% but it also increases latency since 8 2-byte codewords must be stored internally before any output is produced. No details are given on compression performance but an average compression of 0.5 is assumed for this type of compressors working on typical network data. A parallel architecture is presented using 512 PE's organised in a single-dimension array. This architecture uses the same CAM principles as [Surk97] to process 1 byte of raw data per cycle. This architecture is deemed not suitable to support multi-channel compression because the overhead of switching the dictionaries is considered too high if a different dictionary must be uploaded in the CAM each time the compression channel changes. The authors proposed a different mapping of the original algorithm to base the algorithm in RAM and enable multi-channel support. The resulting design is simulated using a 1.2 μm CMOS technology and it clocks at 100 MHz producing a throughput of 50 Mbits/s. The complexity is reported of around 36K transistors.

[Chen98] presents a linear systolic array VLSI design for LZ1 compression. The systolic array includes a dictionary buffer with 512 characters distributed over 64 systolic cells. Each cell compares an input character concurrently with the 8 character dictionary section that it holds in that cycle. The systolic cell outputs to the neighbouring cell the character to be coded plus the longest match string that started with that character. The design has been implemented in a 0.6 μm standard cell library and it uses around 90 K gates. The operating

frequency can reach 91 MHz with a throughput of 728 Mbits/s. The architecture needs a number of comparators equal to dictionary size but it can operate at a frequency independently of the dictionary size. The reason is that cell frequency is not affected by the number of cells present in the systolic array. Latency is proportional to the number of cells present in the array so this architecture using a 512-character dictionary has a latency of 64 cycles. It offers a compromise between the single cycle operation of CAM architectures where a high fanout could prevent dictionary extension and the high latency of PE-based architectures with a single comparison is done in each PE.

[Nusinov94] also presents a VLSI LZ1 derivative for multi-channel compression. The LZ1 proprietary implementation is called Codex Ziv-Lempel (CZL) algorithm. The dictionary length can be of maximum of 1024 bytes. The codewords are organised in the 2-field format: length plus location. The match length is coded using a Huffman-style conversion table while the location is coded using a phased binary coding so when only a few dictionary locations are active the chip obtains improved compression. A match is considered valid if the length field is at least 2. Otherwise a miss is coded using a length 0 plus the byte in literal form replacing the location field. This approach differs from the previously discussed techniques and deals with the expansion problem in a less efficient way. It is probable that the conversion table used for the match length assigns very few bits to the match length 0 to minimise the expansion problem but no information is available in the paper. Multiple dictionaries up to a maximum of 2000 are stored externally in RAM. During coding the appropriate one is uploaded in internal CAM (Content Addressable Memory) memory to allow parallel searching. The overhead of uploading an external dictionary with 1024 bytes to internal CAM should be very high since only an 8-bit bus interface is available. The internal CAM accounts for most of the logic in the chip and it does not include shifting capabilities. A CAM cell can only activate its match signal if the neighbouring CAM cell did so in the previous compare cycle. This mechanism allows the input string to be progressively matched along the CAM dictionary. Updating is done simultaneously in the internal CAM and external RAM so there is not need to download the CAM contents after compression switches to a different channel. During decoding the external RAM is used directly. The need to update the external RAM after every compare cycle means that two cycles are needed to process each byte. The chip clocks at 20 MHz and has a throughput of 80 Mbits/s. Compression is reported to be around 0.5 to 0.33 for typical data but no experimental results are provided. The chip has been fabricated using a 0.8 μm CMOS technology but no details are available on gate count or transistor count.

2.5.2.2 LZ2 Hardware

The LZ2 algorithm was developed 1 year later than the LZ1 variant and it has not become as widely used as this one. The reason is that it uses a more complex dictionary structure where dictionary entries are formed by concatenating the next incoming data character after using a dictionary entry to that entry forming a new dictionary entry. The LZ2 algorithm produces an output code specifying dictionary locations where data and length can be found. This code output can be simple uniform binary coding where the number of bits is the $\log_2(\text{different_possible_codes})$ or more refined coding strategies can be used. Although theoretically superior to the LZ1 algorithm, LZ2 is at a disadvantage when compressing small packets of data and requires more complex structures that hampers its throughput.

The DCLZ [AHA96] family of compressors from AHA (Advanced Hardware Architectures) are LZ2 derivatives. The DCLZ (Data Compression Lempel Ziv) was originally developed by Hewlett-Packard laboratories around 1989 and used in their tape drive [Bianchi89]. The hardware DCLZ works by storing a dictionary of 4096 entries organised as a linked-list with the first 256 values assigned to the ASCII values. Each entry in the dictionary contains 23 bits: 8 bits are assigned to hold an ASCII value, 12 bits are assigned to hold a location value and the rest are used as flags. New dictionary entries are added to the dictionary storing the byte that stopped the string matching procedure in an unused position. The location loaded in this position is a pointer to the dictionary location that holds the previous byte part of the string. The string is linked in this way and eventually a location address points to the byte that originated the string in the first 256 positions. This simplifies dictionary structure since the width of the array is fixed. The AHA3101 chip stores the dictionary in the device and when it becomes full dictionary adaptation freezes. Periodic resets of the dictionary are done when compression performance degrades.

The codewords output by the algorithm are simple dictionary locations addressing the location that holds the byte that terminated the string. There is no need for match location lengths as in LZ1. There is no special handling of a miss condition since misses are coded as pointers to the first 256 locations where the ASCII code is stored. The dictionary needs a 12-bit pointer locations when full and this means that a worst case expansion transforms 8-bit input symbols into 12-bit output codewords. The worst case expansion is then 50% that could be unacceptable in many applications. The expansion activated reset mechanism should avoid this situation because when the dictionary is empty only 9 bits are used for the codeword. The dictionary codewords are of length 9 to 12 bits depending on how many entries in the

dictionary are active. This mechanism is not as efficient as phased binary coding (PBC) where the location length is optimal for a dictionary that grows one position at a time. For example if 513 locations in the dictionary have been used PBC assigns 9 bits the first 511 locations and 10 bits to the other 2 but the previous scheme will always use 10 bits. The throughput is affected by the compression ratio and it is therefore data dependent. It is worst under expansion conditions since it is necessary to access the external memory more often to perform updates. Since the device is based on RAM the searching operations are also very time consuming. The throughput of the AHA3101 is on average 20 Mbits/s.

A more recent addition to the DCLZ family of devices is the AHA3211 [AHA97b]. This chip uses internal CAM to replace the external RAM and to improve the searching and adaptation speed. It clocks at 40 MHz and it has a data independent throughput of 160 Mbits/s. It has been fabricated using a 0.5 μm CMOS technology.

[Bunton92] presents another LZ2 implementation that improves upon the [Bianchi89] DCLZ LZ2. The [Bunton92] algorithm uses a similar dictionary structure to [Bianchi89] but offers a more advanced dictionary maintenance mechanism where a tag is attached to each dictionary location to identify which node should be eliminated once the dictionary becomes full. The tag mechanism implements a Least-Recently-Used (LRU) policy so the oldest node in the linked-list dictionary and always a leaf in the corresponding virtual trie is declared free to continue dictionary adaptation indefinitely after it becomes full. Removing a non-leaf node will fail the algorithm because trie branches would be unconnected. This technique improves over a dictionary that stops adapting and resets if compression degrades when no more empty nodes are available. Better performance is obtained with a similar size dictionary or alternatively the dictionary can be made smaller for the same performance target. The hardware realisation uses only 1K different locations but it performs similar to a 64K resetting technique. Since the dictionary is much smaller the codewords output from the coder have a fixed-width of 10 bits because growing-dictionaries combined with short start-up phases offer little benefit. The hardware complexity is around 210K transistors using a 2 μm CMOS process plus 20 Kbits of off-chip static RAM to store the tag information. An internal CAM is used to store the dictionary. This implementation achieves a data independent throughput of 108.8 Mbits/s. This rate can be improved up to 160 Mbits/s in the same technology if the RAM's are placed on-chip eliminating the need for off-chip communication. This scheme seems to offer better compression and less complexity (1K dictionary against the 4K in DCLZ) than the [Bianchi89] device. The speed is also very competitive for a 2 μm CMOS technology. It is, however, the Hewlett-Packard device the one that has achieved commercial success and it is in use today in many tape drive storage applications [Cressman94].

2.5.2.3 Other dictionary-style hardware

A very simple lossless data compressor is the run-length coder of [Xiong97] where the output is the data byte and a number byte indicating how many times the data byte has been seen. Although little information is provided in the paper it is obvious that such a simple compression technique can produce high throughput at the expense of a low compression ratio.

The X-Match family of devices developed at Loughborough University [Jones92], [Kjelso95], [Kjelso96], [Jones00] belongs to the category of dictionary-based compressors but they are not LZ derivatives. The X-Match model follows the principles of the BSTW algorithm discussed in section 2.4.3. The X-Match model is based on a 4-byte wide CAM dictionary and outputs a dictionary location indicating where a match was found and match type indicating which bytes out of a maximum of 4 were found. This partial matching characteristic gives name to the method. The X-Match coder uses a phased binary code (PBC) for the match locations and a static Huffman code for the match types. The X-Match coder offers single cycle operation and data independent throughput combined with a very low latency, the features of a high performance compressor. Since it processes 4 bytes of input raw data per cycle it can achieve high throughputs with modest clock frequencies due to its parallelism. Pre-layout simulation indicates a performance of 800 Mbits/s clocking at 25 Mhz using a 0.6 μm gate array CMOS technology. Complexity is around 100 Kgates.

2.5.3 Other Hardware

Other work that can not be classified in the range of statistical or dictionary-based methods corresponds to the genetic algorithm developed in the DCP chip [DCP95]. There is little information on the features of the genetic algorithm although a dictionary table is used. The developers claim very high compression ratios that outperform the LZS algorithm from Hi/Fn. The figures in the DCP documentation show an advantage of the DCP algorithm over LZS of around 25 % and in some cases up to 100% better compression when processing databases although the improvement decreases if targeting text and binary data. When compressing standard data such as the Calgary corpus the compression advantage is around 20%. Worst case expansion is limited to 3%. This chip is implemented in a 1 μm CMOS technology and has a throughput of around 1.64 Mbits/s clocking at 40 MHz. The chip named DCP816 has a complexity of around 15K gates. It supports up to 64 channels of compression/decompression and it uses 512 Kbytes of external RAM per dictionary/channel.

It needs on average 190 clock cycles to process one byte of input data so the algorithm complexity seems to be very high and far from the single cycle operation of other compressor solutions.

The hardware presented in [Sakanashi98] for printer image compression is based on evolvable hardware (EH) and genetic algorithm (GA) paradigms. Evolvable hardware technology is able to change the hardware structure depending on the requirements of the target task and it is normally associated to reconfigurable hardware such as an FPGA. The work presented aims to improve the performance of the JBIG standard based on the IBM QM-coder. The QM-coder compresses a bit of data using a context formed by 10 surrounding bits. The shape of the template that defines which bits are chosen as context can be modified only slightly in the QM-coder. The evolvable hardware chip has 2 main hardware components: A RISC processor and a QM-coder. There are 2 modes of operation, the learning mode and the compression mode. The objective of the GA is to use the learning mode to select the template that offers the best compression ratio for a portion of the image. The GA runs in the RISC processor where it selects different templates to perform compression and uses the amount of data output by the QM-coder to choose the best one. In compression mode a context generator uses the previously selected template for each image portion to provide the QM-coder with a context and a pixel to be coded. Compression ratio using this combination of GA and QM-coder is twice as good as the one obtained by the QM-coder on its own. The algorithm throughput is, however, very low because the learning mode is very time consuming since the QM-coder has to run several times, one time for each template tested. Compression throughput is around 12 Kbits/s. The paper does not present the corresponding decompressor. Unfortunately, the evolvable hardware feature in the chip description cannot be properly identified. The use of a GA to select an optimal template for each image portion is clear and well understood. On the other hand the process of context selection based on different pre-calculated templates during image compression seems more of a multiplexing technique than a technique based on selecting a new hardware architecture and downloading it into an FPGA.

2.6 Summary

This chapter has reviewed the current state of lossless data compression. This section highlights the conclusions of the chapter.

- Current software-base lossless data compression offers compression ratio levels that it would be difficult to improve upon in the future.

Statistical techniques such as PPMZ have brought lossless compression ratios to a value close of 0.2. These figures are considered to be very close to the theoretical entropy limit with limited (if any) room for improvements. Further advances will always obey the diminishing returns rule that means that it is easier to improve compression from 0.8 to 0.6 than from 0.2 to 0.19 and complexity increases exponentially. These methods achieve their performance using a lot of resources and have very low throughputs. They do not achieve their optimal working conditions until blocks of data in the order of Mbytes are compressed as single entities because adaptation is slow and their multiple internal data structures use plenty of data.

- Statistical PPM-style algorithms offer compression superior to dictionary-based algorithms but the complex nature of the operations involved and the variable number of them per symbol made them unsuitable for high-speed on-line hardware-based data compression.

As a rule statistical software compression focuses on very good compression ratios while speed is given a second order importance. Dictionary-based software compression is still more popular and commercial algorithms such as PKZIP and ARJ are illustrative examples. The reason is that although their compression is not as good their simplicity and related speed becomes more important in many real applications. It is also true that PKZIP and ARJ have been around longer than PPM style algorithms. These are something of a novelty because until recently there was not suitable hardware in the public domain which sufficient power to execute them.

- Current hardware-based statistical compression is either slow using binary alphabets or offers poor compression performance using 0th order models. Complexity and speed limitations prevents the use of multi-alphabet arithmetic coding (to get speed) and high-order context-based modelling (to get compression) in hardware.

Statistical compression in hardware is rare because the main objective is usually throughput and this is not something in the nature of a statistical method. The limitations on complexity are also harder to break. The most popular statistical hardware chip is the IBM Q-coder whose performance is in the order of Mbits/s which is far from the requirements of Gbit/s established in chapter 1. The Q-coder is a successful example of a binary fixed-order context-based

model plus arithmetic coder in hardware. The main limitation for speed in this case is the bit symbol granularity. Although research has been done to use wider alphabets the compression ratios are poor because complexity (and therefore throughput) requirements prevent from using high-order modelling. The 0th order context-free models used in all the multi-alphabet implementations seen so far are simply not powerful enough to compete with dictionary-based hardware compression. Multi-alphabet variable-order finite-context models such as PPMC or PPMZ currently do not exist in hardware.

- Dictionary-based hardware data compression is popular and well accepted as a means of improving the performance of an electronic system. It offers competitive compression and high-speed to successfully operate on-line in storage and network environments if their speed requirements do not exceed the value of 1 Gbit/s.

Dictionary-based hardware is popular and successful with examples such as LZS(Hi/Fn), ALDC (IBM), DCLZ (AHA) currently improving the performance of data communication networks and storage systems. Attractive compression ratios in the order of 0.5 offer the possibility of doubling the capacity of an electronic system with minimum investment. These algorithms are based on byte alphabets and process one byte of input data per clock cycle. The Hi/Fn device can run up to 640 Mbits/s with a complexity of around 100 K gates and it offers full-duplex functionality. This is the fastest device that is commercially available today as a single lossless data compression solution. The IBM devices are limited to 320 Mbits/s because their chips are based on an older technology (0.8 μm) although IBM offers them as synthesizable cores to be added to a more complex SoC (System On a Chip) device. A throughput up to 800 Mbits/s is expected if using a more up-to-date technology (IBM CMOS 5 0.35 μm).

In general these devices lack the performance to support a >Gbit/s compressed network and could become the bottleneck in the system. Their operational mode also adds considerable data latency because they multiplex pins to get compressed and uncompressed data in and out of the chip

- Common to these high-performance dictionary-based devices is the use of a CAM (Content Addressable Memory) circuit instead of a RAM circuit to store the dictionary. CAM's enable single cycle search and adaptation of the entire model unit whilst RAM based models processed only a dictionary location per cycle.

Table 2.2 shown in the following page summaries the features of the most significant lossless data compression hardware implementations. Only those designs where silicon is available are reported in Table 2.2. It is clear from the throughput measurements of column 11 in Table 2.2 that all the current implementations fall short of the Gbit/s benchmark. There is also an order of magnitude difference between the throughput obtained by the dictionary-based implementations and their statistical counterparts. The reason is that although a similar clock frequency can be obtained with similar technologies the dictionary-based compressors process at least 1 byte per cycle while the statistical implementations are typically limited to 1 bit per cycle.

The following acronyms are used:

BAC= Binary Arithmetic Coder

EHW = Evolvable HardWar

GA = Genetic Algorithm,

ASM = Adaptive Statistical Model

FSM = Fixed Statistical Model

MAC= Multi-alphabet Arithmetic Code

FHC=Fixed Huffman Coder

ADM=Adaptive Dictionary Model

FHSC=Fixed Huffman-Style Coder

PBC = Phased Binary Coder

UBC= Uniform Binary Coder

Chip Name	Developers	Year	Technology			Algorithm			External RAM/ Internal CAM	Throughput (Mbits/s)
			Process	Complexity	Clock (MHz)	Name	Model Type	Coder Type		
Qx-coder	IBM [MITCHEL98]	1998	CMOS IBM 5 0.35 μ m ASIC		75	Q-coder/ QM-coder	ASM	BAC	No/No	64
Q-coder	IBM [ARPS88]	1988	HCMOS 1.5 μ m ASIC		20	Q-coder	ASM	BAC	No/No	20
	[KUANG98]	1998	CMOS 0.8 μ m ASIC		25		ASM	BAC	No/No	3
	DEC corporation [PRINTZ93]	1993	Prototype Board DECPeRLe-1	8 Xilinx xc3090 FPGA's	32		FSM	MAC	No/No	128
ALDC1-40S	IBM [CHENG95]	1995	CMOS 0.8 μ m ASIC	75K gates	40	ALDC	ADM - LZ1	FHSC, UBC	No/Yes	320
AHA 3521	AHA	1997	CMOS 0.5 μ m ASIC		40	ALDC	ADM - LZ1	FHSC, UBC	No/Yes	160
AHA 3211	AHA	1997	0.5 μ m CMOS ASIC		40	DCLZ	ADM - LZ2	UBC	No/Yes	160
	[Burton92]	1992	CMOS 2 μ m ASIC	210K transistors, 20 Kbits SRAM	20		ADM - LZ2	UBC	Yes/Yes No/Yes	108.8 160
Hi/Fn 9610	Hi/Fn	1998	CMOS 0.5 μ m ASIC	100K gates	50	LZS	ADM-LZ1	FHSC, UBC	No/Yes	400
Hi/Fn 9600	Hi/Fn	1999	CMOS 0.35 μ m ASIC	100K gates	80	LZS	ADM - LZ1	FHSC, UBC	No/Yes	640
X-Match	[GOOCH96]	1996	CMOS 0.6 μ m ASIC	100K gates	100	X-Match	ADM - BSTW	FHC, PBC	No/Yes	800
	[JUNG98]	1998	CMOS 1.2 μ m ASIC	36K transistors	100		ADM - LZ1	UBC	Yes/No	50
DCC	Motorola [NUSINOV94]	1994	CMOS 0.8 μ m ASIC		20	CZL	ADM-LZ1	FHC, PBC	Yes/Yes	80

Table 2.2. Summary of lossless data compression hardware.

Chapter 3

The X-Match method

3.1 Objectives of Chapter

The objective of this chapter is to select a research vehicle to progress further the area of lossless data compression hardware. The basis of this selection is to choose a system or a concept that shows high performance features to enable us to achieve the throughput and compression requirements stated in chapter 1.

These requirements can be summarised as follows:

- *Low latency.* Most application environments are sensible to latency that should be kept as small as possible. Latency is one of the variables together with throughput that defined the speed of a compression method. Incremental transmission is also very important so the compressor can start compressing data before the whole data block has been received and transmission of compressed data can start before the whole block has been compressed.
- *Data independent throughput.* It is important to have a constant and data independent throughput in the uncompressed port to ease system integration. In this way the uncompressed section of the system can be kept unaware that a compression element has been introduced in the data path leaving aside a significant increase in throughput. The data throughput in the compressed port is data dependent since it depends on the instantaneous compression ratio.

- *Over 1 Gbit/s throughput.* The throughput in the uncompressed port should be higher than 1 Gbit/s to be able to handle current high performance storage devices and communication networks. The throughput requirements are expected to grow to 10 Gbit/s in the next few years.
- *Compression ratio of 0.5 on typical computer data.* The higher the compression the better but a lossless compressor that doubles the performance of the system where it is integrated clearly justifies the use of compression. This compression ratio must be achieved also when operating with small data blocks since many digital systems work with data blocks ranging in size from ≥ 32 bytes to ≤ 4 Kbytes.
- *Low complexity.* Although the number of gates available in a silicon chip is constantly growing the final aim is to produce an architecture feasible in current or soon to be available technology. Low complexity produces a cost effective solution with the added advantage of low power consumption. FPGA technology is a valuable tool to evaluate the benefits of our design so the constraints of this programmable hardware must be taken into account.

3.2 Features of the X-Match lossless data compression method

3.2.1 Introduction

X-Match was already introduced in chapter 2 as a fast dictionary-based compression algorithm suitable for hardware implementation. We will further analyse its positive and negative points in this section as a possible selection to advance the field of lossless data compression. The main reason to choose X-Match as a valid candidate is that it shows a clear performance advantage if compared with other solutions discussed in chapter 2. None of the binary arithmetic coders of chapter 2 are close to a figure of 1 Gbit/s and they tend to exhibit dependencies between data compressibility and data throughput. The multi-alphabet arithmetic coders do not offer the compression performance because they are limited to context-free models. The dictionary-based machines get closer to 1 Gbit/s but they still struggle because processing is limited to 1 byte per cycle so they need high clock ratios and depend on advance technology. X-match can achieve good throughput with modest technology because it gets its performance from processing multiple symbols per clock cycle and not from high clock ratios.

3.2.2 X-Match algorithm description

The X-Match algorithm uses a dictionary of previously seen data and it attempts to match the current data element with a data element present in the dictionary. It obtains compression when this matching is successful. These are the key features of the algorithm:

- Fixed width dictionary of 4 byte words named tuples to provide high, data independent throughput.
- Variable length dictionary that dynamically grows when unknown data elements are processed. This means that during an initial stage only a valid subset of the dictionary locations are assigned codewords. This feature provides good compression ratio when processing small data blocks.
- A partial matching strategy to improve compression so not all the bytes need to match in a dictionary location for the match to be considered valid.
- Data expansion limited to 3.125% when no valid match is found in the dictionary because a single bit is added to the new tuple (32 bits are translated into 33 bits).

The result of searching the dictionary can be a match or a miss. Since the algorithm uses a partial matching strategy several types of matches are possible where all or some of the bytes at different positions inside the tuple match. Those bytes that do not match are transmitted literally. This partial match concept gives the name to the procedure – the X referring to ‘don’t care’. At least 2 bytes have to match and when no valid match is generated a miss is codified adding a single bit to the literal. The dictionary is maintained using a move-to-front (MTF) strategy [Bentley86] whereby a new tuple is placed at the front of the dictionary while the rest move down one position. When the dictionary becomes full the tuple placed in the last position is discarded leaving space for a new one.

The coding function for a match is required to code 4 separate fields as follows:

- A first bit set to 0 indicating a match.
- *The match location.* It uses PBC (Phased Binary Code) as seen in section 2.4.2.2.1, chosen for its suitability for hardware implementation. PBC is

characterised by using smaller codes during the growing stage of the dictionary that starts in an initial empty state.

- *A match type*. That indicates which bytes of the incoming tuple have matched. This is codified using a static Huffman code as seen in section 2.4.1.2.1.2 based on the statistics obtained through extensive simulation.
- Any extra characters that did not match transmitted in literal form.

The coding function for a miss is required to code 2 separate fields as follows:

- A first bit set to 1 indicating a miss.
- The 4 non-matching characters in literal form.

The algorithm is given as pseudo-code in Figure 3.1.

```

Clear the dictionary;
Set the next free location (NFL) to 0;
DO
    {
    read in tuple T from the data stream;
    search the dictionary for tuple T;
    IF (full or partial hit)
        {
        determine the best match location ML and the match type MT;
        output '0';
        output phased code for ML;
        output Huffman code for MT;
        output any required literal characters of T;
        }
    ELSE
        {
        IF ( T is not the first tuple)
            output '1';
        output tuple T;
        }
    IF (full hit)
        move dictionary entries 0 to ML-1 by one location;
    ELSE
        {
        move all dictionary entries down by one location;
        increment NFL ( if dictionary is not full);
        }
    copy tuple T to dictionary location 0;
    }
WHILE (more data is to be compressed);

```

Figure 3.1. The X-Match algorithm

Initially all the entries in the dictionary are empty and a tuple is added to the front of the dictionary while the rest move one position down if a full match has not occurred. The move-to-front technique is only applied when dealing with full matches. In this case the tuples from the first location until the location previous to the matching tuple move down one location while the matching tuple is placed at the front of the dictionary. The number of entries in the dictionary grows dynamically, thus if the input data only contains a few different tuples then the dictionary remains small. Since the number of bits needed to code each location address is a function of the dictionary size greater compression is obtained in comparison to the case where a fixed size dictionary uses fixed address codes for a partially full dictionary. Only one full match can occur at any time in the dictionary since the algorithm makes sure that no 2 locations contain the same data. Several partial matches are possible simultaneously so the one that produces a shorter output is selected as valid.

3.2.3 X-Match hardware analysis

The architecture is based around a block of CAM to realize the dictionary. This is necessary since the search operation must be done in parallel in all the entries in the dictionary to allow high throughput. Latency is also kept to a minimum because the result of the search operation at time t is available at time $t+1$ for further processing. The size of the CAM is 128 words with 32 bits per word and it has to be selectively shiftable to be able to reorder itself adapting to the incoming stream of data. The selectively shiftable characteristic implies that each word of the CAM maintains its data or loads the data of the previous word depending on the value of its associated bit in the adaptation vector produced by the dictionary maintenance functions.

3.2.3.1. Compressor architecture

An overview of the compressor architecture is presented in Figure 3.2. The tuple to be coded searches the CAM array trying to find a match. The output of this process is passed to the best-match decision logic that resolves which of the possible matches (if any) is the best. Then the match location is coded using a PBC that depends on how many entries are valid in the dictionary as indicated by the next-free-location (NFL) counter and the match type is coded using a Huffman code. Any needed literal characters are added and the result is passed to the assembly logic which packs groups of 64 bits together before indicating the availability of compressed data. The shift control logic generates the adaptation vector to rearrange the dictionary in the next cycle based on the match information.

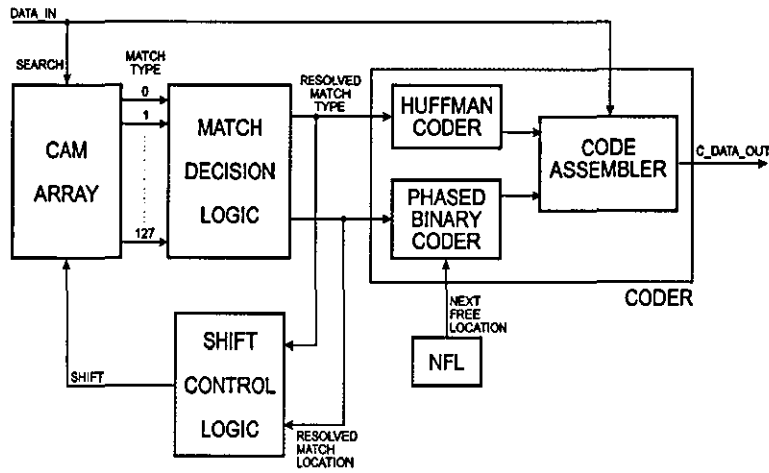


Figure 3.2. Architecture of the compressor

3.2.3.2 Decompressor architecture

Figure 3.3 shows the decompressor architecture. The compressed data enters the decoder to produce a match location and a match type in the phased binary decoder and Huffman decoder. The byte disassembler is used to shift in the correct number of bits of input data as a function of the variable-length codes found. The match location is used to multiplex out a specific position in the CAM array and the match type determines what literal characters (if any) are needed to recreate the original data. The shift control logic generates the adaptation vector to rearrange the dictionary following the same pattern as in compression.

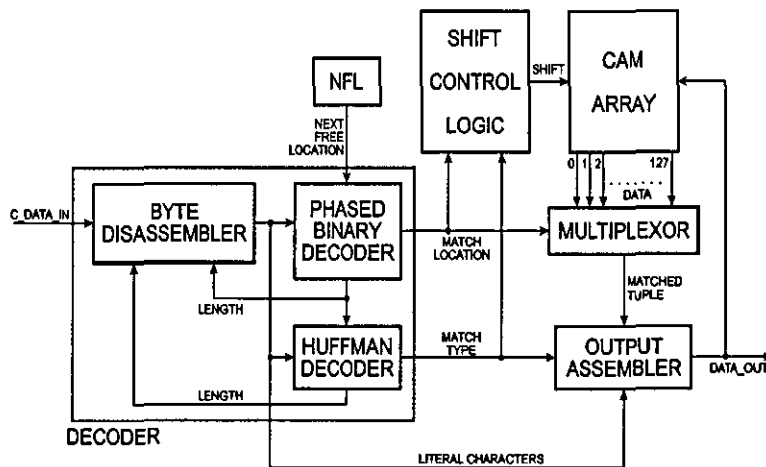


Figure 3.3. Architecture of the decompressor

3.2.3.3 Hardware performance

X-Match has been synthesised into a 0.6 μm gate array technology. Pre-layout simulation results estimated a maximum clock frequency in the compression section of 25 MHz for a data independent throughput of 800 Mbits/s. The decompression section can clock at 35 MHz and the throughput is 1120 Mbits/s. In practice, a single clock should be used for compression and decompression so the overall throughput is 100 Mbytes/s. The slowest critical path extends from the search data, through the CAM array, match decision logic, shift control logic and back to the CAM array to provide the necessary information to reorder the dictionary. The latency of the device due to pipelining is 5 clock cycles during compression. Decompression latency is 2 cycles. Figure 3.4 shows the critical path.

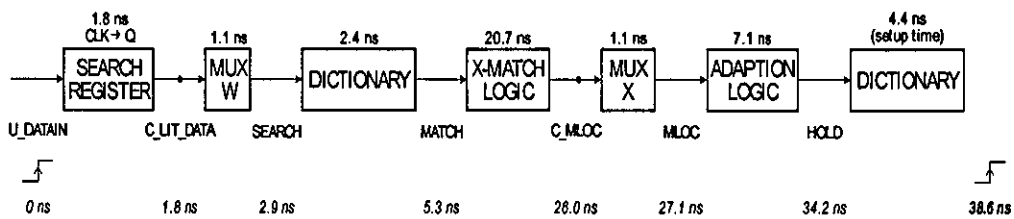


Figure 3.4. X-Match critical path.

The compression process is usually slower than the decompression process because the extensive search operation in the dictionary to find a possible match is replaced during decompression by a simpler look-up operation using the match location to address the dictionary.

The X-Match description includes logic to interface to SRAM compression memory where 64 bits of compressed data are written in each access cycle during compression or read during decompression. An internal register must be loaded with the uncompressed block size at the start of the operation. An internal counter is enabled at the start of the compression or decompression process and the device stops when the count value equals the uncompressed block size. Several uncompressed block sizes can be used or the chip can run in unblocked mode. Since 14 bits are used to interface to the compressed SRAM the maximum compressed block size allowed is $2^{14} \times 64 \text{ bits} = 128 \text{ Kbytes}$. The algorithm does not insert any special termination marker in the compressed stream so the design relies in knowing the uncompressed block size to detect when to stop uncompressing data. The same device can

perform compression and decompression but not simultaneously. Figure 3.5 shows the pin-out of the X-Match design. The need for a 64-bit wide compressed bus is due to expansion conditions where the 32-bit input word is transformed into a 33-bit output word. A bottleneck could appear in the compressed data port if the device uses a 32-bit wide bus because no buffering exists to handle a consecutive series of misses.

The estimated gate count of the design is 100 Kgates including the pipeline registers but excluding additional logic for production testing. The estimated die size is 13.0 x 13.0 mm. Most of the logic (80 Kgates) corresponds to the 128 x 4 bytes CAM logic.

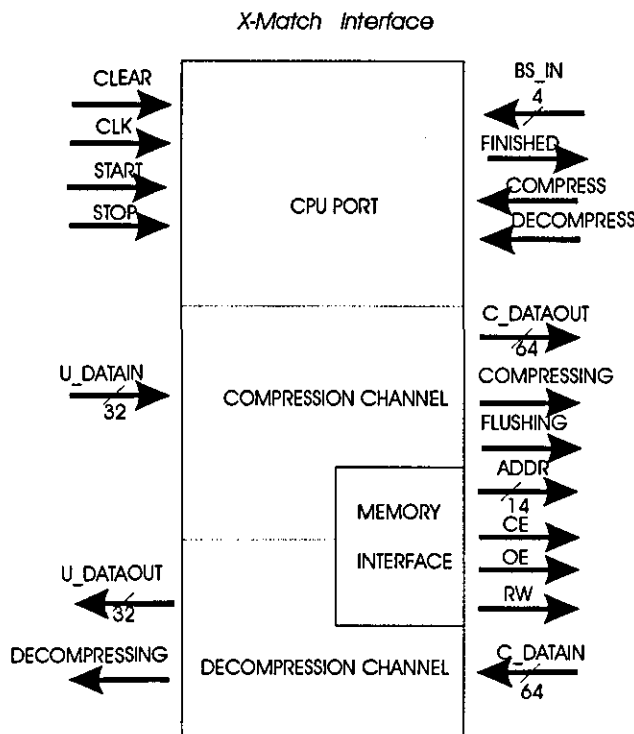


Figure 3. 5. X-Match interface.

3.3 Mapping of research objectives to X-Match

3.3.1 How can we produce a faster X-Match?

The 3 elements present in a compression system, namely: Model, Coder and Packer introduced in section 2.3 affect speed. It is important to identify which one is the performance bottleneck so our efforts can be directed. The original X-Match estimates a bottleneck in the adaptation process in the model as mention in section 3.2.3.3. To identify and solve this

bottleneck is therefore a priority before further analysis is carried out in the other components of the system.

X-Match is targeted to main memory compression where a single bus is used to transfer data to and from main memory. Full-duplex operation is becoming increasingly popular in network standards able to transmit and received data simultaneously. It is also useful in other applications such as a printer that receives data in uncompressed format, compresses it to store it temporarily in local memory and then concurrently decompresses it when the print engine requires more data. This application does not use compression to increase the bandwidth of the data pipe but to increase the storage capacity of local memory. It is important to analyse the possibility of developing a full-duplex solution so both processes compression and decompression can be executed simultaneously for a combined performance twice as high as a half-duplex device.

3.3.2 How can we produce a better compressing X-Match?

Coding better or modelling better can improve compression efficiency. The third element of a compression system, the packer, does not have an impact on compression. Its function is to assemble variable length codes into fixed length codes without affecting the total number of bits. There is a strong interdependency between models and coders so more efficient modelling such as the high-order context-base models of section 2.4.1.1.1 requires more efficient coding such as the arithmetic coders of section 2.4.1.2.1 able to exploit the high accurate information passed by the model. In chapter 2 statistical modellers and coders were classified as those with higher compression performance but they were also found to be particular slow. Arithmetic coding is slow not only because operations involved are complex but also because no feasible parallel implementations are available. The idea of introducing statistical concepts in X-Match is interesting but it is also necessary to study possible ways to improve the compression efficiency of the dictionary-based models and coders already present in the system. This analysis should also evaluate the likely impact on speed of the different solutions proposed to improve compression.

3.3.3 How can we prove the feasibility of our solutions?

System integration is also an important issue. It is necessary to produce a compression/decompression engine not only fast and efficient but also friendly to use from an application point of view. A coprocessor-style interface will make the device a sensible component to be integrated in a computer system data path. The complexity of the whole

design should not exceed that available in current hardware. ASIC's using advanced processes offer plenty of resources and high speed but they are unattractive for first silicon because of their high development costs and lack of flexibility. Recent advances in FPGA technology have produced levels of gate density and speed that enable the migration of the X-Match method to an FPGA implementation. FPGA's with densities of K hundred of gates and manufacture using advanced deep submicron processes enable a full working solution instead of just mere prototyping. The concept of desktop foundry suits our needs to prove the working characteristics of our design whilst ASIC's still remain available as an alternative if higher levels of performance and integration are required.

3.4 Conclusions

The X-Match compression method originates in the research carried out by the System Design Group at Loughborough University using partial matching CAM circuits and multiple-symbol processing to improve speed and compression efficiency. X-Match complies with the high-performance features of section 3.1 and it is therefore a suitable candidate for further research aiming to advance the current state of lossless data compression hardware.

The intention of the rest of this work is to design a general-purpose lossless data compressor coprocessor using the X-Match design as its foundation. The research studies ways to improve the compression performance and the compression throughput of lossless data compression hardware and it also studies ways to ease system integration. Working silicon will be obtained using state-of-the-art FPGA hardware. Finally, a rigorous verification methodology will be used to prove the working aspect of the design.

Chapter 4

Experimental framework

4.1 Objectives of Chapter

The objective of this chapter is to select a common development framework on which to base the experimentation. The selection must include the data sets and software/hardware lossless data compression algorithms needed to compare the compression ratio obtained by our own algorithms. It must also select the lossless data compression chips to be used to compare the throughput figures and the technology to be used to develop the hardware implementation.

4.2 Data set selection

Data set selection is always a complex issue because it is difficult to obtain a data set representative of the data that the compressor will encounter when it is deployed in an electronic system. This problem is exacerbated when the compressor is not aimed to compress a particular data type such as text or images but as a general-purpose compressor.

We have selected 3 data sets to base our experiments: the memory data set, the disc data set and the Canterbury data set.

The memory data set was assembled in the System Design Group at Loughborough University. Much of the previous research uses it so it is easier to compare the new solutions with previous work done in X-Match. The memory data set is formed by data captured directly from main memory in a UNIX workstation whilst running applications. The original data set includes around 100 Mbytes of data but it was reduced to around 10 Mbytes to have sensible processing times and memory resource requirements in some of the highly complex

context-based statistical algorithms. These state-of-the-art software-based algorithms are useful to find out the entropy or information content in the data sets and to establish a reference point. The 9 files that form the data set have the same size of 0.97 Mbytes (1 Mbyte = 1024x1024 bytes) and this corresponds to the first 1024000 bytes of data in each of the original files.

Category	No of Files	Size (Kbytes)
Xman - Unix manual page	1	1000
Text - Textedit with a small C source file open.	1	1000
Ghos - Ghoscript postscript viewer with a technical paper open.	1	1000
Emac - Emacs text editor with an elaborate set-up a few buffers open.	1	1000
Nets - Netscape world-wide-web viewer after some 'net-surfing' activity.	1	1000
Vlab - VlabPlus analogue simulator from Intergraph during extraction and spice simulation of a parallel multiplier	1	1000
Suno - Approximation to the operating system SunOS working set.	1	1000
Matl - Matlab matrix laboratory running a benchmark program.	1	1000
Logs - Logsyn logic synthesis tool from Intergraph during logic optimisation of a parallel multiplier.	1	1000
Total	9	9000

Table 4.1. Memory data set

The disc data set was also assembled in the research group. It is formed by typical data structured in 4 categories found in the hard disk of a workstation used in an engineering environment. The 4 categories correspond to: application data, executable data, general data and user data.

The application category corresponds to data required by applications to correctly function such as database files and setup files and excludes any data generated during program execution.

Category	No of Files	Size (Kbytes)
Library files from Cadence CAE system	5	1211
Component files from Intergraph CAE system	5	959
Simulation libraries from Intergraph CAE system	4	1649
VHDL libraries from Intergraph CAE system	6	785
Logic synthesis libraries from Intergraph CAE system	8	39
Matlab function libraries	6	259
Simulation libraries from Synopsys CAE system	9	1232
Parts files from Unigraphics mechanical CAD/CAM system	3	988
Data files from Visilog image processing system	2	779
Parts files from Xilinx CAE system	4	77
Total	52	8675

Table 4.2. Application disc data set

The executable category corresponds to engineering, user written and general use application.

Category	No of Files	Size (Kbytes)
General applications (ghostview, tin, xups and matlab)	4	4546
CAE applications from Intergraph and Xilinx	3	4841
System Applications (sed, awk, xcal, gtar)	4	633
User programs	5	189
Total	16	10209

Table 4.3. Executable disc data set.

The general category consists of data used by the operating system, textual files and graphical image files.

Category	No of Files	Size (Kbytes)
System font and keyboard definition files	10	863
System library files	6	758
General operating system files	7	2085
Manual pages	9	748
Documents in either postscript, html or pdf format	12	2357
ASCII text files	3	372
Total	47	7183

Table 4.4. General disc data set.

The user category consists of data created by the user such as schematic diagrams, word processing documents and results files.

Category	No of Files	Size (Kbytes)
CAE files from Intergraph and Xilinx	10	5942
Microsoft Excel spreadsheet files	3	328
Graphics files using Coredraw, Drawperfect and Microsoft powerpoint	5	1360
ASCII textual files (C and VHDL source code and a mail folder)	7	290
Results/statistics files	5	528
Word processing files from Wordperfect and Microsoft Word	4	2175
Total	34	10623

Table 4.5. User disc data set.

The Canterbury data set [Arnold97] has been recently introduced as a standard so the data compression research community can use it as a common reference. It was developed to replace the ageing Calgary data set [Bell90] and to include representative data found in modern computer systems. The authors conclude in [Arnold97] that the compression results obtained using the new Canterbury corpus can not be considered absolute measures of compression because the deviation in compression ratio if the current set of files is replaced by a bigger set of files is too high. The Canterbury corpus is, however, a useful tool for

relative measures of compression because the variations in compression using different methods are maintained if the data set is increased. The final conclusion is that the Calgary corpus and the Canterbury corpus offer very similar results so the new corpus does not invalidate the results obtained with the previous one.

Category	No of Files	Size (Kbytes)
alice29.txt - English text	1	148
pttt5 - Fax images	1	501
Fields.c - C source code	1	11.3
Kennedy.xls - Spreadsheet files	1	1003.5
Sum - SPARC executables	1	37.3
Lcet10.txt - Technical documents	1	416
Plrabn12.txt - English poetry	1	470
Cp.html - html	1	24.6
Grammar.lsp - lisp source code	1	3.72
Xargs.1 - GNU manual pages	1	4.23
Asyoulik.txt - Plays	1	126
Total	11	2745.65

Table 4.6. Canterbury data set.

4.3 Hardware selection

The hardware selection was based on using commercially available chips that offer software routines to run them on our data sets. We selected the LZS (LZ1) algorithm used in Hi/Fn devices, the DCLZ algorithm (LZ2) firstly introduced by Hewlett-Packard and now being developed by AHA and the ALDC (LZ1) algorithm from IBM. The lossless data compression chips that realise these algorithms have achieved commercial success because they combined good compression ratios and high speed. Table 4.7 shows a summary of the features of these software routines. A summary of the hardware details of the devices that implement these algorithms can be found in Table 2.2.

They are all dictionary-based compressors but this reflects that hardware statistical implementations are few and far between and none of them are closed to the throughput

requirements of Gbit/s as seen in Chapter 2. The software routines that have been used for the compression performance measurements are DOS and Windows applications that can be obtained from the web pages of the respective companies. We have used in all the cases the default configuration of the algorithm.

Name	Developer	Type	Software Command	Software Version	Software Year	Overhead
ALDC	IBM	LZ1	ENC	1.70	1993	0 bytes
LZS	STAC / Hi/fn	LZ1	LZSdemo	3.1	1992	4 bytes
DCLZ	HP / AHA	LZ2	DCLZ	2.0	1992	2 bytes

Table 4. 7. Hardware-based lossless data compression algorithms selection.

The overhead measure corresponds to algorithm identification headers added by some of the routines. One of our objectives is to measure compression performance when processing small data blocks, therefore, a header overhead should be removed to avoid distorting the compression ratios. The overheads shown in tables 4.7 and 4.8 correspond to invariable data bytes found at the start of the compressed files produced by the algorithms.

4.4 Software selection

The software selection is done because it is useful to learn how hardware compares against software in terms of compression. We selected the popular PKZIP as a representative of advanced dictionary-based software compression. We selected a state-of-the-art representative from the statistical compression world - The PPMZ algorithm review in section 2.4.1.1.1. PPMZ is considered to be one of the best lossless data compression algorithms that exists today. It combines high-order context-based modelling with an arithmetic coder. PPMZ is useful because its compression ratio is considered to be close to the theoretical limit that it is possible to obtain with lossless techniques. PPMZ throughput is very low with one byte being processed every 20 K CPU cycles. We also selected a powerful hybrid that aims to obtain the compression performance of statistical methods and the speed of dictionary-based methods – The HA algorithm. HA is a technique that combines a sliding-window dictionary plus an arithmetic coder. It is in essence a hybrid of a dictionary model plus a statistical coder. It illustrates how techniques from both domains can be successfully combined. Table 4.8 shows a summary of the main features of these algorithms.

PKZIP does not include options to tune compression performance. HA includes 3 different switches : switch 0 only copy files (no compression), switch 1 specifies a sliding window dictionary model LZ1-style plus an arithmetic coder and it is the one use in our experiments, switch 2 uses a variable 4-order model plus an arithmetic coder known as PPMC. Switch 2 defines a more powerful compression technique than switch 1 but the third selected algorithm PPMZ supersedes PPMC. PPMZ can select a model order to start making predictions with the use of LOE (Local Order Estimation). There are different coders available that specify a different maximum model order. Coder 9 is the default option in the algorithm and the one used in these experiments. It uses LOE to select a starting order to predict the next symbol that can be as high as 8th order.

Name	Developer	Type	Software Command	Software Version	Software Year	Overhead
PKZIP	PKWARE Inc	Dictionary	PKZIP	2.50	1999	14 bytes
HA	Harry Hirvola	Hybrid	HA	0.98	1993	38 bytes
PPMZ	Charles Bloom	Statistical	PPMZ	9.1	1997	28 bytes

Table 4.8. Software-based lossless data compression algorithms selection.

4.5 Technology selection

The reduction in feature size and constant advances in the manufacture process have made FPGA technology get closer than ever to ASIC performance allowing the migration of whole systems to a single chip. The development of the prototype core is based on ProASIC FPGA [Actel00] technology from Actel/Gatefield corporation. The reason to choose this technology is partly found on resource availability and also on particular interesting features present in their new non-volatile Flash-based ProASIC devices that suit the flipflop-rich X-Match architecture better than other RAM-based devices. We also selected the Apex [Altera01] and Virtex [Xilinx01] family recently introduced by Altera and Xilinx Corporations respectively with densities in the order of million of gates and 0.18 μm feature size. Table 4.9 shows a summary of these technologies.

Technology		Density (Highest in family)				
Manufacture	Process	Family (Device)	System gates(k)	Typical gates(k)	RAM bits (k)	Logic elements
Gatefield / Actel	FLASH-CMOS 0.25 μm	A500K (A500K510)	1,100	410	138	51,200
Altera	SRAM-CMOS 0.18 μm	APEX20K (EP20K1500E)	2,392	1,500	432	51,840
Xilinx	SRAM-CMOS 0.18 μm	VIRTEX (XCV3200E)	4,074	Not stated	851	73,008

Table 4.9. Technology selection.

4.6 Measurement definitions

Compression ratio (CR): Compression ratio is defined as the ratio $CR = \text{output_bits}/\text{input_bits}$ in the algorithm. This means that the smaller the figure the better the compression. A value larger than 1 implies that data expansion but not data compression took place. Compression is obtained whenever the CR value is in the range (0,1). For example if the $CR = 0.5$ means that 100 Mbytes of uncompressed input data are compressed to 50 Mbytes of compressed data.

Compression gain (CG): Compression gain of algorithm b over algorithm a is a percentage defined as the value $CG = 100*(CR_a - CR_b)/CR_b$. This means that the bigger the number the higher the compression improvement and that a negative value brings compression degradation. For example if an algorithm b has a $CR = 0.25$ and algorithm a has a $CR = 0.5$ the $CG = 100*(0.5-0.25)/0.25 = 100\%$ better compression of b over a .

Block size (BS): Different input block sizes are used to evaluate the performance of the compression algorithms as function of the amount of data to be compressed as an independent block. The experimental methodology uses the following block sizes: 256 bytes, 1K, 4K, 16K, File.

Dictionary size (DS): Different dictionary sizes are used to evaluate the trade off between complexity and compression performance of the algorithms. The experimental methodology uses the following dictionary sizes: 16, 32, 64, 256, 512, 1024.

4.7 Conclusions

This chapter has selected a set of tools to help to carry out the experimental work of chapters 5,6 and 7.

- To measure the compression performance we have selected 3 different data sets for a total of 48 Mbytes of data.
- To compare hardware-based performance we have selected 3 commercially available high-performance lossless data compression chips.
- To reference compression performance we have selected 3 state-of-the-art software-based lossless data compression algorithms.
- To develop our hardware implementation we have selected 3 state-of-the-art FPGA technologies.

These selections together with the measurement definitions will be used in the following 3 chapters that deal with improving the compression efficiency, improving the throughput and finally proving that the solution proposed is feasible and meets the requirements introduced in chapter 1.

Chapter 5

Focus on compression efficiency

5.1 Objectives of Chapter

This chapter deals with the problem of improving compression efficiency whilst maintaining a high throughput. These 2 variables are strongly related since it is usually possible to improve compression by using more complex modelling and coding techniques but this extra complexity has a negative effect on speed. It is also true that simplifying the algorithm tends to enable higher operational speeds. A trivial example is to think of a system that copies directly the input to the output. The speed of such a method could be considered optimal and impossible to improve upon. The compression ratio of the system would be 1.0 and it will be of no use from a compression point of view.

The main body of results are reported based on a single corpus. The Canterbury corpus has been selected because its small size enables fast execution of some of the more complex algorithms. The data mixture that forms the Canterbury corpus is accepted as representative of the data types found in modern computer systems. The final results will be validated using the other 2 corpuses introduced in chapter 4: the memory data set and the disc data set.

Our final aim is to produce a feasible architecture ready to be implemented in current or soon to be available hardware. This means that although the main objective of this chapter is to produce algorithmic techniques that improve the compression performance of X-Match complexity cannot be disregarded. The final algorithm must not only be computationally feasible but it must also be hardware amenable.

5.2 X-Match compression efficiency

The original X-Match algorithm was described in chapter 3. This section analyses its compression efficiency on the Canterbury corpus. The hardware design uses a dictionary size of 128 locations. The original software version of X-Match uses a dictionary with 1024 locations. Dictionary size can vary with minimal impact on speed but important effects on complexity and compression ratio.

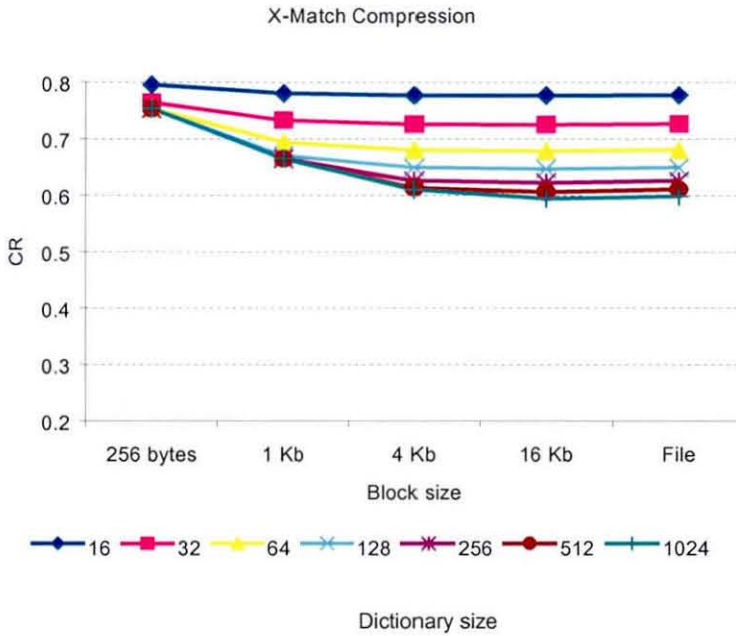


Figure 5.1. X-Match compression on the Canterbury corpus.

From Figure 5.1 it is clear that the compression efficiency of X-Match on the Canterbury corpus is modest. One of the main reasons is that this corpus contains a large amount of textual data. There are 11 files in the Canterbury corpus. The *txt* extension is present in 4 of these whilst the other 4 have also a textual nature such as *html*, *C* or *Lisp* source code. Textual data is heavily byte oriented and the rationality of processing groups of 4 bytes together does not hold.

Figure 5.1 also shows that the compression efficiency of X-Match grows with dictionary size and block size. A larger dictionary increases the chances of having a match in one of its locations. Figure 5.1 shows that for any block size compression increases or remains the same if the dictionary size increases. If the block size has only 256 bytes a dictionary size of 64

locations is enough to store the whole block internally and further increases in dictionary size do not improve compression.

A larger block size means that more data is available to build the dictionary and in consequence better modelling of the input data source can be achieved. This is particularly true if a large block size is combined with a large dictionary size. Small dictionaries saturate quickly and this limits their capacity to adapt the extra data available in large block sizes. Figure 5.1 illustrates this effect with a 16 location dictionary whose compression remains largely invariant with increases in block size. Figure 5.1 also shows that maintaining the same dictionary size and increasing the block size always improves compression except in file-based compression where some minor degradation can take place. The reason is that the combination of PBC plus periodically resetting the dictionary can have a positive effect on compression. The effect of PBC in X-Match is that the dictionary always starts with an empty state to compress a block so only a few bits are needed to code a partially full dictionary. This is useful when processing small data blocks but in file-based compression the effect is negligible because once the dictionary is full all the locations need to be assigned a code. The periodical resetting of the dictionary, that is equivalent to breaking the file in smaller blocks, reactivates the PBC strategy and can improve compression because it increases the adaptability of the model to the local characteristics of the input data source. This effect is called *locality of reference* [Bentley86]. It means that in a typical data block a symbol can be heavily used in a block section but then it can fall in disuse in another block section.

5.3 Dictionary-based approach

5.3.1 Introduction

The dictionary-based approach investigates how the dictionary-based models and coders presented in the X-Match method can be improved to obtain better compression whilst maintaining the high throughput.

5.3.2 The dictionary-based model

The dictionary-based model of X-Match uses a CAM that stores the last 128 tuples (1 tuple = 4 bytes) as its compression history. The move-to-front (MTF) replacement policy is a least-recently-used (LRU) policy that removes from the dictionary the tuple that was used less recently. This technique in effect forms a sliding window of history data that moves over the

input data source but it avoids data duplication at different locations of the history window. This policy tends to be more effective than a least-frequently-used (LFU) policy at exploiting the *locality of reference* effect. A typical inefficiency of LFU is that it could assign a high frequency to a popular tuple during the compression of the first section of a block of data but then the same tuple could fall in disuse during the second section. This tuple remains in the dictionary because it achieved a high count during the initial stage but since it is not used during the second stage it wastes coding space. Combinatorial searching strategies do not improve compression because the extra bits added to the output to distinguish which combination matches offset the extra number of matches [Gooch96]. Better modelling can be achieved by increasing the CAM-size so a larger compression history is maintained. The 128 position CAM represents already 70% of the logic in the chip so the complexity implication of using larger CAM's must be taken into account.

5.3.3 The dictionary-based coder/decoder

5.3.3.1 Introduction

X-Match uses a static Huffman coder to code the match types and a phased binary coder to code the match locations. To code the bytes that are not found in the dictionary X-Match does not use any coding technique but instead the bytes are added to the codeword in literal form. These bytes could also be coded to improve compression but parallel decoding will be then very difficult to implement in hardware. Since the lengths of the individual codes are not known in advance multiple decoders should decode in parallel all the possible length combinations of the 4 coded bytes. A typical Huffman code generates 7 different lengths for a 256-symbol alphabet so a total of $7^0+7^1+7^2+7^3=400$ independent decoders are needed. The first decoder decodes the first symbol. The next 7 decoders decode 7 possible symbols depending on the first symbol. The next 49 decoders deal with the third symbol and the final 343 decoders deal with the fourth symbol. The technique is unfeasible because of its scaling complexity.

5.3.3.2 Match location coding techniques

Phased binary coding is a technique used to code the dictionary locations of a dictionary that starts empty and then it grows accommodating new data found in the block being compressed. The advantage is that a smaller dictionary uses fewer bits to code its positions so there is a compression gain during the growing stage. This advantage is lost once the dictionary becomes full after a number cycles. The number of cycles that it takes before the dictionary

fills depends on its maximum size and the redundancy of the original data since only tuples that are not fully matched increase the size of the dictionary. This is done to maintain a high dictionary efficiency because each location stores unique data. If the dictionary size is small the gains obtained with PBC are negligible because the dictionary fills very quickly. This means that a simpler form of coding such as uniform binary coding (UBC) where every position uses $\log_2(\text{dictionary size})$ bits can be used. Figure 5.5 shows the compression gain (defined in section 4.6) obtained by PBC compared against an alternative using UBC processing the Canterbury corpus.

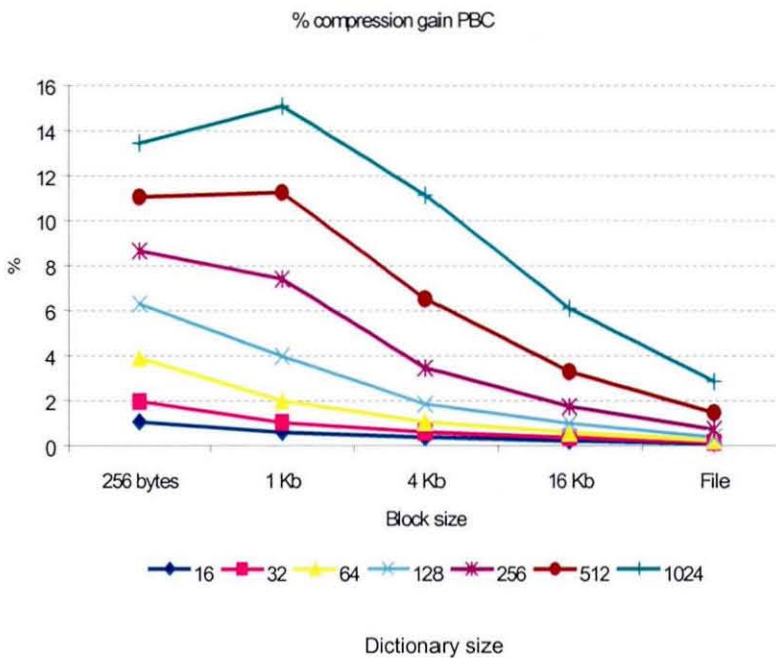


Figure 5.5. Compression gain PBC versus UBC in X-Match processing the Canterbury corpus.

PBC shows itself as a useful technique mainly when coding small to medium size block sizes and using large dictionaries. A large dictionary uses a large number of bits to code its locations if all them are active from the start. If the block size is small these locations remain empty because there is not enough data to fill the dictionary but they waste coding space because dictionary addresses remain assigned to them. This is the reason the compression gain obtained by PBC against UBC is so significant with small block sizes and large dictionaries. For a typical block size of 4 Kb PBC does not show any significant advantage until a dictionary larger than 64 locations is used. The reason is that a small dictionary fills quickly and once it is full there is no different between UBC and PBC.

An alternative to PBC for the match locations is to use a Huffman code based on the fact that the LRU maintenance policy forces more common data to be stored closer to the top of the dictionary. The probability of having a match in these locations is higher than the locations closer to the bottom. A static Huffman tree can be designed based on this property with dictionary locations closer to the top of the dictionary also closer to the root of the Huffman tree. The match frequency distribution in the dictionary has been used to generate Huffman trees varying the dictionary size. For a typical dictionary size of 256 elements the Huffman codes varied in size from 2 bits for location 0 to 12 bits for location 255. Figure 5.6 shows the tree shapes obtained after processing our data sets.

The shape of the trees remains largely invariant independently with the data sets because all the compressible data sets exhibit *locality of reference* that increases the probability of having matches closer to the top of the dictionary. A static Huffman code is a good solution to code this match distribution because the tree shape is largely data independent. An adaptive technique will not provide any significant advantage.



Figure 5.6. Huffman tree shapes.

Figure 5.6 shows that the 3 different data sets produce similar tree shapes. The disc data set shape is obtained as an average among the 4 data set components: executable, general, application and user. This is the reason why the disc ‘shape’ exhibits some noise.

Figure 5.7 shows the compression gain obtained by X-match when using a Huffman code for the match locations.

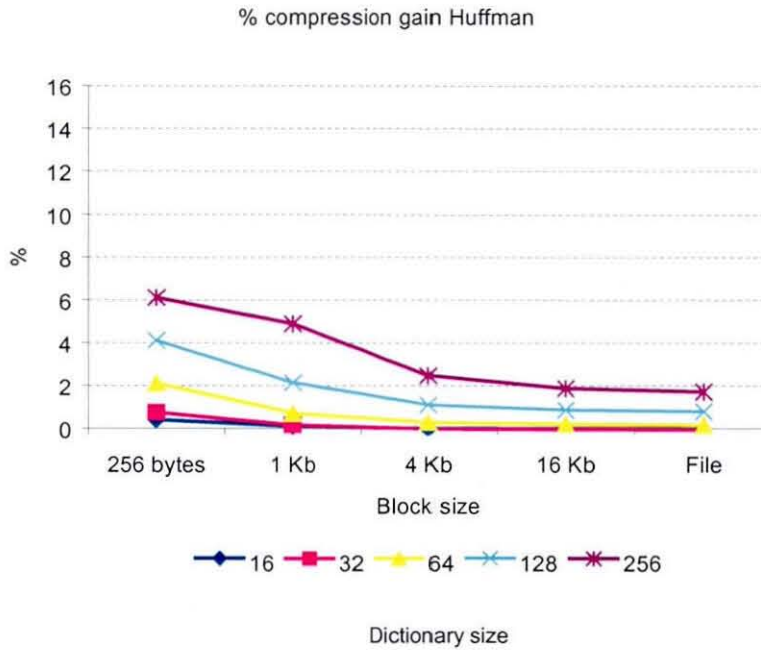


Figure 5.7. Compression gain Huffman versus UBC in X-Match processing the Canterbury corpus.

The dictionary size is limited to 256 locations because the tree generator used in the experiments cannot handle trees larger than 256 leaves. The largest code in this case is 12 bits which means that an already complex 4096 positions look-up table is needed for the decoding. Figure 5.5 and Figure 5.7 show that Huffman coding offers a small compression gain over PBC when handling large block sizes but PBC is more efficient when compressing small blocks. PBC is also simpler since no large look-up tables are needed. Huffman coding like PBC also needs a dictionary size larger than 64 locations to offer any significant gain.

A combination of the concepts of both Huffman coding and PBC creates a Phased Huffman Coder (PHC). This implementation uses a growing dictionary and a number of Huffman trees equal to $\log_2(\text{dictionary size})$. The dictionary grows in powers of 2 and depending on how many dictionary entries are valid a different tree is used. The rationale is to have the good performance of PBC with smaller block sizes and Huffman coding with large block sizes respectively. Extra complexity is added in the coding and decoding processes because the system must store a number of Huffman coding and decoding look-up tables and efficiently switch among them when the next dictionary size is activated. Figure 5.8 shows the compression gain of PHC.

The phased Huffman coder offers better compression than Huffman coding for the 256 bytes block sizes and also better compression than PBC for file-based compression. On the other

hand its performance is inferior to PBC for any practical block size smaller than 16 Kbytes. The complexity impact of PHC is also considerable because for a 256 location dictionary it is necessary to store and manage 8 independent Huffman trees.

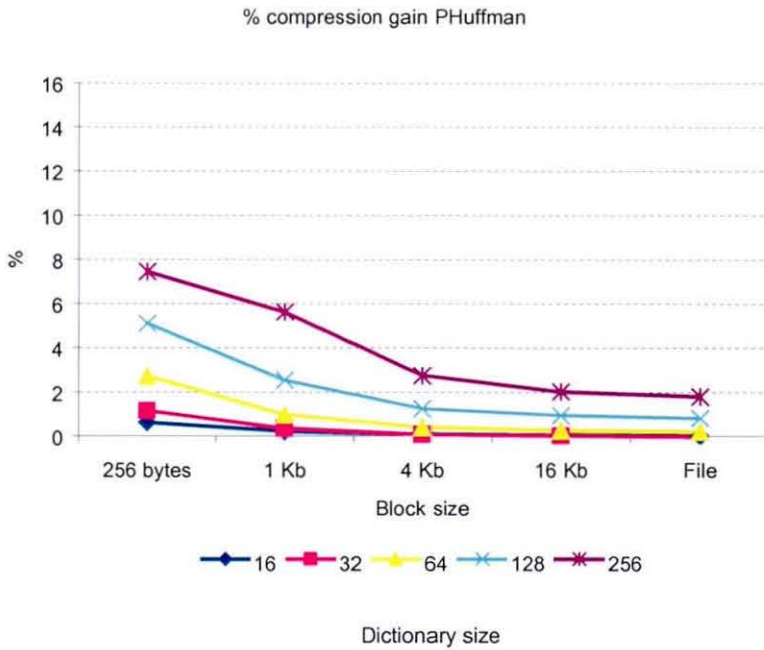


Figure 5.8. Compression gain PHuffman versus UBC in X-Match processing the Canterbury corpus.

This concept of a phased Huffman coder is closed to other alternatives such as the hardware amenable Rice coding of section 2.4.1.2.1.3. Rice codes are in essence Huffman codes that can be adjusted using a parameter that modifies the shape of the tree. This parameter would be controlled by the dictionary size in our case. Rice codes offer less flexibility than Huffman codes because the alphabet size is unbounded and their performance is limited. The alphabet size is unbounded because a maximum size it is not defined when constructing the Rice code.

5.3.3.3 Run-length coding techniques

Run length coding can also be used to improve the coding efficiency of the dictionary-based coder. Run length coding is based on signalling repetitive patterns and code them together indicating the pattern that was found and how many times it repeated. The effect of a run length coding applied to X-Match is to code repeating patterns of 32 bits in a single code because of its tuple granularity. Several solutions are possible depending on where the run length coder is placed relative to the other functions present in the algorithm.

In principle there are 3 different locations for a run length coder in the X-Match algorithm depending on what sorts of runs it aims to code: front (Run Length Front : RLF), middle (Run Length Internal : RLI), back (Run Length Back : RLB). To add a run length coder to the back of the algorithm is not a sensible option for a simple reason: the effect of compression is to produce a randomised output where run lengths are non-existent. After packing the codewords get disaligned so a run length coder will be unable to detect 32 bit repeating aligned patterns. We will now study the other 2 options: RLF and RLI.

Our RLF alternative is sensitive to 32-bit repeating patterns of the same byte for example 'aaaa'. This way only one byte is needed in the RLF code to know which byte was repeating ('a'). It is necessary to determine which is the minimum run length that must activate the RLF technique. Following the X-Match algorithm described in chapter 3 and assuming a maximum run of 255 repetitions the output generated by a run of length 2 will be: 1 bit for the match, $\log_2(\text{dictionary size})$ bits for the RLF code, 8 bits for the repeating byte and 8 bits to indicate the length. The total is $17 + \log_2(\text{dictionary size})$ bits.

A non-RL output when data is not in the dictionary will be an initial miss of 33 bits and a full match of $3 + \log_2(\text{dictionary size})$ bits (1 bit match, $\log_2(\text{dictionary size})$ bits match location, 2 bits match type). If we assume a practical dictionary size of 256 locations RLF improves compression because it outputs 25 bits and the non-RL output is 44 bits. On the other hand if the data is indeed in the dictionary RLF will fail to improve compression because it will still output $17 + \log_2(\text{dictionary_size})$ bits whilst a non-RL alternative will output 2 codes of $3 + \log_2(\text{dictionary_size})$ bits = $6 + 2\log_2(\text{dictionary_size})$ bits. For a practical dictionary size of 256 locations this is 25 bits > 22 bits. Therefore depending on how often data is found in the dictionary RLF can be made sensitive to runs of length 2 or not. Figure 5.9 shows the distribution of full matches, partial matches and misses over the Canterbury corpus with different block sizes and a practical dictionary size of 256 locations.

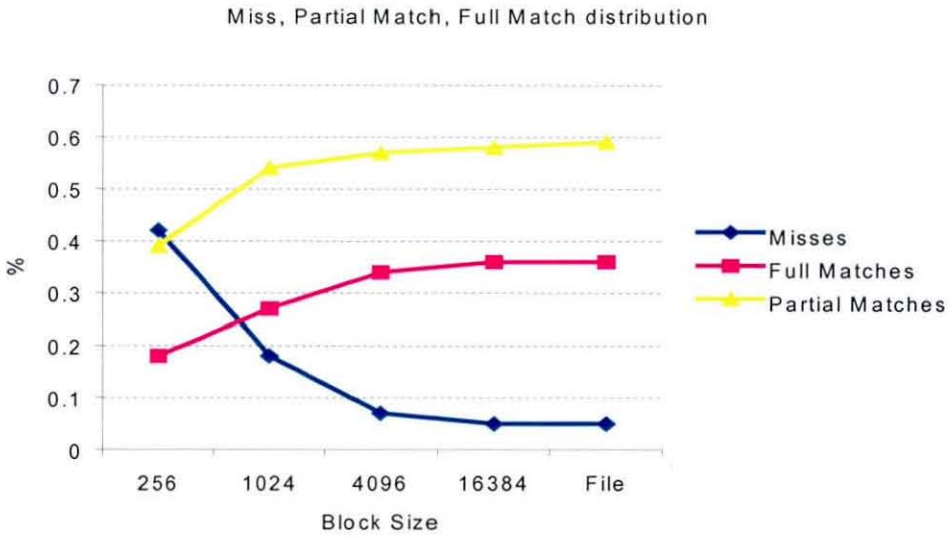


Figure 5.9. Distribution of Misses, partial matches and full matches over the Canterbury corpus.

Assuming a worst case for the run length option we select a distribution of 0.05% misses, 0.36% full matches and 0.59% partial matches that corresponds to a block size of 16 Kbytes or bigger. Then, we can compare the number of bits produce by a RLF alternative sensitive and non-sensitive to runs of length 2. We call $x = \log_2(\text{dictionary size})$ and we assume steady state. A full match outputs a match location code of x bits, a single bit indicating a match and 2 bits indicating a full match type. A miss outputs a single bit indicating a miss and 32 bits of literal characters. The number of bits produced by a partial match depends on the type of match. For the calculation we assume again a worst case for run length so the number of bits produced by the partial match is minimum: 1 bit for the match, x bits for the match location code, 3 bits for the match type and only 1 missing byte added = 12 bits.

$$\begin{aligned} \text{No of bits (non-sensitive)} &= \\ &= 0.05(33) + 0.36(3+x) + 0.59(12+x) + (3+x) = 1.95x + 12.81 \end{aligned} \quad [5.1]$$

$$\text{No of bits (sensitive)} = 17+x \quad [5.2]$$

$$12.81 + 1.95x > 17+x \Rightarrow x > 4.41 > 5 \Rightarrow$$

$$\Rightarrow \text{No of bits(non-sensitive)} > \text{No of bits(sensitive)} \quad [5.3]$$

From equation [5.3] RLF can be made sensitive to repetitions of length 2 if the dictionary size is larger than $2^5 = 32$. Equation [5.1] depicts a worst case for RLF because it measures the minimum number of bits produced by a non-RL alternative. It assumes a miss probability based on a 256-location dictionary however if the dictionary is smaller than 256 locations the

percentage of misses increases and therefore the number of bits in equation [5.1] increases reinforcing the result of equation [5.3]. Equation [5.1] also assumes that all the partial matches are matches of 3 bytes and they only need one missing byte to be added to the code. If partial matches of only 2 bytes are included in the calculation the number of bits produced in equation [5.1] increases again reinforcing the result of equation [5.3].

We will now study the RLI alternative. RLI combines with MTF to efficiently run length code any repeating 32 bit pattern. Since the MTF dictionary maintenance policy forces any repeating pattern to be located at position 0 (top of dictionary), RLI detects and run length codes any tuple that is fully matched at the top of the dictionary 2 or more times. The tuple always has to be present in the dictionary in location 0 for the RLI event to become active because RLI codes runs of full match at location 0 and not runs of repeating tuples. This means that the first tuple in the input data source that starts a run of repetitions is stored in location 0 and only the following repeating tuples can be coded as part of a RLI event. The output of a RLI code is always $9 + \log_2(\text{dictionary_size})$ (1 bit indicating a match, $\log_2(\text{dictionary_size})$ bits for the RLI code, 8 bits for the run length). The following set of equations is obtained comparing the output produce by a RLI sensitive and non-sensitive to repetitions of length 2.

$$\text{No of bits (non-sensitive)} = 6 + 2x \quad [5.4]$$

$$\text{No of bits (sensitive)} = 9 + x \quad [5.5]$$

$$6 + 2x > 9 + x \Rightarrow x > 3 \Rightarrow \text{No of bits(non-sensitive)} > \text{No of bits(sensitive)} \quad [5.6]$$

From equation [5.6] RLI can be made sensitive to repetitions of length 2 because it saves bits for any dictionary size bigger than $2^3 = 8$.

The most common repeating pattern (in our experience) is a run of zeros, however other repeating patterns also exist like the space character in a text file or a constant background in a picture. This situation is illustrated in Figure 5.9 that shows an accumulative distribution of run lengths. The X axis is the repetition length of the run while the Y axis is an accumulative distribution that specifies a repetition length frequency.

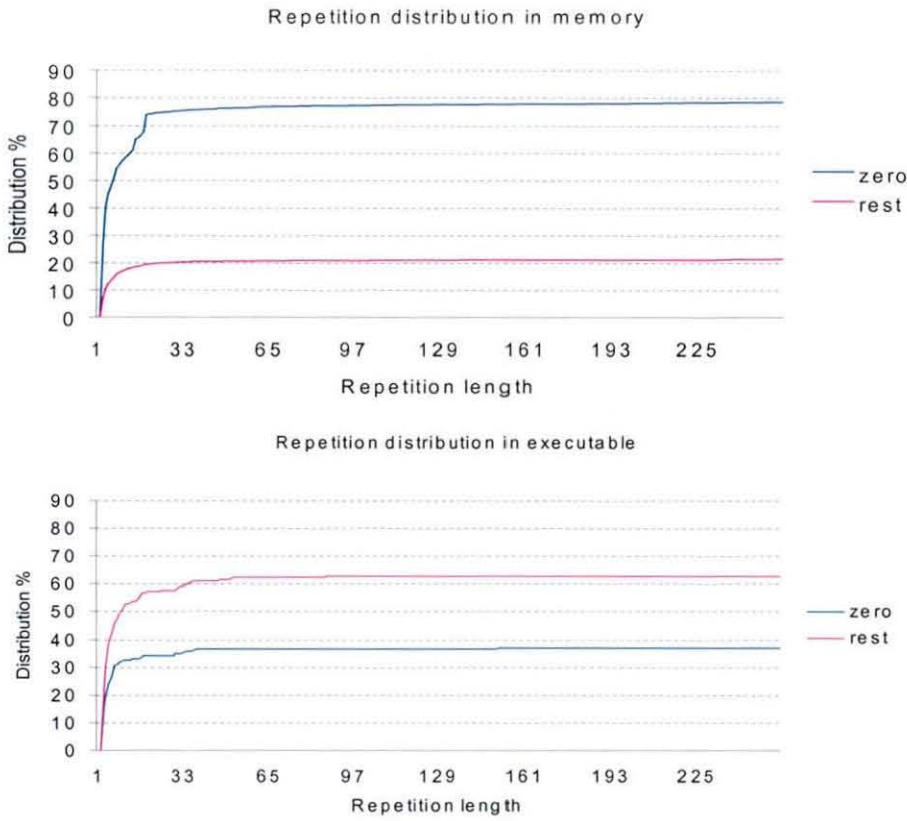


Figure 5.9. Repetition distribution on data sets.

The results of the analysis of the repetitiveness of 32-bit patterns in 2 of our data sets show that most of the runs are of length 15 or less whilst the contribution made by longer runs is small. Figure 5.9 shows that the distribution line stops growing around a value of repetition length around 15. On the other hand long runs offer more compression advantage because more bits are coded in a single codeword. The memory data set in Figure 5.9 is formed by around 9 Mbytes of data obtained from the main memory of a workstation used in an engineering environment. The most common event that uses RLI codes is the tuple formed by 32 zeros although other patterns account for 20% of the RLI codes. The executable data set in Figure 5.9 is formed by 35 Mbytes of executable data files found in the hard disk of the same workstation this situation is inverted an non-zero 32-bit repeating patterns are predominant with 60% of the total.

A disadvantage of RLI is that it requires at least a repetition length of value 3 to be activated because the first tuple is used to place the match location at position 0. RLI is only sensible to matches at location 0 unaware of the data that generates the match. On the other hand a RLF

(Run Length Front) located at the entrance of the dictionary could be activated with only 2 repetitions.

Both techniques RLI and RLF are experts at different data sources. RLI is activated by any 32-bit repeating pattern of length minimum 3 while RLF codes any 32-bit repeating pattern of the same byte of length minimum 2.

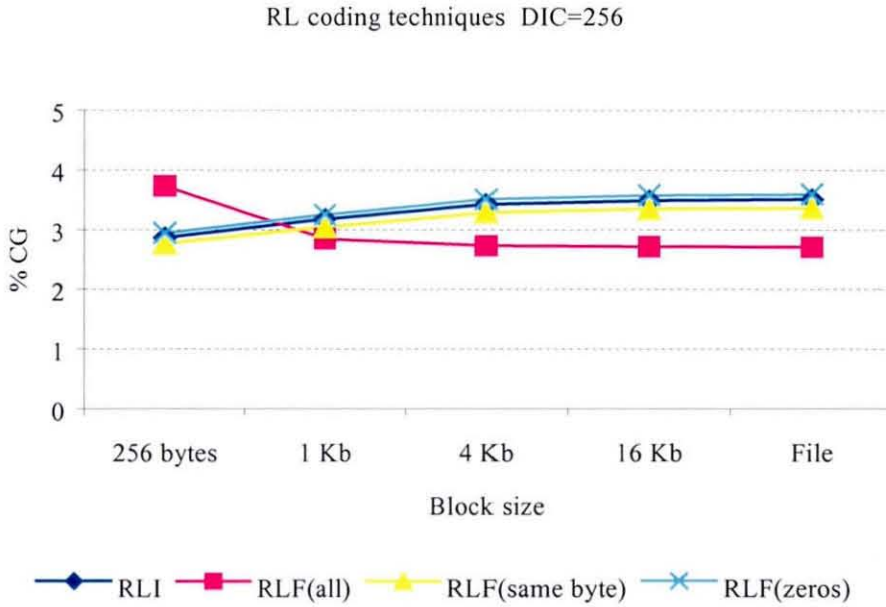


Figure 5.10. Run length coding techniques in X-Match processing the Canterbury corpus.

Figure 5.10 shows the compression gain of these RL coding techniques applied to X-Match versus a non-RL alternative over the Canterbury corpus. We use a typical dictionary size of 256 locations. RLF(same byte) is the technique described previously that needs 1 byte to identify the repeating tuple. We have include 2 variants of the run length front technique of the same byte for completeness: RLF(zeros) is only sensitive to repetitions of zeros, so no extra byte is needed to indicate the repeating tuple. RLF (all) is sensitive to repetitions of any byte like RLI but it needs to have the whole repeating tuple (4 bytes) added to the RLF code so the decoder knows which tuple originated the run.

From Figure 5.10 RLF(zeros), RLF(same byte) and RLI offer the best results. RLF(all) has a better performance for very small block sizes but its performance degrades with larger blocks. The compression gain is not very significant because the Canterbury corpus is textual bias and it does not contain the long runs typical of binary data. As we will see in the following sections performance improves with the other 2 data sets. We have chosen RLI to be developed in hardware because it integrates neatly in the X-Match architecture and shares the

dictionary logic to perform the comparisons keeping complexity to a minimum. Figure 5.11 shows the effects of dictionary length on run length coding efficiency for the RLI alternative.

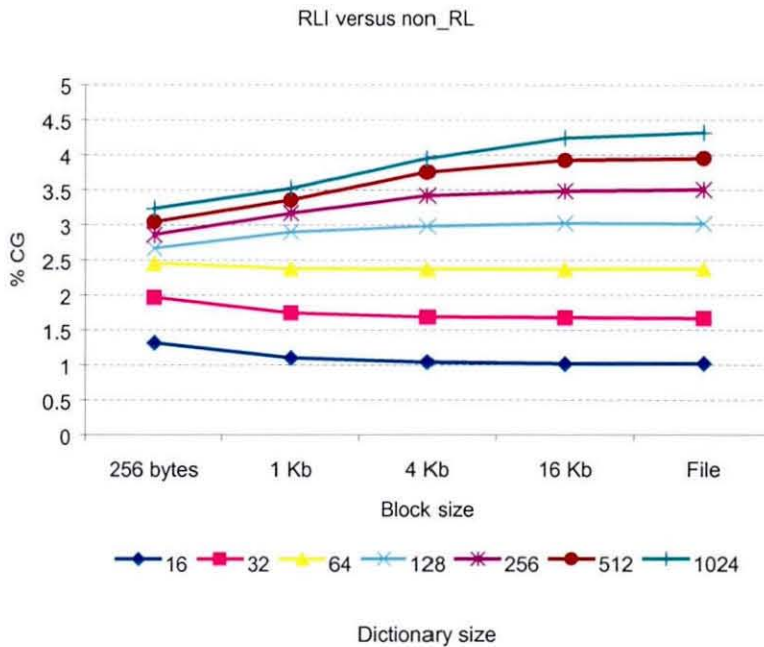


Figure 5.11. Compression gain of RLI versus non-RL in X-Match processing the Canterbury corpus.

From Figure 5.11 it is clear that the effectiveness of run length coding improves with dictionary size but is largely invariant with block size. The maximum number of repetitions that can be coded together is 255 in this implementation. We have found that this offers the best compression. Using 7 or 9 bit counters damages compression. A RLI code is coded as a 0 indicating a match followed by the binary code corresponding to the last location in the dictionary and followed by an 8-bit code with the number of repetitions. This means that the dictionary reserves one location to code RLI events and consequently has one word less to store frequent 32-bit vectors. This is one of the reasons that justify why RLI works better with large dictionaries because the effect of losing one dictionary location has a more significant impact on compression with small dictionaries. It is also true that a large dictionary has a better compression ratio and in consequence the compression gain obtained by RLI measure as a percentile improvement ($100 \times (CR_{\text{before}} - CR_{\text{after}}) / CR_{\text{after}}$) is more noticeable when CR is small. Assuming a dictionary size of 256 locations a maximum compression ratio of $17 / (255 \times 4 \times 8) = 0.002$ is enabled by the RLI module when a full run of 255 repetitions is encountered. The maximum compression ratio achievable by X-Match without RLI is limited to $11 / (4 \times 8) = 0.34$.

5.3.3.4 Conclusions

PBC seems to be the best overall solution for the dictionary-based match location coder because it combines good performance on small to medium block sizes and simple implementation. PBC should be used if the dictionary size is larger than 64 locations to avoid damaging compression when processing small data blocks. If this is not the case, simpler UBC will suffice to implement the match location coder. RLI offers a compression gain over a non-RL alternative with minimum investment on extra complexity because only a counting mechanism and a way of detecting full matches at location zero are needed to enable the technique. RLI can be used with any dictionary size but its efficiency improves with dictionary size. A dictionary length of 256 offers a good trade-off complexity/performance and it effectively uses the PBC and RLI techniques. Figure 5.12 shows that the performance improvement in X-RLI with 512 and 1024 dictionary locations is within a narrow margin of the 256 dictionary locations solution. A dictionary size of 256 is therefore selected for the compression performance measurements of section 5.5.

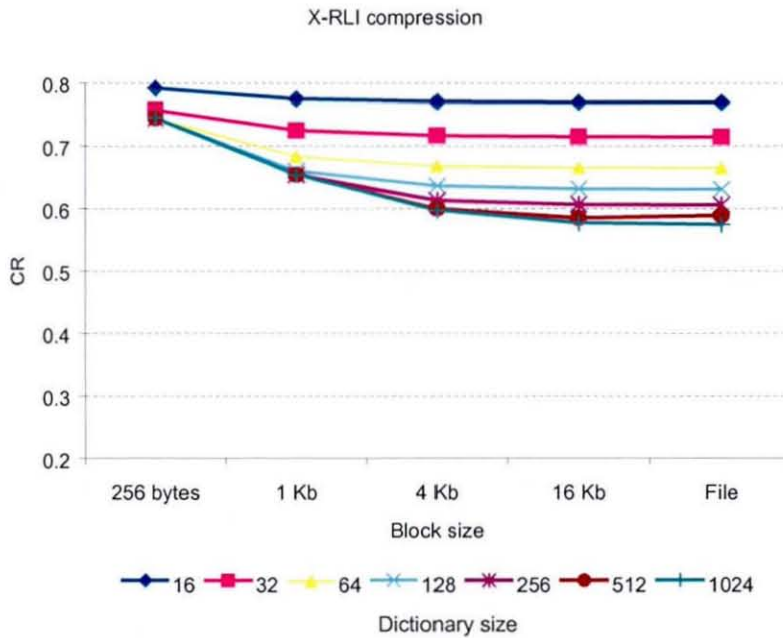


Figure 5.12. X-RLI compression on the Canterbury corpus.

Figure 5.13 shows the new algorithm named X-RLI in pseudo-code format that uses PBC and RLI. The instructions shown in bold letter are not present in the original X-Match algorithm.

```

Set the dictionary to its initial state;
Set next free location counter = 0;
Run length count = 0;
DO
{
    read in tuple T from the data stream;
    search the dictionary for tuple T;
    IF ( full hit at location zero)
        increment run length count by one;
    ELSE
    {
        IF ( run length count = 1)
        {
            output '0';
            output phased binary code for ML 0;
            output Huffman code for MT 0;
        }
        IF ( run length count > 1)
        {
            output '0';
            output phased binary code for ML MAX_TABLE_ENTRIES-1;
            output Binary code for run length;
        }
        set run length count to 0;
        IF (full or partial hit)
        {
            determine the best match location ML and the match type MT;
            output '0';
            output phased binary code code for ML;
            output Huffman code for MT;
            output any required literal characters of T;
        }
        ELSE
        {
            output '1';
            output tuple T;
        }
    }
    IF (full hit)
        move dictionary entries 0 to ML-1 by one location;
    ELSE
    {
        move all dictionary entries down by one location;
        increase next free location counter by one;
    }
    copy tuple T to dictionary location 0;
}
WHILE (more data is to be compressed);

```

Figure 5.13. The X-RLI algorithm

5.4 Compression performance comparison

This section analyses the compression performance of the X-RLI algorithm of section 5.4.3.4 with a dictionary size of 256 locations. We test the compression performance of this algorithm against the software and hardware based algorithms selected in sections 4.3 and 4.4 using the data sets selected in section 4.2. We have also included the original PBC-based X-Match extended to a dictionary size of 256 locations for the sake of completeness.

5.4.1 Canterbury data set compression performance comparison

Figure 5.14 shows the compression ratios achieved by our software and hardware based compression algorithms on the Canterbury corpus. We can clearly identify 3 main areas.

The 3 software-based compression algorithms offer the best compression with similar results. PPMZ is the top performance whilst the behaviour of the algorithms HA and PKZIP is remarkably similar once the block size reaches 1 Kb. These 2 algorithms use a similar dictionary-based sliding-window modelling technique but HA uses an arithmetic coder as the back-end of the algorithm. HA only manages to improve PKZIP marginally. We have removed the overhead effects by deleting the bytes that do not form part of the compressed code as described in section 4.3.

The 3 commercially available hardware algorithms offer very similar performance whilst X-RLI falls behind. The textual nature of the Canterbury corpus can explain the limited performance of X-RLI over this type of data.

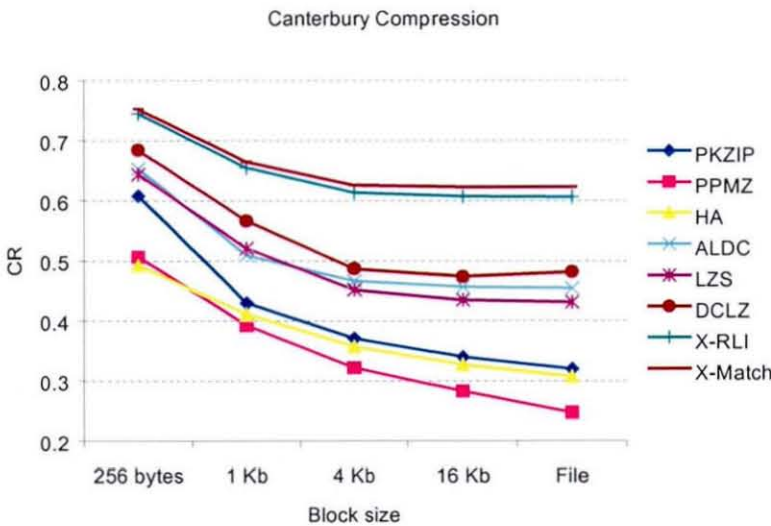


Figure 5.14. Canterbury corpus compression performance.

5.4.2 Disc data set compression performance comparison

Figure 5.15 shows the compression performance on the disc data set. The performance of the DCLZ and X-RLI algorithms is quite similar mainly when dealing with small block sizes. The performance of the ALDC and LZS algorithms is superior to the previous ones and similar between them. The disc data set has 4 components and as expected the performance of X-RLI was particularly good when compressing the executable component of the data set due to its 32-bit granularity.

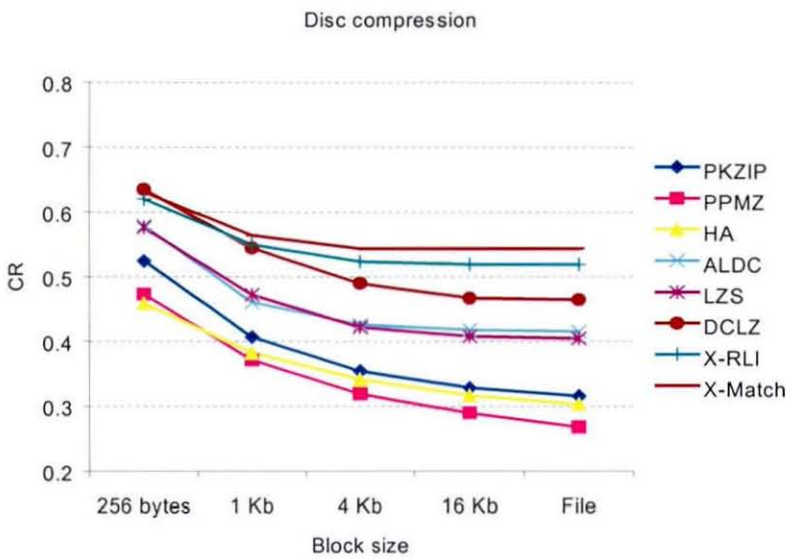


Figure 5.15. Disc data set compression performance.

5.4.3 Memory data set compression performance comparison

Figure 5.16 shows the performance of the algorithms on the memory data set.

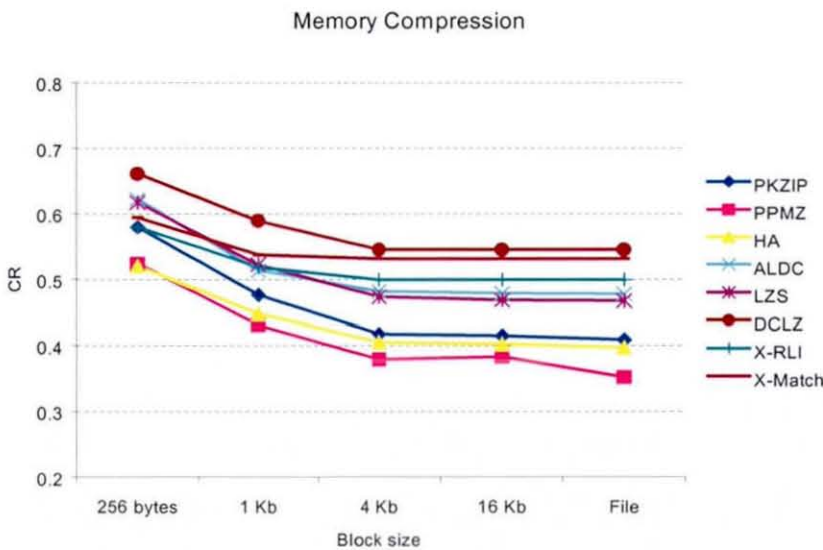


Figure 5.16. Memory data set compression performance.

There are again 3 areas in Figure 5.16 although there are closer to each other than in the other 2 data sets. The 3 software algorithms offer very similar performance whilst DCZL gives the worst results. X-RLI is competitive with the ALDC and LZS versions and performs better with smaller block sizes. The memory data set is formed with data found in the main memory of a workstation running different applications. This binary data is well suited to X-RLI compression because it has a 32-bit granularity not present in the other data sets. The new RLI technique include in X-RLI proves more useful in the disc data set and memory data set where the compression improvement of X-RLI over X-Match is more noticeable than in the case of the Canterbury corpus.

5.5 Conclusions

This chapter has proposed a number of techniques to improve the X-Match compression performance. Increasing dictionary size and introducing an internal run length coding technique can improve the dictionary-based modeller and coder without affecting speed. We select the X-RLI algorithm for further research because it is able to obtain meaningful compression ratio in our data sets, it is hardware amenable and it has a parallel single-cycle execution that enables the throughputs required in section 3.1.

Chapter 6

Focus on compression throughput

6.1 Objectives of Chapter

This chapter deals with the issue of increasing the throughput of the design whilst maintaining the compression ratio. High throughput was identified as one of the main motivations to undertake this research so the outcome of this chapter is fundamental.

6.2 Introduction

The 3 elements that form part of the compression/decompression engine: Model, Coder/Decoder, Packer/Unpacker have a direct effect on throughput. The method followed to improve the X-Match performance consists of a rigorous analysis of each of these components to solve possible bottlenecks present in the architecture. Our design methodology has accessed the structural VHDL description of the original blocks present in X-Match [Gooch96]. The redesign architecture is described in VHDL using a structural and hierarchical approach to obtain a more predictable output from the FPGA-based synthesis engines used to synthesise and map the VHDL to a technology-dependant netlist. The reports provided by these tools are used to guide the optimisation process. To be able to validate our solutions in hardware we use the ProASIC FPGA's manufactured by Actel corporation as our silicon test bench. The experimental methodology is based around a dictionary size of 16 tuples to be able to target the A500K130 ProASIC FPGA for rapid prototyping. The ProASIC A500K130 is one of the first devices to become available in the high-density A500K family. This device constitutes an invaluable tool to validate our designs. Its ASIC-style architecture, re-programmability and non-volatility features couple with its test capabilities enable us to

test in hardware circuits that otherwise will be only proven through simulation. The architecture of the new engine is described using PBC because chapter 5 proved that this coding strategy is a valuable technique to improve the performance of medium and big dictionaries. The prototype is based on a dictionary with 16 locations to be able to target the available A500K130 device. Chapter 5 showed that a 16,32 and 64-tuple dictionaries do not benefit from PBC so, in order to further reduce the resource requirements in the A500K130 FPGA, the prototype implemented in section 6.6 to evaluate throughput uses simpler UBC for the match locations. This is a compromise we need to make to validate our design in the available silicon. The design scales up easily with technology and the modifications needed to add PBC when the technology density enables the use of dictionaries larger than 64 locations are small, as we will see in the next sections.

6.3 Model architecture

Figure 6.1 shows the architecture of the model in the X-Match design.

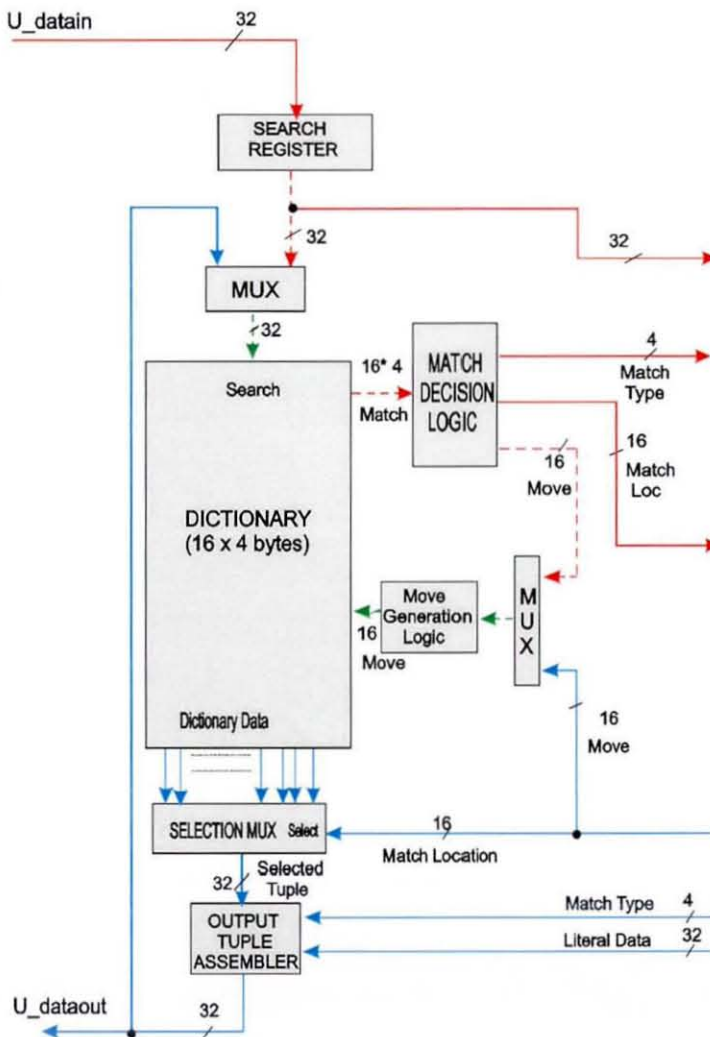


Figure 6.1. X-Match model architecture.

An initial effort to map X-Match to FPGA technology [Nunez99] reveals that the same critical path depicted in chapter 3 for the adaptation process holds in the FPGA implementation. The signals depicted in red colour correspond to compression related signals, those in blue colour relate to decompression related signals and those in green colour are shared by both channels. Dotted lines are used for critical paths. This colour scheme is maintained for the rest of the work.

The model comprises the following blocks:

- *Dictionary*: CAM-based dictionary with 16 tuples. The 16-tuple dictionary is formed by a total of $16 \times 32 = 512$ CAM cells. Figure 6.2 shows a section of the dictionary architecture with 4 tuples.

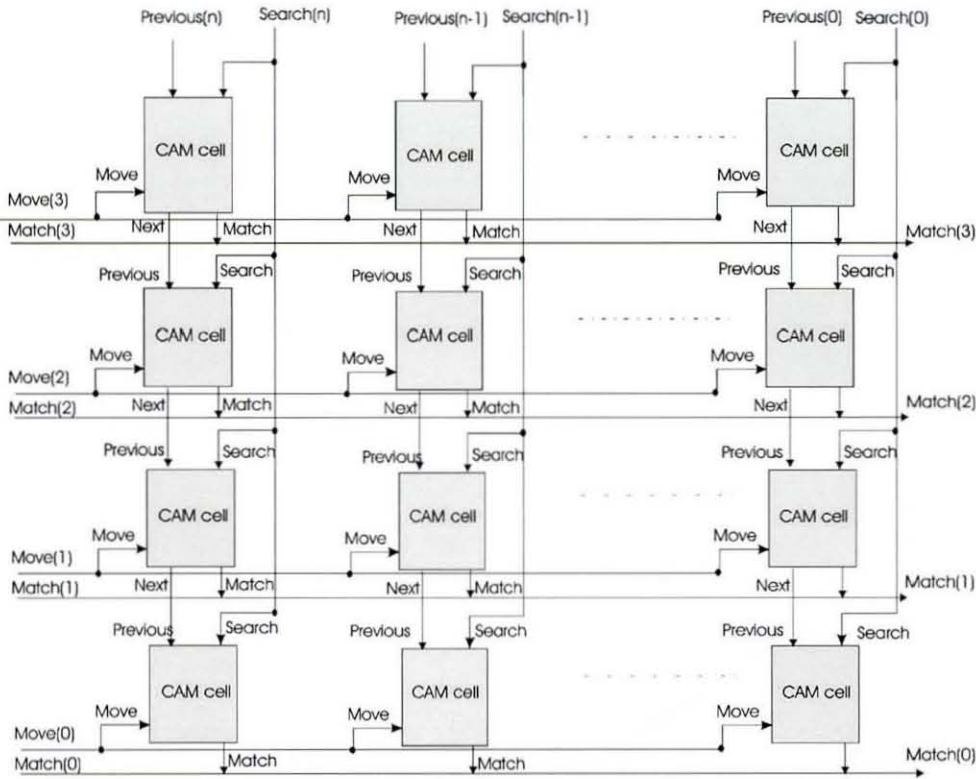


Figure 6.2. CAM-based dictionary architecture section.

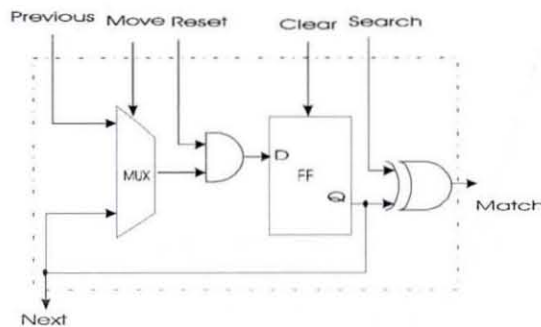


Figure 6.3. CAM-cell architecture.

Each tuple is formed by a row of 32 identical CAM cells. The CAM cell is illustrated in Figure 6.3. Each cell stores one bit of a data tuple. The cell can maintain its content or load the value store in the north neighbouring cell available in the *previous* input as indicated by the *move* input. The bit stored in the cell is compared with the bit present in the *search* input using a *xor* gate. This bit is also available to the south neighbouring cell in the *next* output.

- *Match decision logic*: Logic that assigns a different priority to each possible match type in the dictionary and selects one of the matches as the best for compression.
- *Move generation logic*: Generation of the adaptation vector depending on the match type (full match or partial match) and the match location.
- *Selection multiplexor*: Logic that selects one data tuple from the dictionary to be input in the output tuple assembler during decompression.
- *Output tuple assembler*: Module that assembles a decompressed tuple using dictionary information and any literal characters present in the code.

The critical path involves a feedback loop that extends from the search register, first multiplexor, CAM dictionary, best match decision logic, second multiplexor, movement generation logic and back to the CAM dictionary to provide the necessary information to reorder the dictionary. The feedback loop prevents us from inserting a simple pipeline register without affecting the algorithm functionality. This feedback loop is illustrated with a dotted line in Figure 6.1.

Careful study of this path reveals that the vector that defines how the dictionary adapts to the data can be generated much earlier at no extra cost in terms of area. The reason is that the shift down operation is only local to some dictionary positions when a full match occurs. Therefore it is not necessary to resolve the best match to know how to shift the dictionary. It is only necessary to know if a full match has happened and where to be able to generate the adaptation vector. If there is not full match the shift affects all the locations and if there is a full match this is known before accessing the best match location logic. This change together with moving the search multiplexor out of the critical path leaves the architecture as shown in Figure 6.4 where the match decision logic has been split into 2 components: the priority logic and the match decision logic.

- *The priority logic:* This logic assigns a different priority to each of the possible matches and it was originally embedded in the match decision logic.
- *The match decision logic:* It uses the priority information to select one best match and it moves out of the critical feedback loop.

After this modification the critical path is approximately 10% faster but it still remains the slowest part of the device. Although the search operation in the CAM dictionary and the priority assignation are parallel processes the generation of the adaptation vector by the move generation logic involves propagating the match location up so all the locations on top of the match location can move down. This propagation is critical and the number of levels of logic depends on dictionary size with the expression $O(\log_2(\text{dictionary size}))$. The timing of the search operation is also affected by the dictionary size because of the higher fanout associated to larger dictionaries.

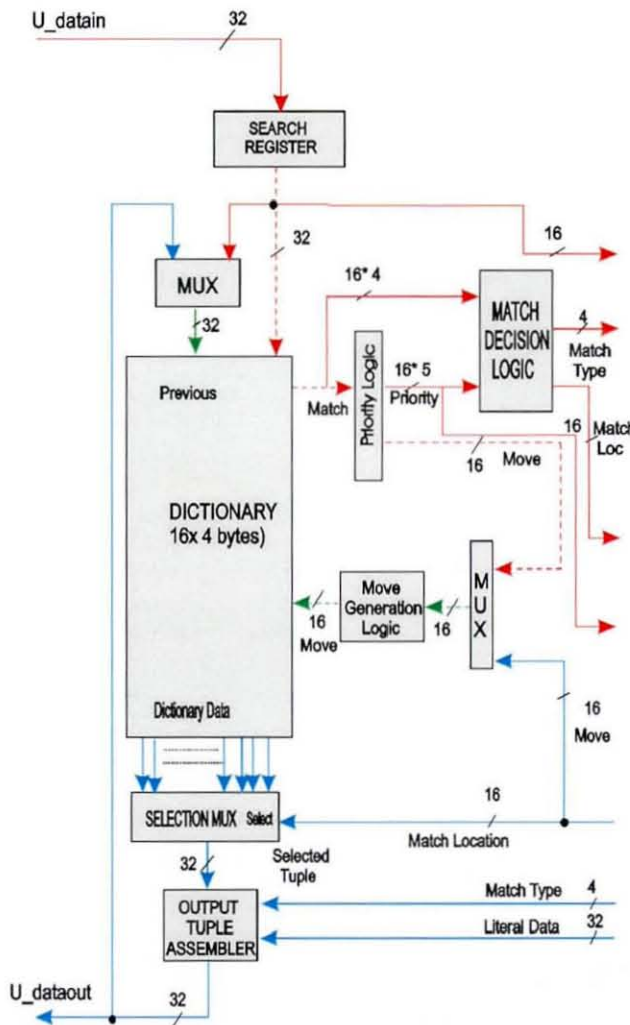


Figure 6.4. Modified X-Match model architecture.

6.3.1 Out of Date Adaptation (ODA) description

To improve further the critical path it is necessary to modify the algorithm functionality by introducing an Out of Date Adaptation (ODA) mechanism in the model. ODA implies that adaptation at time $t+2$ takes place with the information provided by the previously processed tuple at time t and not the one at time $t+1$. This technique breaks the fundamental feedback critical loop effectively in 2. The danger is that dictionary efficiency could be lost if the ODA technique duplicates the same tuple in different positions in the dictionary. In the architecture depicted in Figure 6.1 and 6.4 the adaptation vector at time t provides information to reorder the dictionary at time $t+1$ and makes sure that tuples are unique in the dictionary. In ODA the adaptation vector at time t is not effective until time $t+2$ so adaptation at time $t+1$ could insert a tuple in the dictionary that already exists in some other dictionary location degrading dictionary efficiency. Dictionary efficiency is quickly lost if the same data is duplicated in different positions of a small dictionary. The way to avoid this is by forcing the current adaptation vector to adapt not only to the CAM as before but also the next adaptation vector. Figure 6.5 illustrates this process. The only negative effect is then that the dictionary behaves like it has one entry less but data duplication is restricted to position 0.

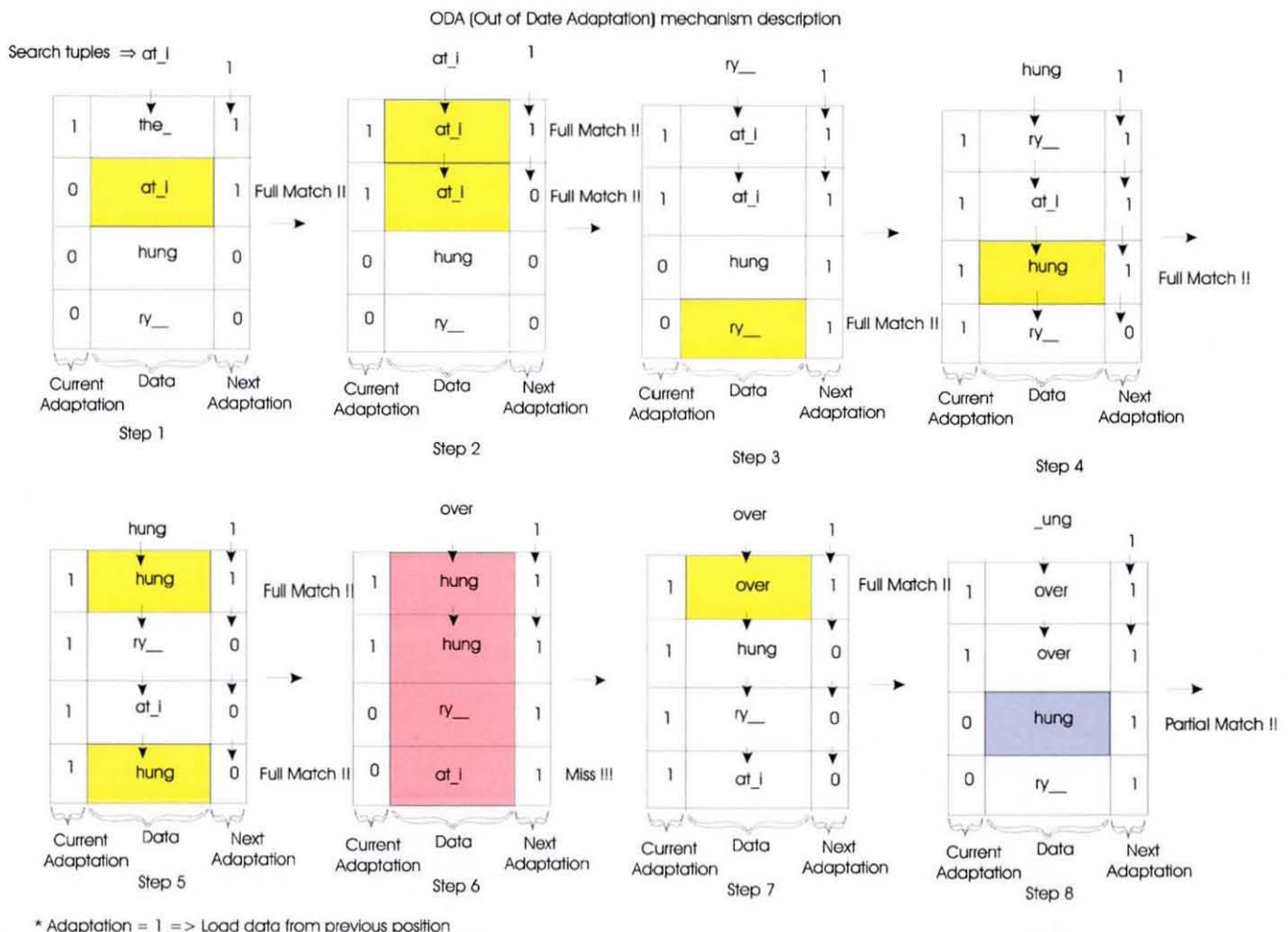


Figure 6.5. ODA mechanism.

Figure 6.5 shows how a simple dictionary of only 4 positions adapts to the incoming data source using the ODA technique. Every step corresponds to a different cycle. The yellow boxes show how possible full matches occurred simultaneously at position 0 and position bigger than zero but in this cases match at position zero is always selected as valid. The next adaptation vector depicted at the right of the dictionary depends exclusively on this match information. The figure shows how the ODA technique adapts the dictionary at time $t+2$ using a modified adaptation vector originally generated at time t and how data duplication is restricted to position 0. For example, the current adaptation vector depicted at the left of the dictionary for step 3 is generated shifting down the next adaptation vector of step 2 as indicated by the current adaptation vector of step 2. The current adaptation vector at step 3 will adapt the dictionary for step 4. By using this simple technique the effect of ODA on the compression ratio is negligible because in the worst case only one dictionary position contains repeated information and in the best case all dictionary positions contain different data.

Table 6.1 explains the steps of Figure 6.5.

Step number	Action
1	<ul style="list-style-type: none"> • Full match detected at position 1. • Next adaptation vector set to 1 at positions 0 and 1. • Current adaptation vector loads search tuple in position 0.
2	<ul style="list-style-type: none"> • Full match detected at position 0 and 1. • The algorithm selects the one closer to the top as valid (position 0). • Next adaptation vector is set to 1 at position 0. • Current adaptation vector loads position 0 in position 1 and search tuple in position 0. It also shifts next adaptation vector one position down.
3	<ul style="list-style-type: none"> • Full match detected at position 3. • Next adaptation vector is set to 1 all positions from 0 to 3. • Current adaptation vector loads position 0 in position 1 and search tuple in position 0.
4	<ul style="list-style-type: none"> • Full match detected at position 2. • Next adaptation vector is set to one in positions 0,1 and 2. • Current adaptation vector shifts down all the data one position, loads search tuple in position 0 and also shifts down the next adaptation vector.

Table 6.1. ODA description (Continued next page).

5	<ul style="list-style-type: none"> • Full match detected at positions 0 and 3. • Position 0 selected as valid. • Current adaptation vector shifts down all the data one position, loads search tuple and it also shifts the next adaptation vector.
6	<ul style="list-style-type: none"> • Miss detected. • A miss sets all the bits in the next adaptation vector to 1. • Adaptation is as previous cycles.
7	<ul style="list-style-type: none"> • Full match at position 0. • Adaptation is as previous cycles.
8	<ul style="list-style-type: none"> • Partial match at position 2. • Partial matches are dealt with as misses for adaptation purposes. • Adaptation is as previous cycles.

Table 6.1. ODA description (End).

Figure 6.6 shows the new architecture with one component added:

- *The ODA logic:* It uses a multiplexor and a register to store the next adaptation vector shifting it down one position as indicated by the current adaptation vector. The register breaks the feedback loop.

ODA proves a very effective technique to ensure that data duplication at position 0 is only effective for one cycle and this technique maintains the original dictionary efficiency. The logic cost of ODA is small because only a register and a multiplexor of length equal to dictionary length are required. The control bus in Figure 6.6 decides if the new adaptation vector is loaded directly or one position down. This operation is critical to guaranty that a data tuple duplicated at time t will be quickly deleted from the dictionary at time $t+1$. Figure 6.6 shows with a dotted line how the original critical path has been split into 2 non-critical paths that correspond to the search operation and the adaptation operation. These two paths have been balanced to have a similar delay. The ODA-based architecture is approximately 100% faster than the non-ODA of figure 6.4.

ODA could also be applied to improving the performance of on-chip cache memories that use a least-recently-used algorithm [Tanenbaum90] to know which cache line should be evicted once the cache becomes full. This problem arises in full-associative or multiple-way set-associative caches where different cache lines can be allocated to the same data item. There is a third type of cache organisation: the direct-mapped cache where the selection of the data

item for eviction is trivial because the new data item can only be stored in one particular cache line. Associative caches offer a higher hit ratio than direct-mapped caches [Drach95] but many cache designs use the second ones due to their fast access time and ease of implementation [McFarling91]. This means that a lot of research has been carried out in improving the performance of direct-mapped caches or caches with low-associativity (2-way to 4-way) aiming to maintain their simplicity whilst improving their hit ratios [Wilson97], [McFarling91], [Jou90], [Wolf91]. Associative caches are kept simple by using in many cases a random policy to select one cache line for eviction but it is generally accepted that a more sophisticated policies [Hallnor00] such as least-recently-used increase the hit rate at the expense of a higher access time. ODA can effectively decrease the amount of time needed to select one cache line for eviction enabling higher-levels of associativity. (> 8-way). This area remains for future research.

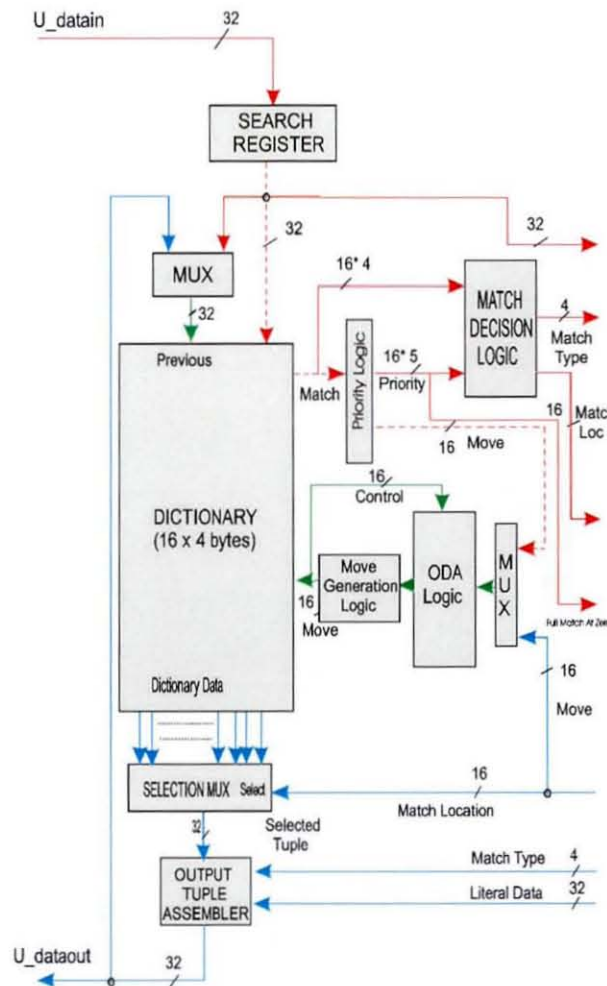


Figure 6.6. ODA-based X-Match model architecture.

6.4 Coder/Decoder architecture

Figure 6.7 shows the components of the code/decode logic, namely:

- *16-to-4 encoder, 4-to-16 decoder*: Logic that assigns a 4-bit binary code to the 16-bit match location vector or a 16-bit match location vector to a 4-bit binary code respectively.
- *Binary code generator*: Logic that generates a phased binary code (PBC) or a uniform binary code (UBC) depending on the implementation and concatenates it with a bit indicating a match. It also supplies the length of this match location code.

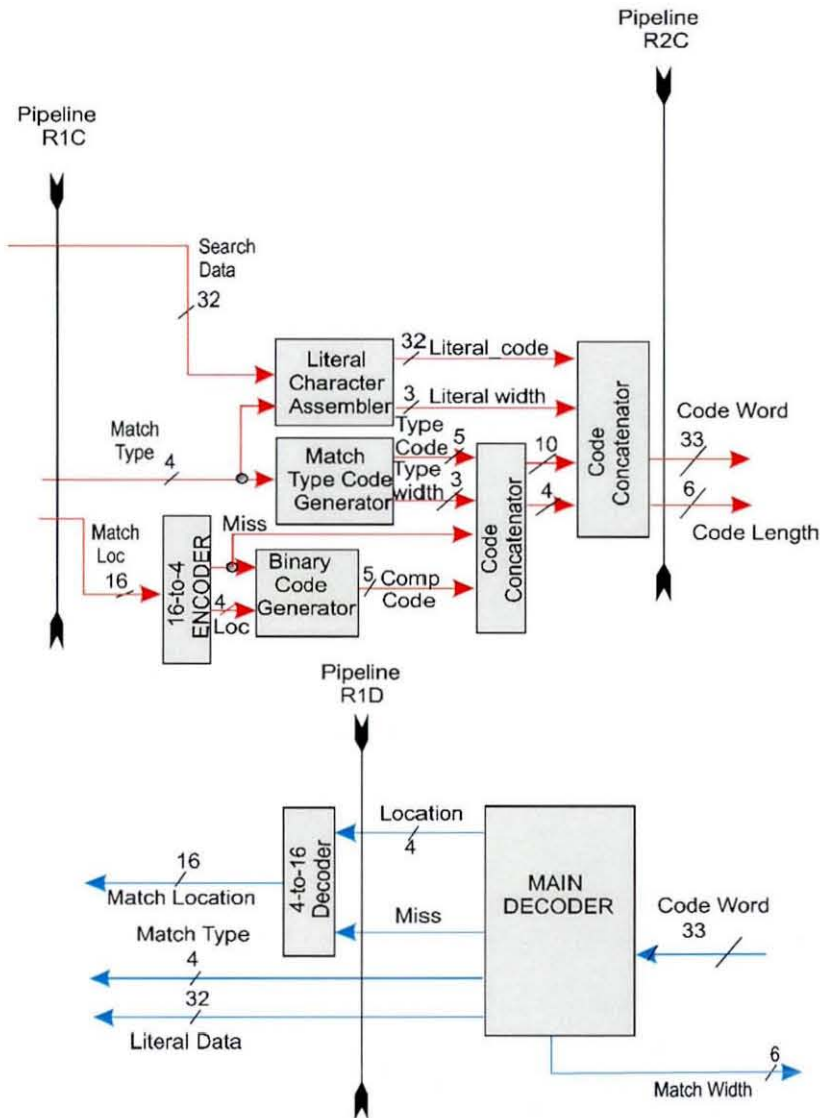


Figure 6.7. X-Match coder/decoder architecture.

- *Match type code generator*: Logic that assigns a static Huffman code to each possible match type. There are 11 possible different match type combinations of 2,3 or 4 bytes matching in the tuple. The Huffman tree obtained after extensive simulation has only 4 different code lengths of 2,3,4 and 5 bits. The full match is the most probable match type and its Huffman code is only 2-bit long. Matches of 3 non-consecutive bytes are the most improbable and they are assigned 5-bit long Huffman codes.
- *Literal character assembler*: Logic that uses the match type information to produce a code formed by the bytes that are not part of the match.
- *Code concatenators*: Logic that concatenates the codeword components into a single code to be supplied to the bit packer.
- *Main decoder*: The main decoder obtains a match type and a match location from the codeword supplied by the bit unpacker. The first bit defines if a miss or a match follows. If a match is detected the next following bits in the codeword define the match location and the number of them depends on how many entries are valid in the dictionary if using PBC. This number is fixed at $\log_2(\text{dictionary_size})$ in a UBC implementation. The match type code follows the match location code. If the match is partial the missing bytes follow the match type. If instead of a full or partial match a miss is detected the next 32 bits following the first bit correspond to the 4 missing bytes.

The coding operations in X-Match are not time critical because only 11 different static Huffman codes are used for the match types and the PBC codes are, in essence, UBC codes which lengths depend on how many entries are valid in the dictionary and on where the match is located. None of these techniques require complex or slow operations. There are also no feedback loops (as Figure 6.7 illustrates) so pipeline registers can be inserted if required. The position of the pipeline registers is also shown in Figure 6.7. The coding logic also assembles these codeword components into a single codeword before they are made available to the bit packer. Decoding is also simple but in this case a feedback loop exists between the decoder and the unpacker. The reason is that the unpacker needs to know the number of bits used by a codeword before it can shift out old bits and concatenate new bits to the uncompressed code. The number of bits used by a codeword is not known until the codeword has been decoded in

the decoder. The feedback signal *match width* in Figure 6.7 carries this information to the bit unpacker. Section 6.5 deals with this feedback loop.

6.4.1 Run Length Internal (RLI) description

The new coder/decoder adds extra functionality because the RLI technique that codes multiple full matches at location zero into a single run-length code is embedded in the architecture. Figure 6.8 shows the coder/decoder architecture with the RLI logic added.

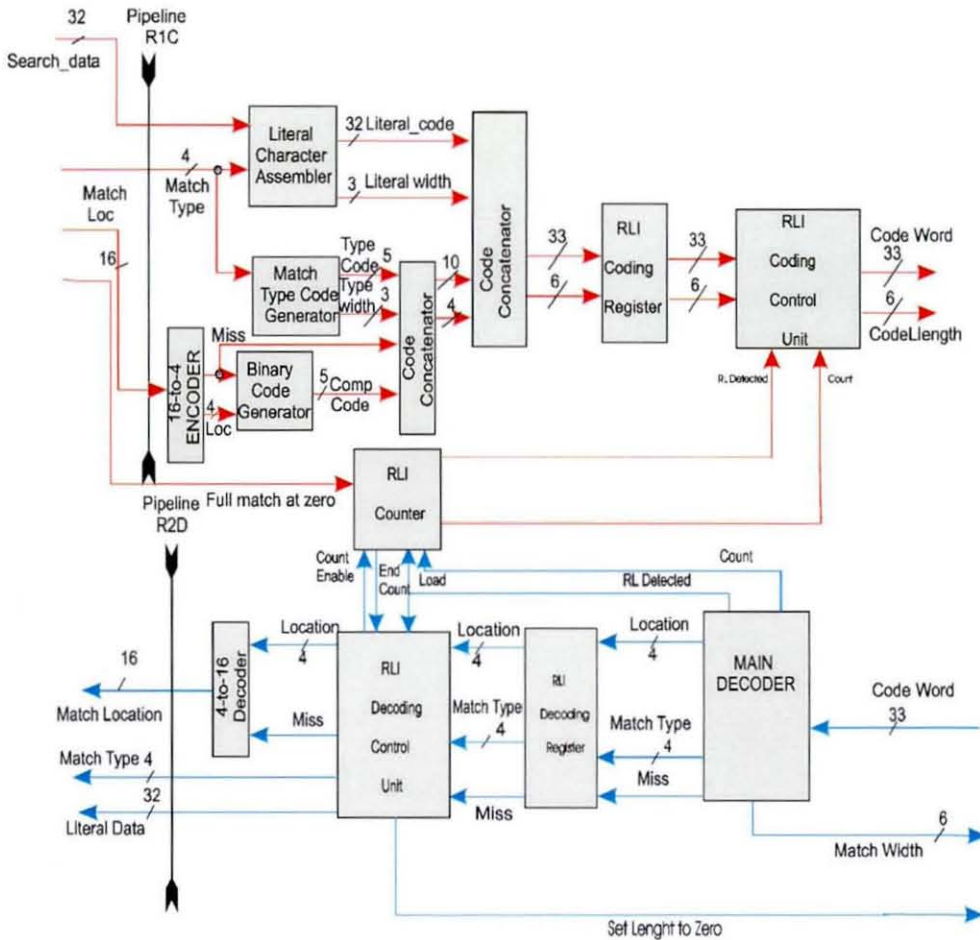


Figure 6.8. RLI-based X-Match coder/decoder architecture.

RLI adds the following components:

- *RLI coding register*: Buffers the codeword before it enters the bit packer logic. This buffering function is necessary to enable resetting the pipeline from a full match at position zero that will be coded as part of an RLI event. The pipeline will not be reset

- if the RLI counter does not exceed the count of 1. If the count remains at one, then, a single full match at location zero has been detected and a valid RLI event is not present.
- *RLI coding control unit*: The RLI coding control unit monitors the output of the RLI counter. If this value is equal or bigger than 2 then a RLI event is detected as valid.
- *RLI counter*: The RLI counter changes its operational mode if compressing or decompressing. In compression it counts consecutive full matches at location zero in the dictionary up to a maximum value of 255. In decompression it is loaded with a value that indicates the length of the run and then it counts up until this value is reached.
- *RLI decoding register*: Buffers the output of the main decoder before it enters the RLI decoding control unit. This buffering effect is needed to allow the timing of the signal *set length to zero* to be correct. *Set length to zero* signals that an RLI event is active and the bit unpacker must maintain its current state as many times as indicated by the length of the run.
- *RLI decoding control unit*: The RLI decoding control unit monitors the existence of the binary code corresponding to the last position in the dictionary. This code is reserved for RLI events. If this code is detected the run length value is loaded in the RLI counter and the RLI control logic outputs full match at position 0 until the run is exhausted.

An 8 bit counter is shared by the coding and decoding RLI logic. In compression mode this counter does not use any specific technique to detect an overflow condition if a pattern repeats more than 255 times. The count simply loops back to zero. This condition is detected by the RLI control logic as an end of run and a RLI code is output. The next code after a RLI code is always a normal code even if the pattern continues repeating. If this is the case the 8-bit counter exceeds the count of 1 again and the run length detection signal is reactivated. This simple mode of operation simplifies the RLI control logic. Figure 6.8 illustrates how the RLI logic is neatly integrated with the rest of the coder/decoder logic.

In compression mode the output of RIC is used to code the match location using a 4 bit ($2^4=16$) binary code and the match type using a static Huffman code. Any needed literal characters are added and the result accesses the RLI coding logic. If the following tuple $t+1$ to

access the match decision logic keeps the RLI counter enable an RLI event is detected as valid because at least 2 tuples have generated consecutive full matches at location zero. This means that the compressed code corresponding to tuple t is eliminated from the pipeline and replaced by an RLI code where tuples $t, t+1, \dots, t+RLI_length$ will be efficiently coded.

The RLI event remains active for as long as the full match at zero signal is set or for a maximum of 255 repetitions. Then, the RLI code is output always followed by the normal code of the tuple that terminated the run length. The result accesses the bit packing logic. In decompression mode the compressed data enters the main decoder to produce a match location and a match type and any possible RLI events are promptly detected. A RLI condition is signal to the RLI decoding control unit which changes its mode of operation. The output of the RLI decoding logic is pipelined in register R2D after decoding the match location in the 4-to-16 decoder.

Figure 6.9 shows an example of the RLI process.

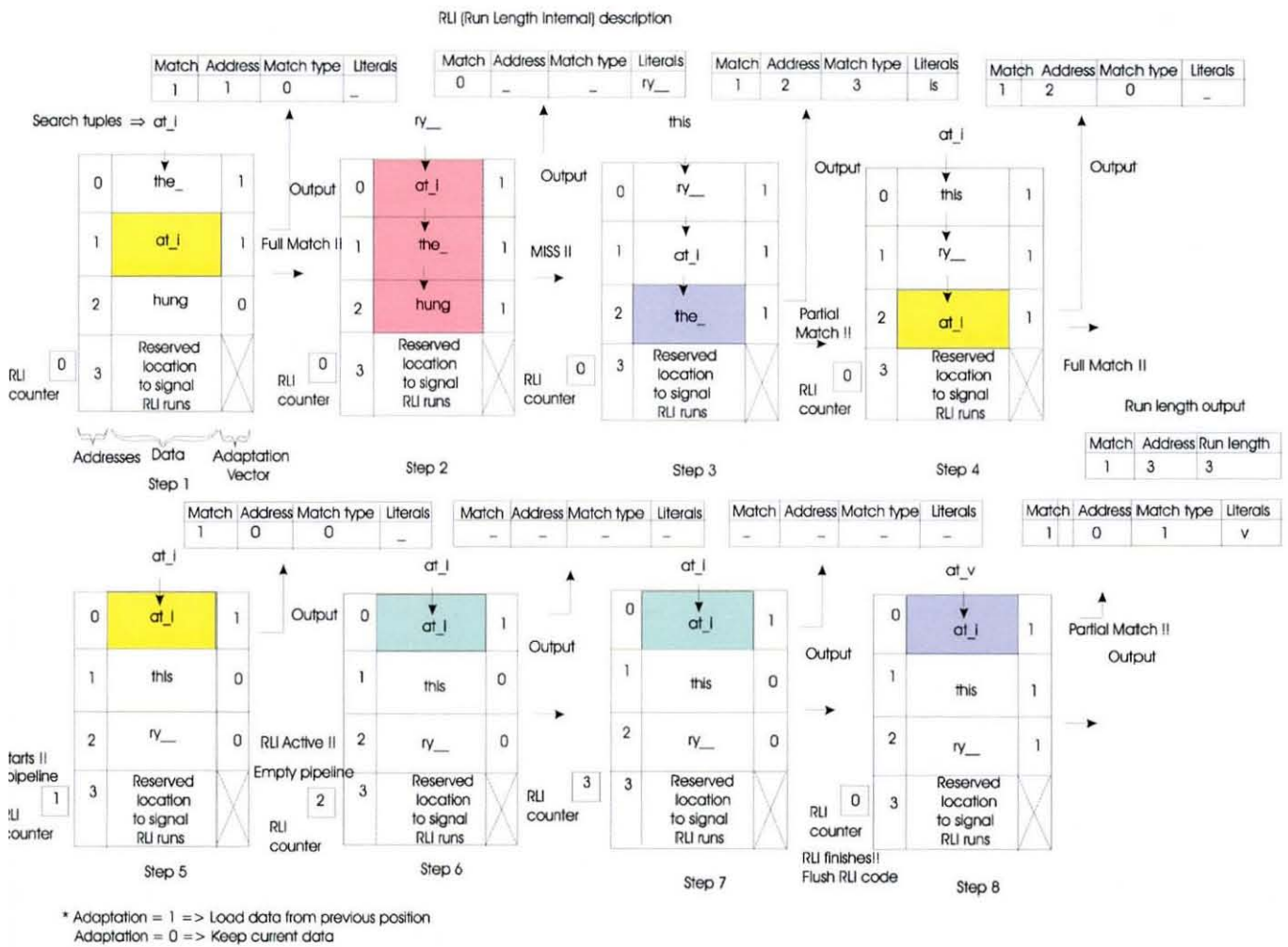


Figure 6.9. RLI mechanism.

An RLI coding event is active in steps 5,6 and 7. The RLI output is generated at step 8 when the run stops with a length of 3. Figure 6.9 shows that the counter only increments when the search data is present at position 0. The code generated at step 5 is deleted from the pipeline when the RLI count exceeds 1 because it will be coded as part of the run-length. Table 6.2 explains the RLI process.

Step number	Action
1	<ul style="list-style-type: none"> • Full match found at position 1. • Normal output. • RLI counter=0.
2	<ul style="list-style-type: none"> • Miss detected. • Normal output.
3	<ul style="list-style-type: none"> • Partial match detected at position 2. • Normal output.
4	<ul style="list-style-type: none"> • Full match detected at position 2. • Normal output.
5	<ul style="list-style-type: none"> • Full match detected at position 0. • Normal output but possible start of internal run length. • RLI Counter = 1
6	<ul style="list-style-type: none"> • Full match detected at position 0. • Valid run length detected. • Empty pipeline from the previous code. No output. • RLI Counter = 2.
7	<ul style="list-style-type: none"> • Full match detected at position 0. • Valid run length continue. No output. • RLI Counter = 3.
8	<ul style="list-style-type: none"> • Partial match detected at position 0. • Run length finishes. • Flush run length code. • RLI Counter = 0. • Normal output of data terminating the run length.

Table 6.2. RLI description.

6.5 Packer/Unpacker architecture

Figure 6.10 shows the packer architecture with the following components:

- *Code concatenator*: Logic that concatenates the current buffer codewords with a new codeword produce by the coder. The coder produces a new variable-length codeword each cycle. This logic assembles this variable-length codewords in 64-bit fixed-length codes than are then output to the compressed bus. The logic requires a 64-bit output bus because the maximum codeword length is 33-bit when a miss is detected and a 32-bit output bus could create a bottleneck.
- *Register*: Logic that buffers a maximum of 96 bits of code plus the number of valid bits in this code. A 96-bit register is necessary because in the worst case there could be 63 bits in the buffer waiting to be output and a 33 bits codeword could be generated ($63+33 = 96$). The active code length is stored in 7 extra flip-flops.

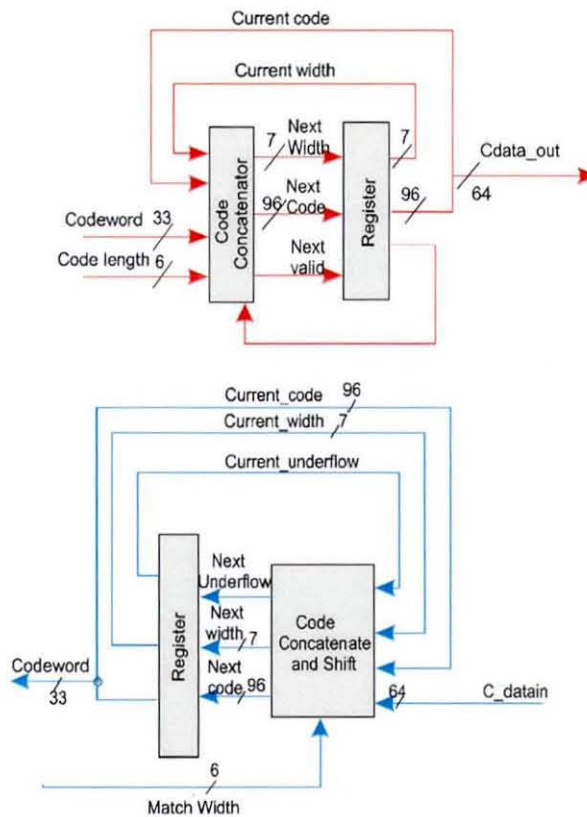


Figure 6.10. Packer/unpacker X-Match architecture

Figure 6.10 shows the unpacker architecture with the following components:

- *Code concatenate and shift*: This logic unpacks 64 bits of compressed data into variable-length codewords. To be able to shift out old data and concatenate new data the codeword length must be supplied by the decoder logic using the signal *match width*. This forms a critical feedback loop difficult to improve.
- *Register*: Register that buffers the current code before accessing the decoding logic. At least 33 bits of data must be valid in each cycle to prevent the decoder from failing. A register of 96 bits is needed because a new 64-bit compressed code must be added to the internal code when 32 or fewer bits are valid ($32+64 = 96$). The unpacker uses 7 extra flip-flops to store the active code length like the packer.

The architecture has been redesigned to reduce the logic present in the critical path and, hence, to improve its timing characteristics. Figure 6.11 shows a block diagram of the logic involved in the critical path and how the calculation of the match length must precede the concatenation of new data to the data not used in the previous uncompressing cycle. The critical path is depicted as dotted line.

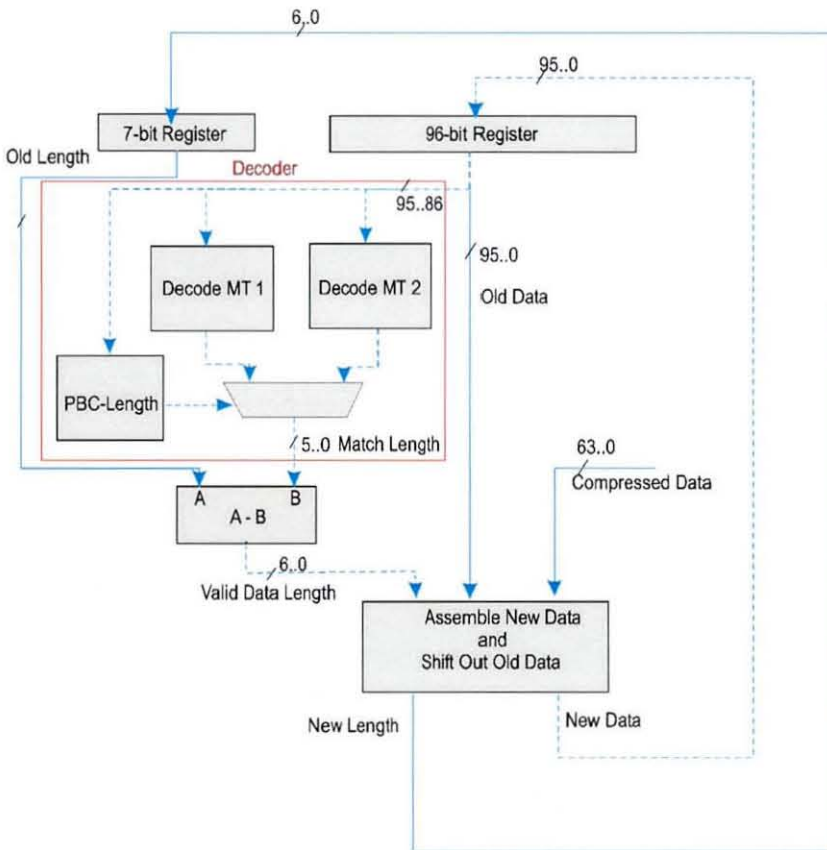


Figure 6.11. Decoder/Unpacker feedback loop.

The red box defines logic present in the decoder and illustrates how the feedback loop extends from the unpacker, to the decoder and then back to the unpacker to supply the *match length* signal. The *match length* is then subtracted from the *old length* to obtain the *valid data length* and this information is used to shift old data and to concatenate new data. This last step is very complex because it involves multiple multiplexing logic. If valid data length is bigger than 32 the input is shifted to eliminate data already used in the uncompressed cycle but new data is not added. If data valid length ranges from 0 to 32 new compressed data is concatenated to the right position of old data and a shifting operation takes place to eliminate data already used.

The redesigned architecture is based on concatenating new data in the *assemble new data* logic in parallel and independently to the process of calculating the decoded length in the decoder. Figure 6.12 illustrates the new architecture.

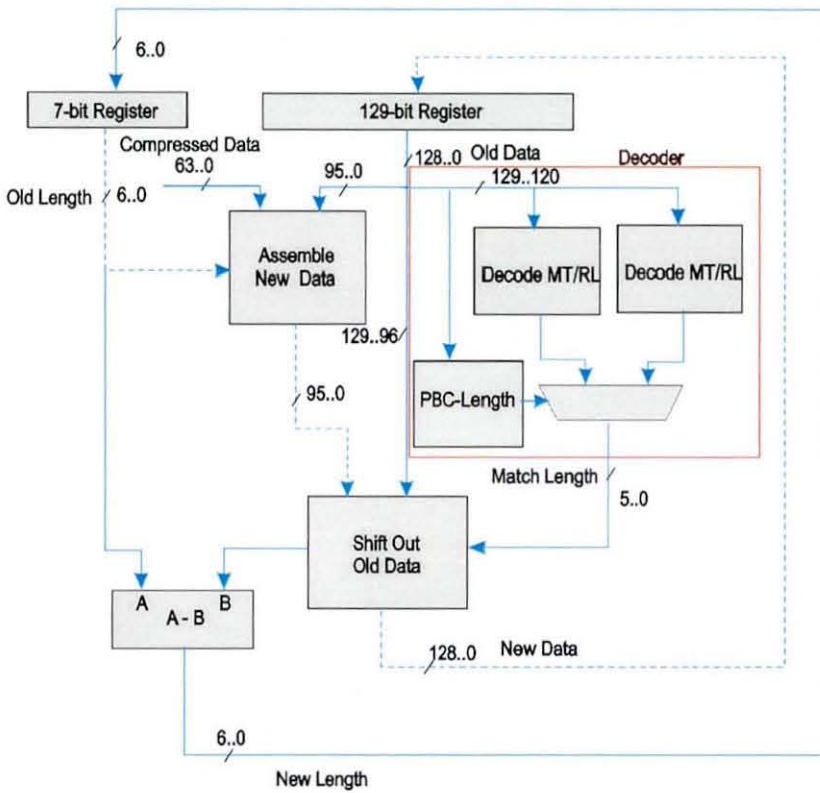


Figure 6.12. Decoder/unpacker modified feedback loop.

The new design uses *old length* to add new data when the number of valid bits is less than 66. This means that if there is at least 66 valid bits no concatenation of new data takes place for the next decoding cycle. The current decoding cycle can consume a maximum value of 33 bits so at least 33 bits (66-33) are left as valid in the register and the next decoding operation

can take place independently of how many bits are decoded in the current cycle. Since 64 bits are added to *old data* when the number of valid bits is less than 66 the decoding register is extended from 96 bits to 129 bits ($65+64 = 129$ bits). The complexity of the logic in the critical path is now simplified because the new *shift out old data* logic does not perform a concatenation operation which is parallel to the decoding process. The logic is simpler because it needs to perform only a shifting function, therefore, the resulting circuit is smaller and can run faster. The *match length signal* supplied by the decoder controls how the shifting is done.

This redesign speeds up the unpacking process with an estimated critical path 40% faster in Figure 6.12 compared with Figure 6.11. It remains, however, a critical component for performance since the feedback loop from the decoder is still present. Figure 6.12 shows with a dotted line the new critical path that does not include the subtracting operation or the decode logic. Figure 6.13 shows the new architecture of the unpacker. The same components are present but with different data widths. The inclusion of the RLI logic adds an extra control signal to the unpacker the *set length to zero* signal. This signal is active when an RLI event is active and indicates to the bit unpacker that it must copy the contents of its registers directly without shifting data until the RLI event finishes.

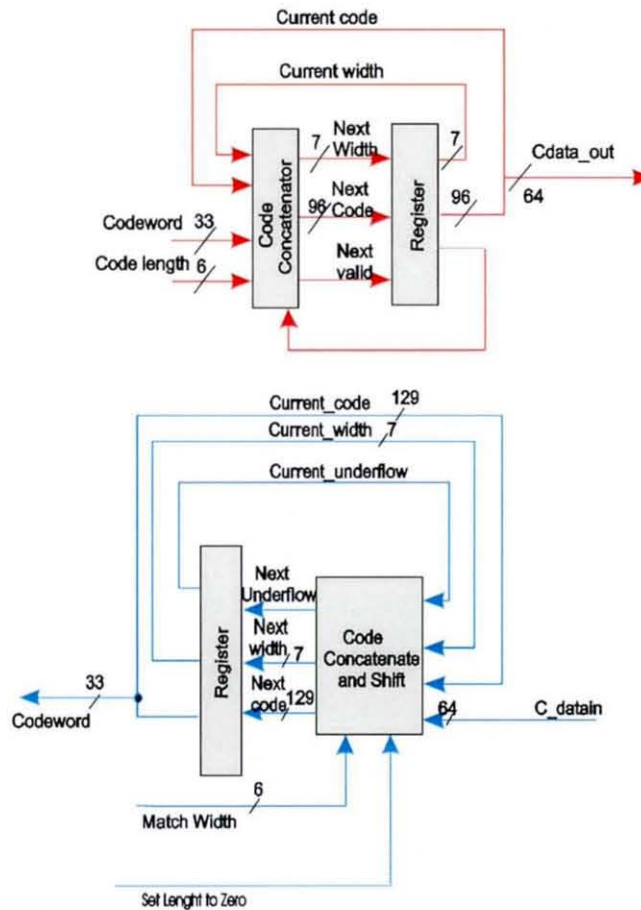


Figure 6.13. Packer/Unpacker modified X-Match architecture.

6.6 Compression/decompression core throughput evaluation

The 3 components described in the previous chapter formed the core or engine of the lossless data compressor. This new algorithm and architecture have been renamed X-MatchPRO as the next generation of the X-Match family of high speed lossless data compressors. To validate our architectural solutions in silicon we have synthesised and placed&routed the architecture into an A500K130 FPGA. The test of the FPGA is split into 2 different phases after post-layout back-annotated simulation is completed successfully. The first phase aims to verify that the functionality of the device is correct. The second one aims to verify that the timing characteristics reported after performing timing analysis in the placed&routed netlist are met in real operating conditions.

6.6.1 Serial test methodology

The functional test of the device uses a low cost PC-based test methodology and the JTAG port available in the FPGA. A text file is written automatically by a PERL script translating the original test vectors to the standard JAM [Altera98] programming and test language. JAM is a vendor-and-platform-independent interpreted language for programming and testing devices via the IEEE standard 1149.1 TAP controller, commonly known as JTAG. This file contains the test vectors and JAM instructions ready to be executed by the Gatefield ProASIC JAM player [Gatefield99] that controls the JTAG port shifting in the input test vectors clocking the device and shifting out the output test vectors. These vectors are compared with the expected output and fail or pass is reported. The same test vectors used during the simulation phased are now used in this verification phase to maintain consistency during the whole testing process. Figure 6.14 shows how the JAM player applies the test vectors to the JTAG port and reads back the clocked results.

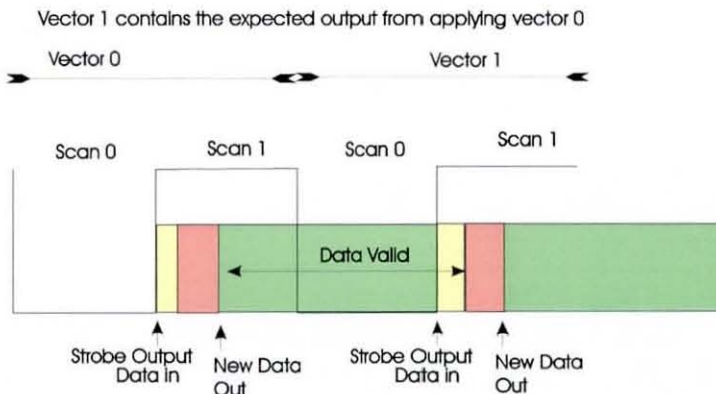


Figure 6.14. Serial test methodology

Each original test vector is decomposed into 2 vectors one corresponds to clock cycle low and the other to clock cycle high. After some propagation time showed as a shaded area in yellow and red the output of the circuit is ready to be strobed and scan out.

Figure 6.15.a shows the setup of main components present in the serial test hardware. The following components are present:

- A relay board used to switch voltages depending if the system is being used for programming or testing.
- A Hewlett/Packer E3611A DC power supply which is used to power the relay board at 8 volts.
- A Hewlett/Packer programmable E3631A DC power supply that provides the right voltages to the device.
- A National Instruments IEEE-488 GPIB controller board that connects the programmable power supply to the PC.
- A Corelis Corporation JTAG controller board that it is used to control the JTAG port of the FPGA.
- A PC where the JAM player and rest of the software are executed.

Figure 6.15.b shows a close-up on the A500K130 ProASIC FPGA and the ISP (In System Programming) module that holds the device during testing and programming cycles. This test allows us to verify the correct functionality of the device but since it is done serially at low speed it does not provide any information on the timing characteristics of the implementation.



Figure 6.15.a. Serial test hardware

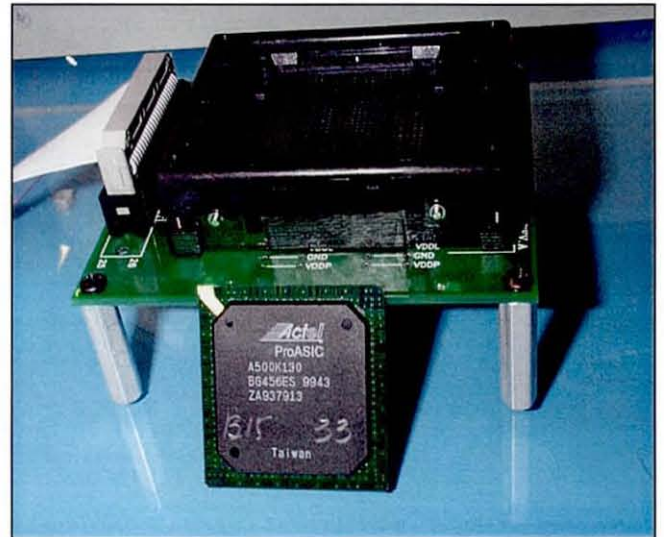


Figure 6.15.b. Close-up on programming/test module and A500K130 device.

6.6.2 Parallel test methodology

To evaluate the performance of the FPGA we make use of the Credence VistaLOGIC LT1101 parallel tester shown in Figure 6.16.

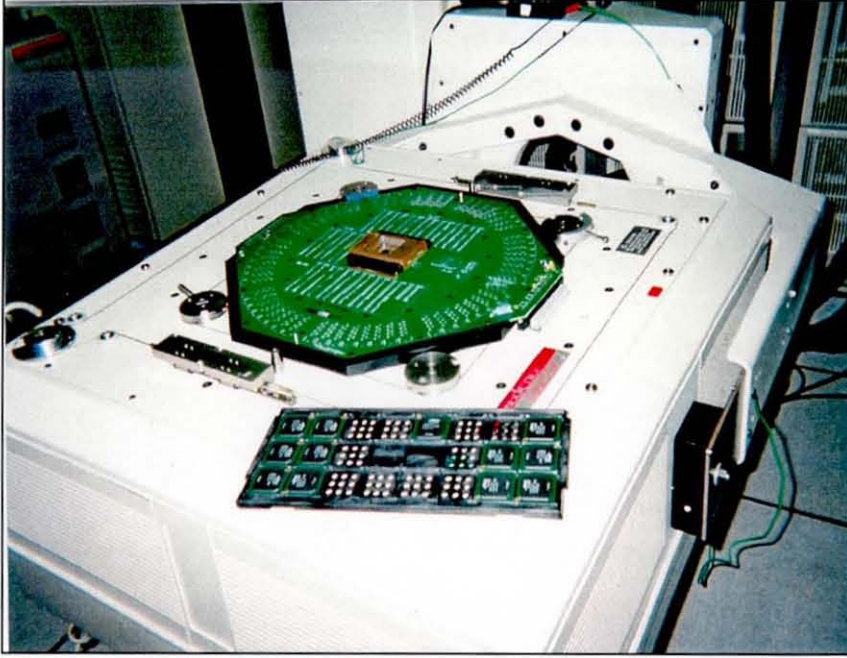


Figure 6.16. View of the VistaLogic LT1101 parallel test system.

This test system enables the identification of the maximum operating frequency changing variables such as strobe time, cycle time and operating conditions (supply voltage and room temperature).

Figure 6.17 corresponds to the SHMOO plots obtained in the tester with typical operating conditions of room temperature of 25 °C and supply voltage 3.3 volts. The X axis is the clock rising time (CRT) (The device is triggered with the positive edge of the clock) and the Y axis is the strobe time (when we read the output). All time figures are measured from the negative edge of the clock as illustrated in Figure 6.18.

The cycle time (CT) in figure 6.17.a is fixed at 27 ns but the duty cycle varies with the clock rising time. The area in green color corresponds to the valid working area. The ‘star’ zone situated on top of the green area corresponds to a strobe time higher than clock rising time and it is indicated in yellow in Figure 6.18. The output of the device is still being compared

correctly with the expected output because none of the output buffers have started changing its value but a new cycle has already started so it can not be considered a valid working area.

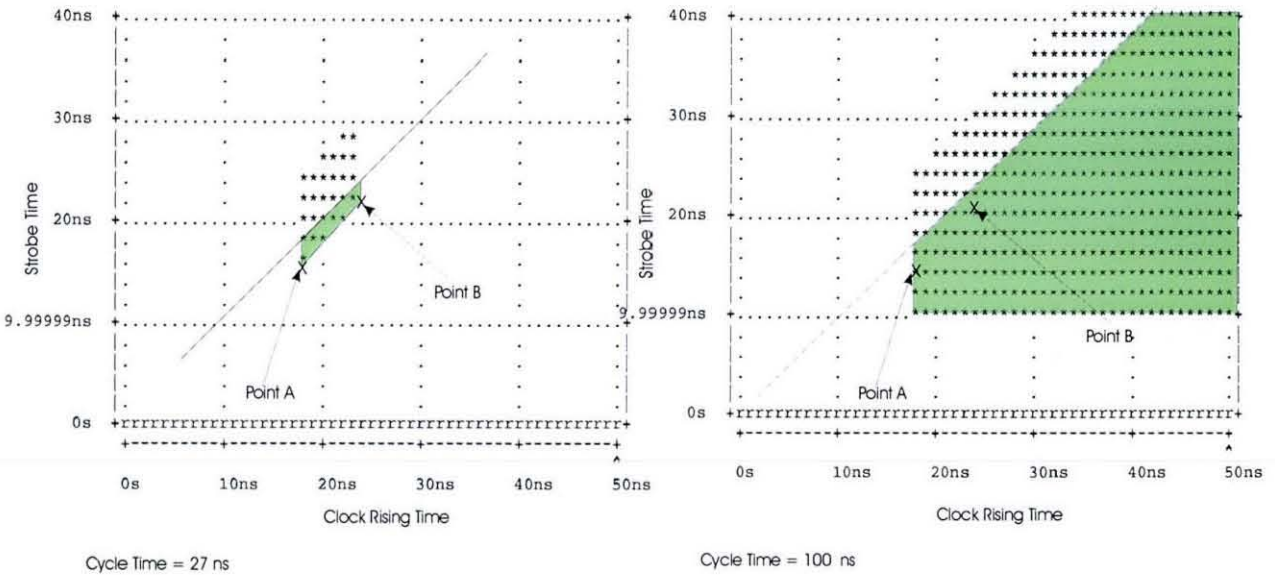


Figure 6.17a. SHMOO with 27 ns fixed cycle time. Figure 6.17.b. SHMOO with 100 ns fixed cycle time

Figure 6.18 illustrates operational point A in figure 6.17.a. Point A defines the minimum clock rising time and the minimum strobe time needed for the circuit to work. The circuit stops working in points located left of point A because the setup time available for the input vector to access the input buffers and reach the internal flip-flops before the circuit is clocked is not enough. The circuit stops working in points located south of point A because the strobe time that defines when the output is read is not enough for the output buffers to change and get stable. Point A is indicated in figure 6.18 in the transition from the red area to the green area and defines the minimum strobe time (mst). The area in green and yellow corresponds to moving north from point A in figure 6.17.a. The maximum strobe time (MST) maintaining constant the clock rising time at 17 ns is 25 ns. This point is not a valid working point because it corresponds to a new cycle. Figure 6.18 shows this point as a transition from yellow to red areas. The red area in Figure 6.18 corresponds to any other point in Figure 6.17 outside the 'star' area.

Point B in Figure 6.17.a corresponds to the maximum clock rising time. The device stops working in points located right of point B because the hold time available from the rising edge of the clock to the time a new vector is set in the input buffers is not enough. Vectors are always set with the falling edge of the clock as illustrated in Figure 6.18. The vector must be

stable in the input buffers for some time after the rising edge of the clock before it can be replaced by a new vector.

The strobe time increases when moving from point A to point B because of the increase in the clock rising time. The cycle, that extends from one positive edge to the next, starts later and therefore the output has to be strobe later as well for the circuit to operate.

Figure 6.17.b relaxes the clock cycle from 27 ns to 100 ns and as expected increases the valid working area shown in green. The minimum strobe time remains constant at 10 ns because of the time required to reset the circuit during the first cycle before the output is stable at 0 and it can be compared correctly with the expected output. Otherwise the strobe time should be 0 ns because the time elapsed from the positive edge of the clock to the negative edge of the clock (between 100 ns to 50 ns in Figure 6.17.b) is more than enough to account for the propagate time of the output buffers.

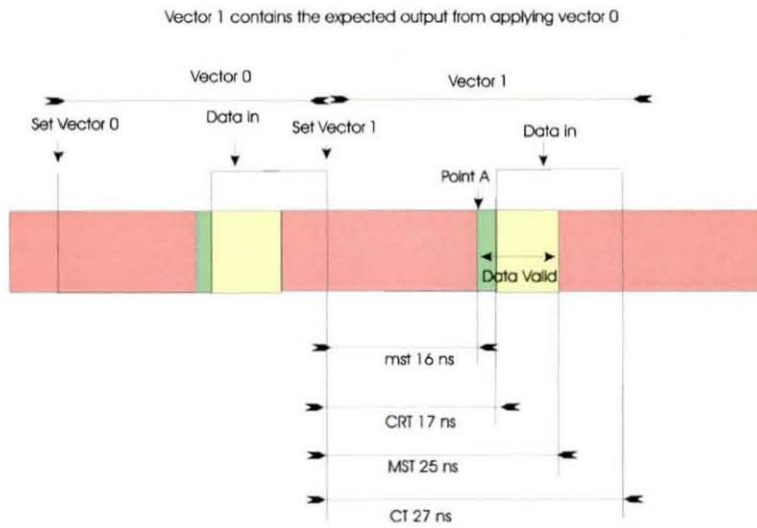


Figure 6.18. Timing relations at working point A.

Figure 6.19 corresponds to the SHMOO plots obtained in the tester with typical and worst operating conditions. The typical operating conditions correspond as in figure 6.17 to room temperature of 25 °C and supply voltage 3.3 volts. The worst operating conditions correspond to a room temperature of 70 °C and supply voltage 2.5 volts. The X axis is the strobe time (when we read the output) and the Y axis is the cycle time (clock period). The low time of clock is fixed at 20 ns while the cycle time varies from 20 ns to 50 ns.

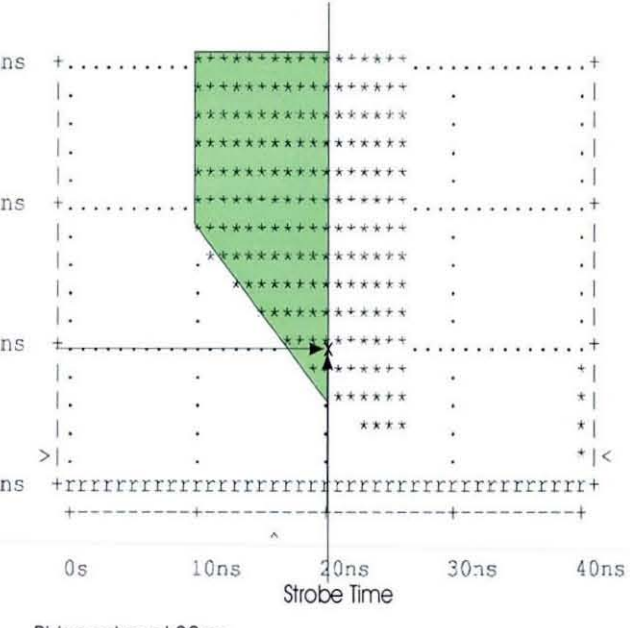
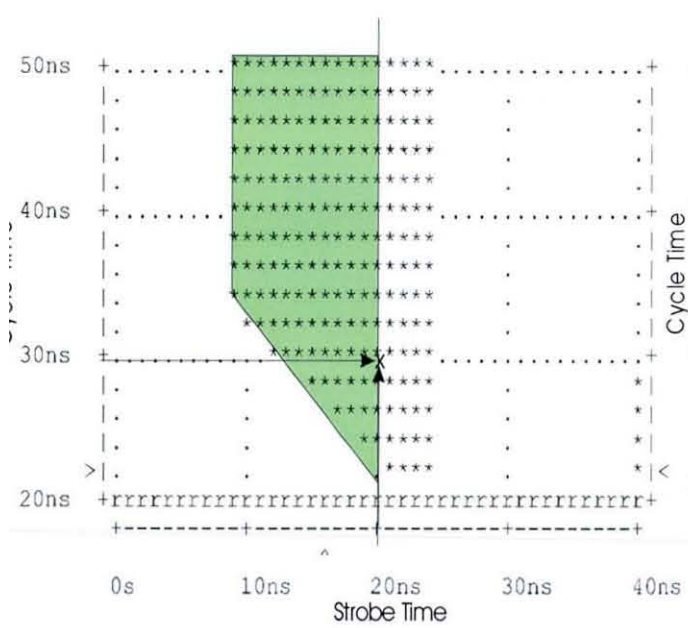


Figure 6.19.a. Typical conditions SHMOO

Figure 6.19.b. Worst conditions SHMOO.

The 'star' area in Figure 6.19 corresponds to the zone where the behavior of the device is as expected. The chosen operating point is marked with an 'X' in figure 6.19.a. Figure 6.19.b shows that under worst conditions our operating point gets closer to the non-functionality area but it is still within a safe margin. This operating point corresponds to the transition between green and yellow areas in Figure 6.20.

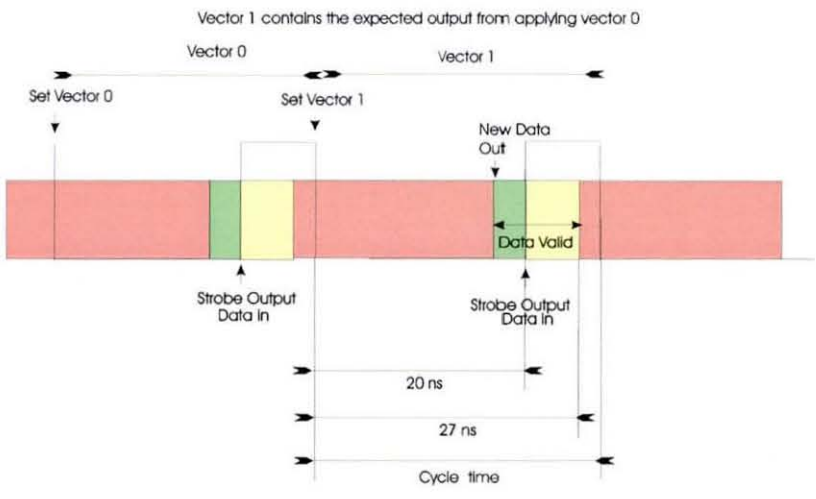


Figure 6.20. Parallel test methodology

The figure also shows that whilst the clock cycle could be reduce down to 25 ns and the operating point will still be inside the 'star' area the strobe time does not allow us to do that. The following relation must hold $strobe\ time \leq 20\ ns$ otherwise data is read in the yellow area after the clock has gone high and the current cycle has completed. Although figure 6.19.b shows a 'star' area extending from 20 ns to 27 ns strobe time this zone corresponds already to a new cycle and it should not be used. This area is shown in yellow in Figure 6.20. The area shown in red in Figure 6.20 corresponds to any other point outside the 'star' area of Figure 6.19.

6.7 Conclusions

This chapter has focused on analysing the throughput performance of the 3 main components of the architecture: Model, Coder/Decoder, Packer/Unpacker decomposing them into their simple components. A performance bottleneck has been identified in the model due to the existence of a feedback loop in the search and adaptation process. A novel solution based on adapting the dictionary using out of date information without losing dictionary efficiency has been shown to effectively remove the feedback loop. Another critical feedback loop has been identified between the decoder and the unpacker because the last one needs to know from the first one how many bits form the variable length codeword before old data can be shift out and new data added. The architecture of the unpacker has been redesigned to increase its level of parallelism and speed up the circuitry. The inclusion of the RLI functionality depicted in chapter 5 in the coder/decoder has been carefully executed to obtain the required behaviour avoiding generating throughput bottlenecks in this circuitry.

Finally, we have proven the correct functionality and good timing characteristics of the design using a A500K130 FPGA as our silicon testbench. A conservative operating point under worst-case operation conditions with a cycle time of 30 ns enables a 33 MHz clock cycle producing a data independent throughput of 1 Gbit/s (132 Mbytes/s). These tests validate the compression/decompression core design as meeting the requirements of section 3.1.

Chapter 7

X-MatchPRO

lossless data compression technology

7.1 Objectives of Chapter

Chapter 6 described in detail the architecture and performance of the new X-MatchPRO compression/decompression core and targeted an A500K130 ProASIC FPGA for its implementation. This was particularly useful to validate the correct functionality and benefits of the design. This chapter investigates the extension of the half-duplex to a full-duplex architecture minimising the impact on complexity. It also aims to expand the engine to a coprocessor-style architecture by adding a suitable system interface. Finally, it introduces the other 2 FPGA technologies of chapter 5 (Xilinx Virtex and Altera Apex) and validates X-MatchPRO as a high-performance portable design.

7.2 Full-duplex processing

Full-duplex processing is a valuable extension to the X-MatchPRO architecture to enable handling of both a compression and a decompression data streams simultaneously. In principle full-duplex functionality can be readily achieved by duplicating the dictionary so 2 independent dictionaries are used by compression and decompression. The decompression dictionary does not need to be a CAM because no parallel searching is needed to read the dictionary location pointed to by the match location component of the codeword. The decompression dictionary needs, however, to be able to shift the data to model the move-to-front (MTF) replacement policy used in the CAM. This feature prevents a straightforward

RAM-based implementation of the decompression dictionary. A shift-enable decompression dictionary based on flip-flops needs a total of *dictionary length* x 32 storage elements and it almost doubles the device complexity. The challenge is then to realise a shift-enable dictionary based on RAM. Embedded RAM is plentiful and ready to use in modern FPGA's such as the A500K, Apex or Virtex families so its usage does not have a direct impact on complexity. The design uses a pointer array logic to model the move-to-front replacement strategy used by the compression CAM shifting addresses to the dictionary data instead of the dictionary data itself. The width of the pointer word (4 bits in a 16-word dictionary, 6 bits in a 64-word dictionary) is a fraction of the width of the data word (32 bits) so the savings in logic are significant.

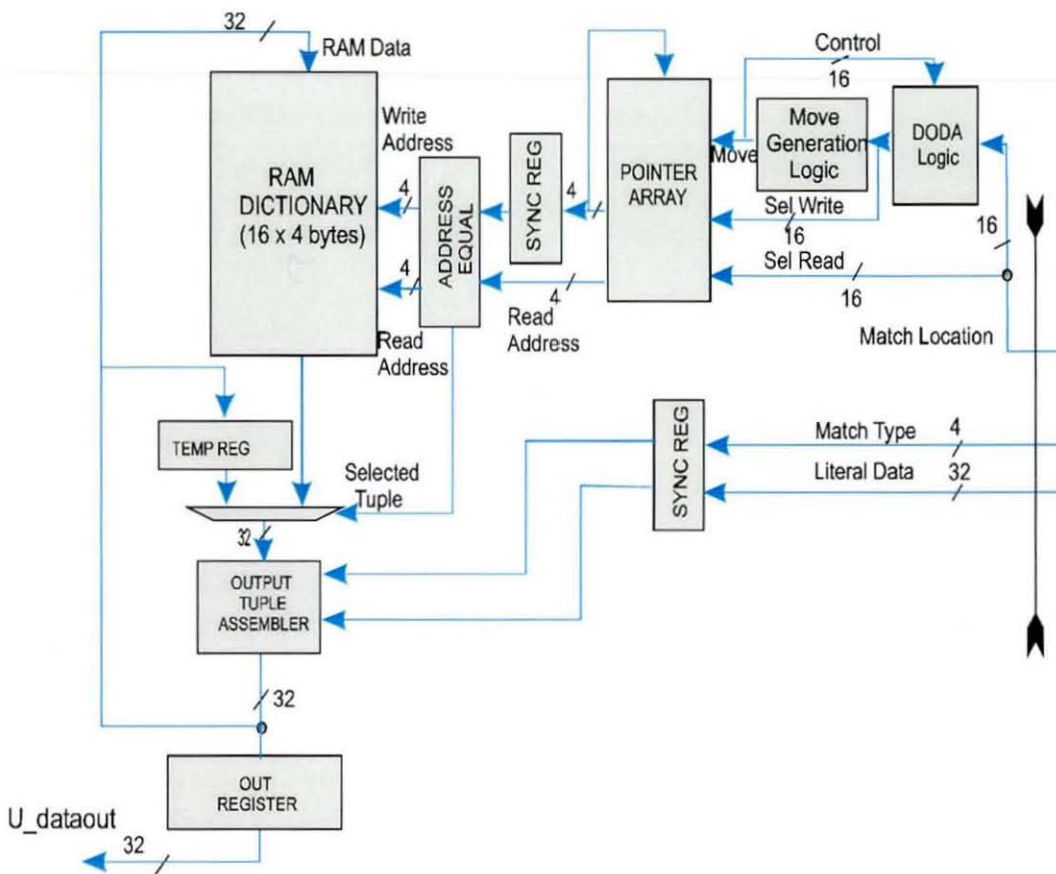


Figure 7.1. RAM-based decompression model.

Figure 7.1 shows a diagram of the RAM-based decompression model that comprises the following components:

Ram dictionary: Fully synchronous RAM-based dictionary that stores the history data during a decompression operation. The contents of the RAM dictionary during decompression must be same as the contents of the CAM dictionary during compression in each cycle. Adaptation must take place in exactly the same way to enable correct decompression of the compressed

block. The initialisation of the compression CAM sets all words to 0. This means that a possible input word formed by 0's will generate multiple full matches in different locations. The algorithm simply selects the full match closer to the top. This operational mode initialises the dictionary to a state where all the words with location address bigger than 0 are declared invalid without the need for extra logic. The reason is that location x can never generate a match until the data contents of location $x-1$ are different from 0 because locations closer to the top have higher priority generating matches. The MTF adaptation mechanism shifts down the dictionary when full matches are not detected and, therefore, ensures that the last word from this initial state to be deleted from the dictionary is always the word located at location 0 at time 0. This operational mode in compression enables the decompression RAM dictionary to have only location 0 loaded with value 0 during the initialisation phase because references to a RAM location y higher than 0 are not possible before the contents of the previous locations $y-1$, $y-2$, ..., 0 are updated. This technique avoids having a long overhead equal to dictionary_size cycles to initialise each position in the RAM to a predefined value before each decompression operation.

Pointer array: The pointer array logic performs an indirection function over the read and write addresses that access the RAM dictionary. It models the MTF maintenance policy of the CAM dictionary moving pointers instead of data. The pointer array enables mapping the CAM dictionary to RAM for decompression. Since the pointer array is much smaller than the CAM dictionary the savings in complexity allow having the full-duplex architecture in a single device. Each position in the pointer array is reset in a single cycle to a value the same as its physical location in the array before each decompression operation.

Sync reg: The *sync* registers form part of a pipeline level partially embedded in the RAM dictionary. From Figure 7.1 the read address does not have a *sync* register. The *sync* register corresponding to the read address has been embedded in the RAM to obtain fully synchronous RAM operation in the read and write ports. The algorithm maps naturally to a RAM read in asynchronous mode and written in synchronous mode because this is the mode the CAM is read and written in compression. This asynchronous mode of operation, although possible, results in a less portable and less robust design. A fully synchronous design can target different FPGA and ASIC technologies with a higher degree of confidence.

Address equal: This logic monitors the read and write addresses. If both addresses are the same the algorithm needs to read the data that is going to be written in that common address. This data is not present in the memory yet but it is present in the RAM *data in* bus. The RAM *data in* bus is written in the memory normally but it is also latched temporarily in the *temp*

register. The multiplexor associated to the *address equal* logic selects the input coming from the *temp* register instead of the input coming from the memory when the same address is being read and written. The *address equal* logic also modifies the read address to make it different from the write address and avoid corrupting the RAM contents.

Move generation: This logic generates the move vector depending on the match type and match location. The move vector adapts the CAM dictionary in compression and the pointer array in decompression.

DODA (Decompression Out of Date Adaptation) logic: This component forces the dictionary to adapt with previous match information and breaks the compression critical path improving speed. The ODA logic in decompression is used to replicate the adaptation process in the compression dictionary. They have exactly the same functionality although its usage to improve the timing characteristics of the design is restricted to the compression channel.

Temp reg: This register is used to hold a copy of the last data tuple written in the synchronous memory.

Output tuple assembler: Module that assembles a decompressed tuple using dictionary information and any literal characters present in the code.

Out register: Register that outputs the uncompressed data to the system.

Figure 7.2 shows how the indirection function works on the RAM dictionary and how the data contents of the decompression RAM are the same as the data contents of the compression CAM of Figure 6.5 in each cycle. In Figure 7.2 the yellow areas relate to read operations in the RAM dictionary. Blue areas relate to write operations in the RAM dictionary.

The presence of current and next adaptation vectors is due to the ODA policy described in section 6.3.1. It is possible to verify that decompression is taken place correctly because the output uncompressed data is the same as the input compressed data of Figure 6.5. The only exceptions are steps 6 and 8 that required extra data not present in the dictionary that must be obtained from the codeword literals. These 2 steps correspond to a miss and a partial match event respectively.

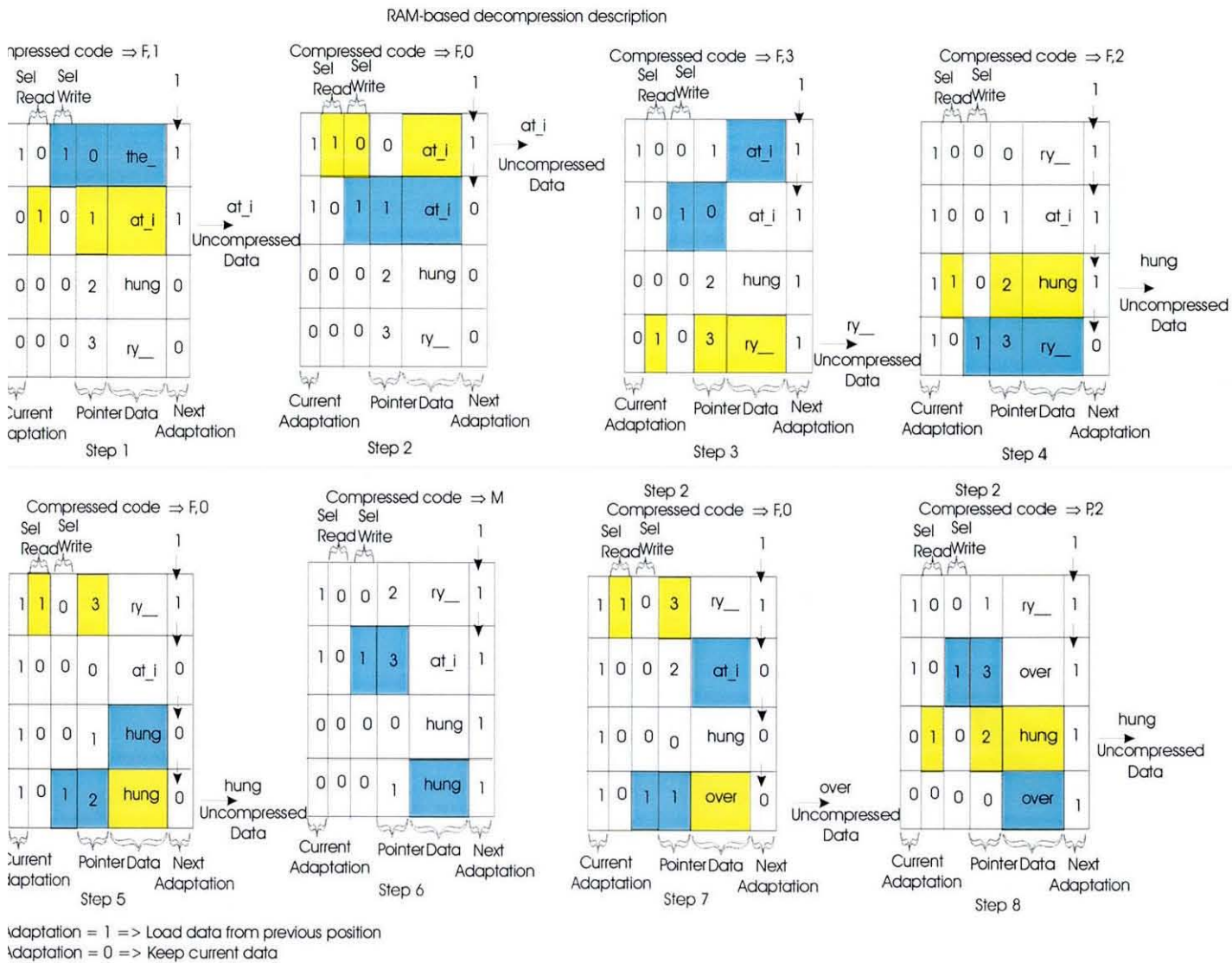


Figure 7.2. RAM-based decompression model mechanism.

Step number	Action
1	<ul style="list-style-type: none"> Compressed code indicates full match at location 1. The next adaptation vector is generated as defined by location 1. The pointer array contains address 1 at location 1 for reading. The uncompressed code 'at_i' is read from the memory. The current adaptation vector points at location 0 in the pointer array. The pointer array contains address 0 at location 0 for writing. 'at_i' is written at RAM position 0. The current adaptation vector shifts the pointer array and the next adaptation vector.

Table 7.1.a. RAM-based decompression description (Continued next page)

Step number	Action
2	<ul style="list-style-type: none"> • Compressed code indicates full match at location 0. • The next adaptation vector is generated as defined by location 0. • The pointer array contain address 0 at location 0 for reading. The uncompressed code 'at_I' is read from the memory. • The current adaptation vector points at location 1 in the pointer array. The pointer array contains address 1 at location 1 for writing. 'at_i' is written at RAM position 1. • The current adaptation vector shifts the pointer array and the next adaptation vector.
3	<ul style="list-style-type: none"> • Compressed code indicates full match at location 3. • The next adaptation vector is generated as defined by location 3. • The pointer array contains address 3 at location 3 for reading. The uncompressed code 'ry__' is read from the memory. • The current adaptation vector points at location 1 in the pointer array. The pointer array contains address 0 at location 1 for writing. 'ry__' is written at RAM position 0. • The current adaptation vector shifts the pointer array and the next adaptation vector.
4	<ul style="list-style-type: none"> • Compressed code indicates full match at location 2. • The next adaptation vector is generated as defined by location 2. • The pointer array contains address 2 at location 2 for reading. The uncompressed code 'hung' is read from the memory. • The current adaptation vector points at location 3 in the pointer array. The pointer array contains address 3 at location 3 for writing. 'hung' is written at RAM position 3. • The current adaptation vector shifts the pointer array and the next adaptation vector.
5	<ul style="list-style-type: none"> • Compressed code indicates full match at location 0. • The next adaptation vector is generated is defined by location 0. • The pointer array contains address 3 as location 0 for reading. The uncompressed code 'hung' is read from the memory. • The current adaptation vector points at location 3 in the pointer array. The pointer array contains address 2 at location 3 for writing. 'hung' is written at RAM position 2. • The current adaptation vector shifts the pointer array and the next adaptation vector.

Table 7.1.b. RAM-based decompression description (Continued next page)

Step number	Action
6	<ul style="list-style-type: none"> • Compressed code indicates miss. • A miss sets to 1 all bits in the next adaptation vector. • No reading • The current adaptation vector points at location 1 in the pointer array. The pointer array contains address 3 at location 1 for writing. ‘over’ obtained from a literal codeword is written at RAM position 3. • The current adaptation vector shifts the pointer array and the next adaptation vector.
7	<ul style="list-style-type: none"> • Compressed code indicates full match at location 0. • The next adaptation vector is generated as defined by location 0. • The pointer array contains address 3 at location 0 for reading. The uncompressed code ‘over’ is read from the memory. • The current adaptation vector points at location 3 in the pointer array. The pointer array contains address 1 at location 3 for writing. ‘over’ is written at RAM position 1. • The current adaptation vector shifts the pointer array and the next adaptation vector.
8	<ul style="list-style-type: none"> • Compressed code indicates partial match at location 2. • A partial match sets to 1 all the bits in the next adaptation vector. • The pointer array contains address 2 at location 2 for reading. The uncompressed code ‘hung’ is read from the memory. ‘hung’ will be used to partially reconstruct the compressed code as indicated by the match type to obtained ‘_ung’. • The current adaptation vector points at location 1 in the pointer array. The pointer array contains address 3 at location 1 for writing. ‘_ung’ is written at RAM position 3. • The current adaptation vector shifts the pointer array and the next adaptation vector.

Table 7.1.c. RAM-based decompression description (End)

7.3 Width Adaptation Logic

The use of a different bus width for the uncompressed data port (32 bits) and compressed data port (64 bits) complicates system integration. A single data bus width will enable the device to form part of a data path with minimum disruption to the original system. The variable nature of the data flow in the compressed port needs also to be addressed. Compressed data is requested or produced at discrete instants. A buffering function in the compressed port will smooth the data flow in an out of the device efficiently using the external system bus. The uncompressed port does not have a variable data rate but a constant and independent rate of

32 bits per cycle. This means that the device will consume 32 bits of uncompressed data every clock cycle during compression and it will produce 32 bits of uncompressed data every clock cycle during decompression. A buffering function is not needed in the uncompressed data port because of its synchronous nature. Figure 7.3 shows the architecture of the width adaptation logic in the compressed port. This logic serves a dual purpose. It transforms the 64-bit data bus from the compression engine or to the decompression engine into a more manageable 32-bit data bus. It also buffers the data smoothing the compressed data flow. A total of 4 Kbytes of RAM are present in this logic. The compression section uses 2 Kbytes and the decompression section uses the other 2 Kbytes. Both sections are completely independent to allow simultaneous operation in full-duplex mode. The compression buffer is organized in 2 blocks of 256 locations and 32 bits per location. The compression engine writes 64 bits of data in parallel to the 2 blocks. 32 bits of data are read from memory each cycle alternating read operations on each block. A threshold value determines how many 64-bit compressed words must be available in the buffer before compressed data is output to the 32-bit compressed bus. The decompression buffer has an equivalent organization but this time 32 bits of data are written each cycle to each block alternatively. Data is read from the buffer to the decompression engine 64-bit at a time. A threshold value controls how many 64-bit words of compressed data must be available in the decompression buffer before the decompression engine is activated. The threshold value offers a compromise between a smooth data flow using a high threshold setting or a small latency using a low threshold setting. The width adaptation logic comprises the following components:

- *Address read, Address write:* Counters that generate the read and write addresses for the coding. The write address must always precede the read address otherwise invalid data is output from the buffers.
- *RAM 256x32:* The buffers are organized in 4 blocks to enable a direct interface of the coding buffer with the compression engine. The RAM is fully-synchronous dual-port RAM so reading and writing operations can be done simultaneously.
- *Coding buffer control unit, Decoding buffer control unit:* These control units are used to enable the reading and writing of the memories when required and they also detect possible overflow and underflow conditions in the buffers.

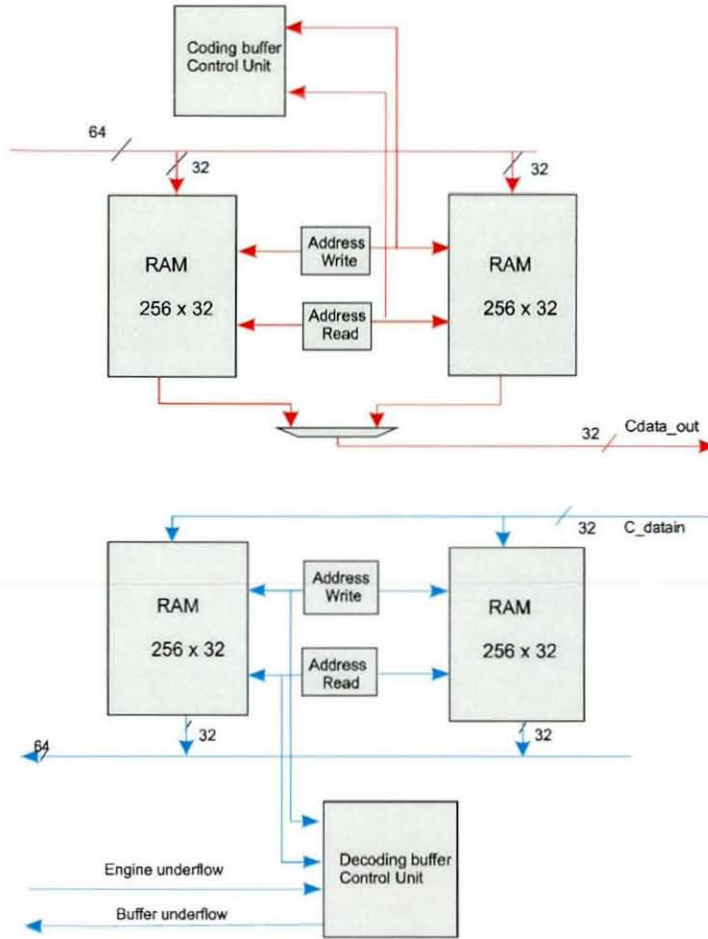


Figure 7.3.Width adaptation logic.

7.4 Full-duplex X-MatchPRO architecture

Figure 7.4.a depicts the global X-MatchPRO full-duplex compression architecture that comprises 3 major components: Compression Model, Coder, and Packer. Figure 7.4.b depicts the global X-MatchPRO full-duplex decompression architecture that also comprises 3 major components: The Decompression Model, Decoder and Unpacker. The model of section 6.3 has been split into 2 independent entities to accommodate the full-duplex processing: The CAM-based compression model that uses the compression elements of the section 6.3 model and the RAM-based decompression model of section 7.2. The Coder and Decoder architecture remains unchanged from section 6.4 but the RLI counter that was initially shared by both components has been duplicated to enable simultaneous operation of the compression and decompression channels. The Packer and Unpacker components of section 6.5 have been extended to include the width adaptation logic of section 7.3.

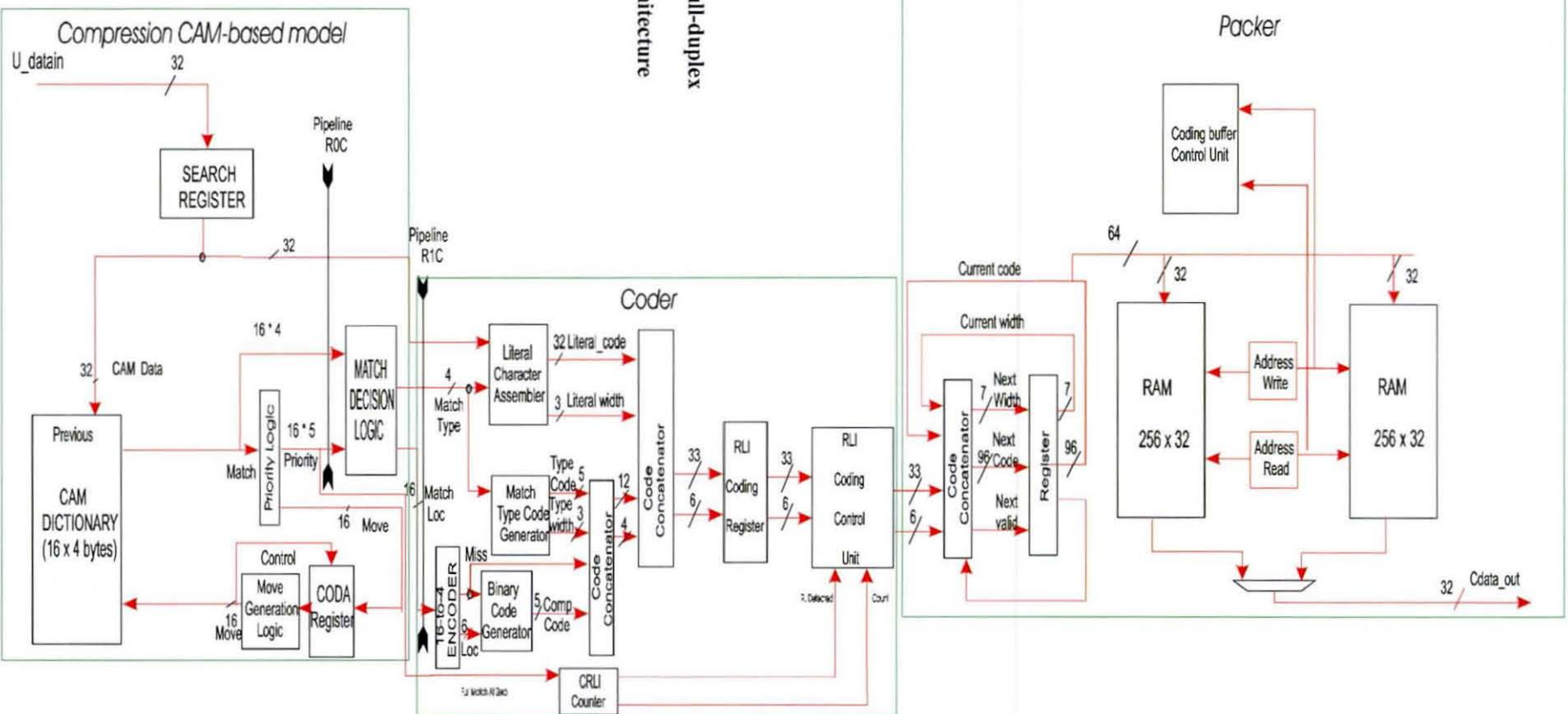


Figure 7.4.a
X-MatchPRO full-duplex
compressor architecture

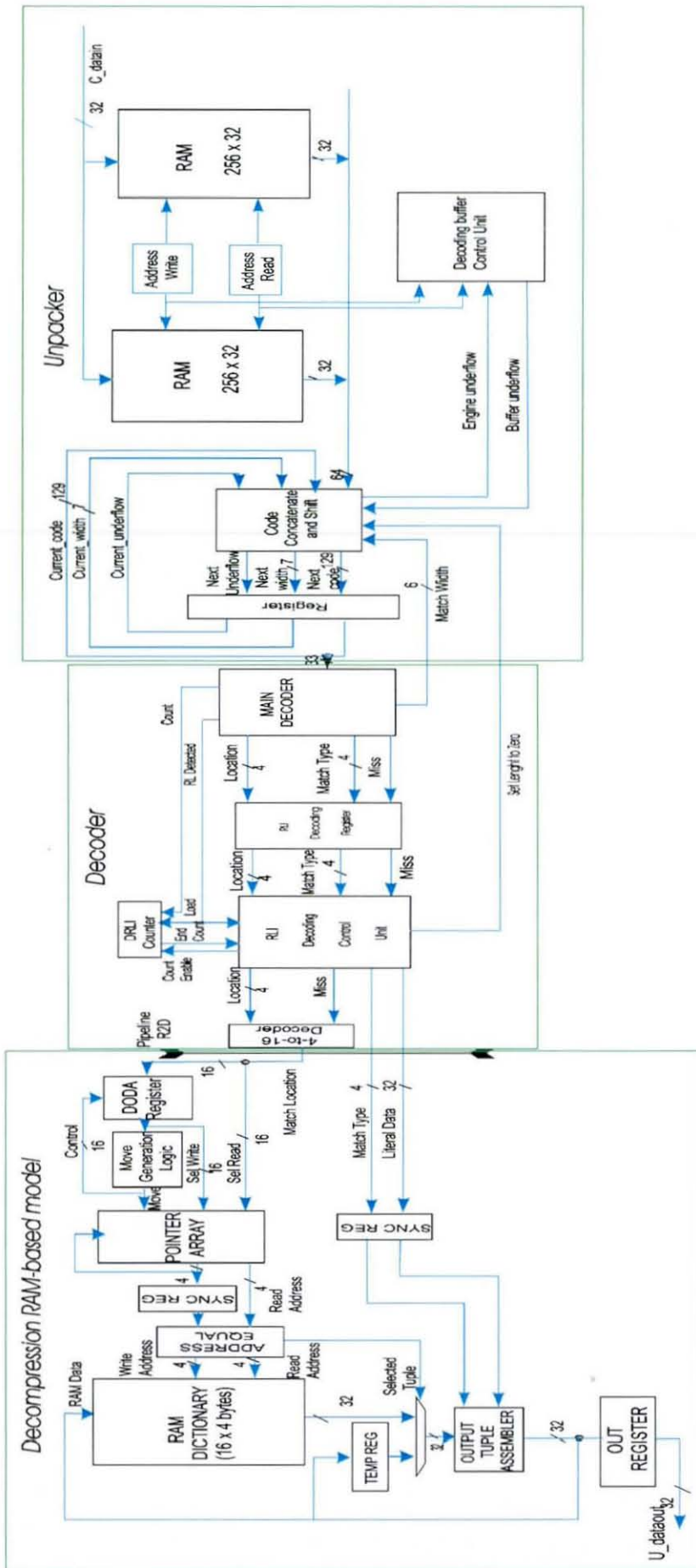


Figure 7.4.b
X-MatchPRO full-duplex decompressor architecture

7.5 X-MatchPRO operation

The architecture of chapter 6 lacks an appropriate coprocessor-style system interface where a main CPU can issue compression and decompression commands to the compressor, monitor the compression/decompression operation, and communicate with the device using a single control bus.

7.5.1 X-MatchPRO interface

Figure 7.5 illustrates the new X-MatchPRO interface.

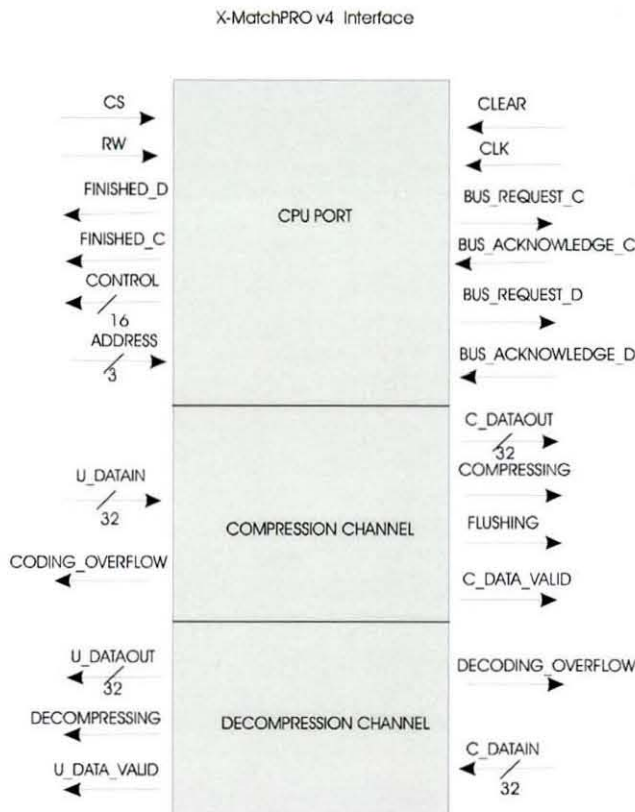


Figure 7.5. X-MatchPRO interface.

X-MatchPRO uses a simple coprocessor style interface to communicate with the rest of the system. Compression and decompression commands are issued through a common 16-bit control bus. A 3-bit address is used to access the internal registers that store the commands plus information related to compressed and uncompressed block sizes. A total of 6 registers form the register bank. 3 registers are used to control the compression channel and the other 3 for the decompression channel. The first bit in the address line indicates if the read/write operation accesses compression or decompression registers. The chip is designed to compress

any block size ranging from 8 bytes to 32 Kbytes. A decompression operation can be requested in the middle of a compression operation and vice versa. Table 7.2 describes the functionality of these signals. There are a total of 162 pins in the device. All the signals are active low and fully synchronous.

Signal name	Direction	Width	Function
CS	IN	1	Enable access to the internal registers.
RW	IN	1	Enable reading or writing the internal registers.
ADDRESS	IN	3	Internal register address.
CLK	IN	1	System clock. Positive edge active.
CLEAR	IN	1	Asynchronous clear of all the storage elements.
BUS_ACKNOWLEDGE_C	IN	1	The system grants the compressed data out bus.
BUS_ACKNOWLEDGE_D	IN	1	The system grants the compressed data in bus.
BUS_REQUEST_C	OUT	1	The chip requests the compressed data out bus. Compressed data ready to be output.
BUS_REQUEST_D	OUT	1	The chip requests the compressed data in bus. The chip request compressed data to be decompressed.
FINISH_C	OUT	1	The chip signals end of a compression operation.
FINISH_D	OUT	1	The chip signals end of a decompression operation.
CONTROL	INOUT	16	Common control bus to issue compression and decompression commands to the chip. The control bus is also used to write or read the compressed and uncompressed block size registers if required.
U_DATA_IN	IN	32	Uncompressed data input during compression.
C_DATA_OUT	OUT	32	Compressed data output during compression.
CODING_OVERFLOW	OUT	1	Data overflow in the coding buffers. Error condition
C_DATA_VALID	OUT	1	Valid compressed data present in the compressed data out bus.
COMPRESSING	OUT	1	Compression engine active.
C_DATA_IN	IN	32	Compressed data input during decompression.
U_DATA_OUT	OUT	32	Uncompressed data output during decompression.
FLUSHING	OUT	1	Compression engine inactive emptying the coding buffers.

Table 7.2.a. Chip pin-out.(Continued next page)

Signal name	Direction	Width	Function
DECODING_OVERFLOW	OUT	1	Overflow in the decoding buffers. Stop inputting uncompressed data until the bus is requested again. Engine continues decompressing data. No error condition.
U_DATA_VALID	OUT	1	Uncompressed data valid in the uncompressed data out bus.
DECOMPRESSING	OUT	1	Decompression engine active.

Table 7.2.b. Chip pin-out. (End)

7.5.2 Register bank description

A total of 6 registers form the register bank that controls the compression/decompression engines and coding/decoding buffers. These registers are accessed using the address bus and the control bus and can be read or written. Table 7.3 and Figure 7.6 show the format of these registers.

Address	Channel	Register	Function
000	Decompression	CRD Command Register Decompression	Activates or stops the decompression channel
001	Decompression	UBSRD Uncompressed Block Size Register Decompression	Sets the number of bytes of the uncompressed block after decompression
010	Decompression	CBSRD Compressed Block Size Register Decompression	Sets the number of bytes of the compressed block before decompression
100	Compression	CRC Command Register Compression	Activates or stops the compression channel
101	Compression	UBSRC Uncompressed Block Size Register Compression	Sets the number of bytes of the uncompressed block before compression
110	Compression	CBSRC Compressed Block Size Register Compression	Sets the number of bytes of the compressed block after compression

Table 7.3. Register access description.

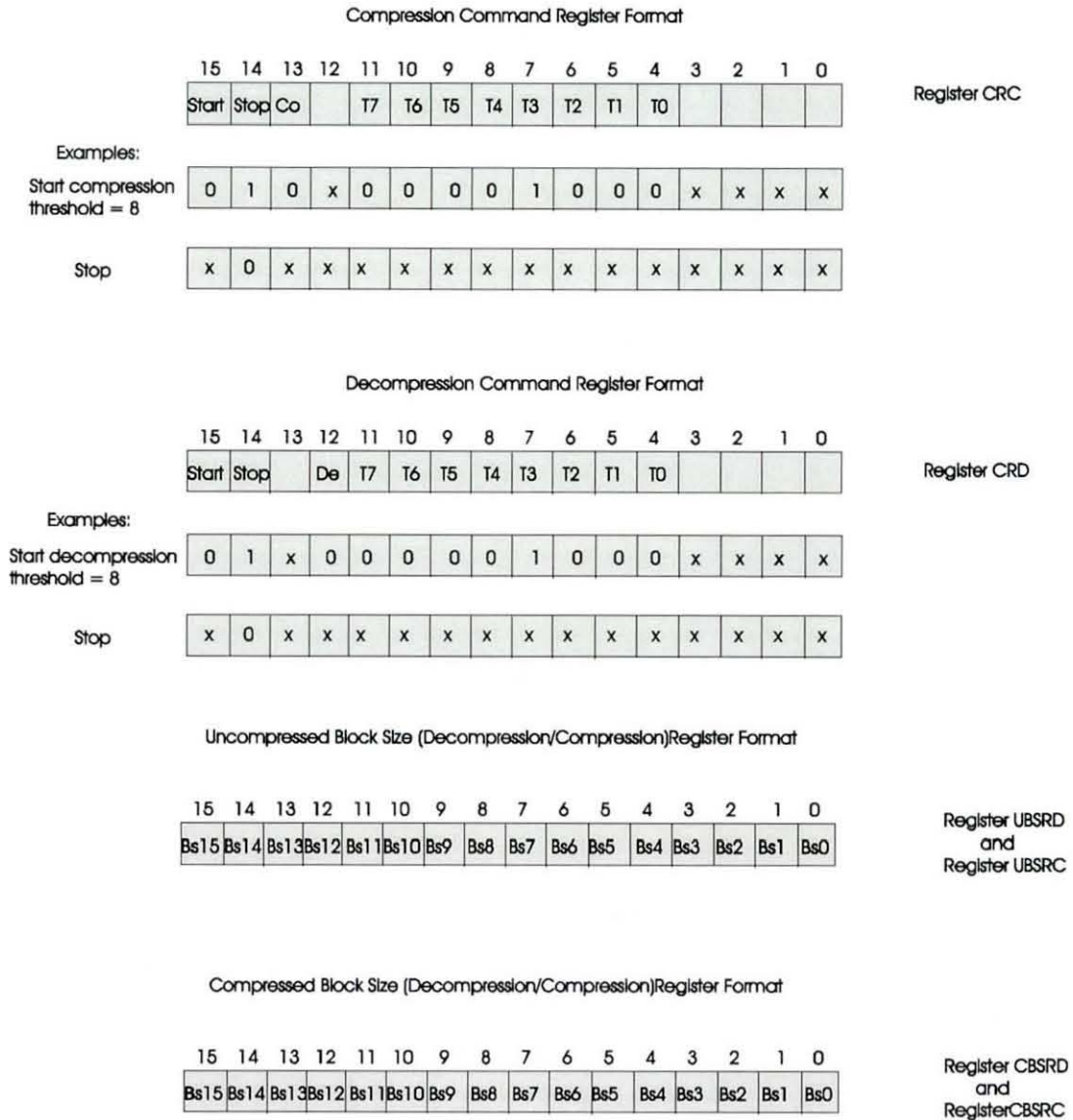


Figure 7.6. Register format.

7.6 X-MatchPRO threshold value

The threshold value is input with the command and written in the command register. It defines a programmable latency. A small value means a low latency but it is more probable that coding and decoding underflows will take place. A larger value introduces more latency but these conditions are not so frequent. The reason for coding underflows with small threshold values is that during compression the coding buffer is emptied very rapidly if little data is present when the read operation starts. In decompression the underflow can take place if the buffer is emptied because the data expanded instead of compressed during the compression operation. This means that the decompression engine consumes more data than

can be written in the buffer and eventually the buffer becomes emptied. After an underflow in the coding or decoding buffer the threshold value also defines the distance between write and read addresses before more compressed data is output or requested respectively. Underflow conditions are not error conditions but they will generate bubbles where valid data is not present in the compressed or uncompressed data out streams during compression or decompression respectively.

The threshold can have any value between 1 and 128. A threshold of 1 implies minimum latency => 1x64 bits of data are written in the buffer before the bus is requested during compression to output compressed data or before the decompression engine is started to produce uncompressed data during compression. A threshold of 128 implies maximum latency or blocked operational mode => 128x64 bits of data are written in the buffer before the bus is requested during compression to output compressed data or before the decompression engine is started to produce uncompressed data during decompression.

7.7 X-MatchPRO latency

In compression latency is defined as the number of cycles found between the moment the compression engine stops inputting data and the coding buffers finish emptying the buffers (=> chip ready to start a new operation). The compression latency has 2 components one fixed and one variable. The fixed component of 4 cycles is defined by the levels of registers located between the input search register and the coding buffers (5 levels) and the variable component is defined by how much data is present in the internal buffers when the compression engine finishes its operation (flushing operation). The probability of having a long flushing operation is small when the threshold value setting is small. This variable component depends, however, in the input data. If the data expands the latency will grow because more data will be left in the buffers to be output during the flushing operation.

In decompression latency is more predictable. Latency can be defined as the number of cycles that elapse between the first tuple of compressed data enters the chip and the first tuple of uncompressed data leaves the chip. There are again 2 components but both are fixed. The levels of registers (5 levels) between the decoding buffers and the output register in the device introduced a fixed component of 4 cycles. The decoding buffer introduces the other component and it depends on the threshold value. A threshold value of 8 introduces a latency of 16 because 16 32-bit tuples must be written in the buffer before the number of 64-bit words exceeds the threshold value and the decompression engine is activated.

7.8 X-MatchPRO operational modes

The following figures show the device running in half-duplex mode. The letter *C* should be added to the control signals: *bus request*, *bus acknowledge* and *finished* and registers *CR*, *UBSR* and *CBSR* for the compression channel and *D* for the decompression channel to obtain the full-duplex equivalents.

7.8.1 Compression mode

Figure 7.7 corresponds to a typical compression operation. To start a compression operation the CPU must write 2 registers: The uncompressed block size register (*UBSR*) must be written first and the command register (*CR*) must be written second. The *UBSR* tells the compression engine when it must stop after processing all the bytes of data present in the block. The *UBSR* specifies the number of bytes present in the block and can be any value between 8 and 32768. The *CR* puts the device in compression mode and it also contains the threshold value to control the coding buffer. The chip requests the compressed bus when the number of 64-bit words available in the coding buffer is larger than the threshold value.

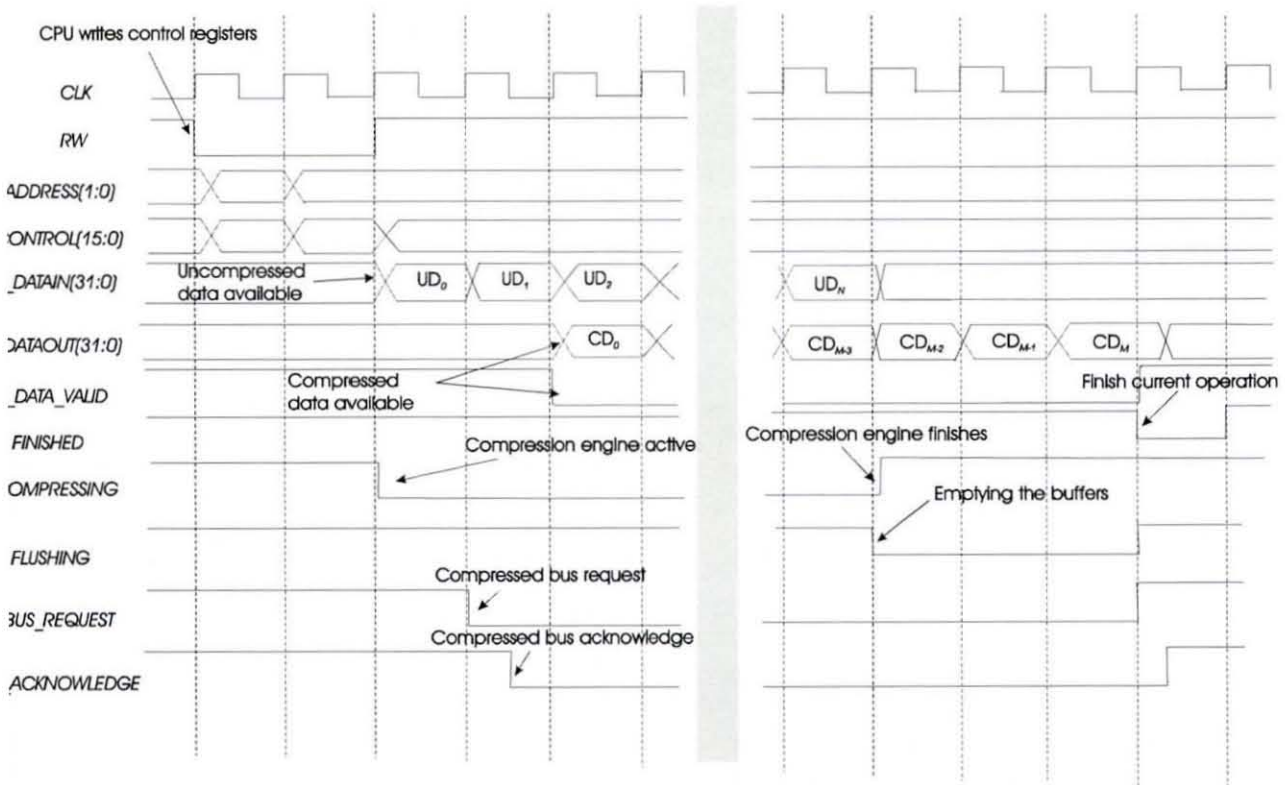


Figure 7.7. Compression operation

The system is responsible to set a new 32-bit of uncompressed data in the uncompressed bus in the immediate cycle after the *CR* is written and in every cycle thereafter. When the device produces compressed data in the compressed bus it asserts the *compressed data valid* signal active. The engine is known to be active because the *compressing* signal is active. The chip stops processing data when the value stored in *UBSR* is reached. Then a *flushing* signal is activated to indicate that any remaining compressed data in the coding buffers is being flushed out. When the buffers are emptied of their contents the device asserts the signal *finished* active for one cycle. The system can read the compressed block size register (*CBSR*) at the end of a compression operation to obtain the resulting compressed block size in bytes. This value could be compared with the original uncompressed block size to evaluate the compression efficiency. After this cycle the device is ready to start a new compression operation.

7.8.2 Decompression mode

Figure 7.8 shows a typical decompression cycle. To start a decompression operation the system must write 3 registers. The *UBSR* and the *CR* have the same function as in compression. The *CBSR* must be written with the value of the compressed block size that the decompressor is going to process. This must be done to avoid the decoding buffer requesting more data when the decompression engine is still running but all the data has already been written in the decoding buffers. Alternatively the register could be set to *FFFF*. This means that when the system denies the bus the device will assume that all the compressed data is present in its internal buffers.

The device requests the bus with the *bus request* signal and the bus is granted with the *bus acknowledge* signal. The decompression engine is activated when the number of 64 bit words of compressed data in the decoding buffer is larger than the threshold value. The *bus request* during decompression is equivalent to a compressed data request. Once the bus is granted the system is responsible to make available 32 bits of compressed data per cycle as long as the bus request signal is maintained active. The *bus acknowledge* signal cannot go inactive until all the compressed data has been loaded in the chip. The device uses the event of the *bus acknowledge* signal going inactive to know when all the compressed data is present in its internal buffers.

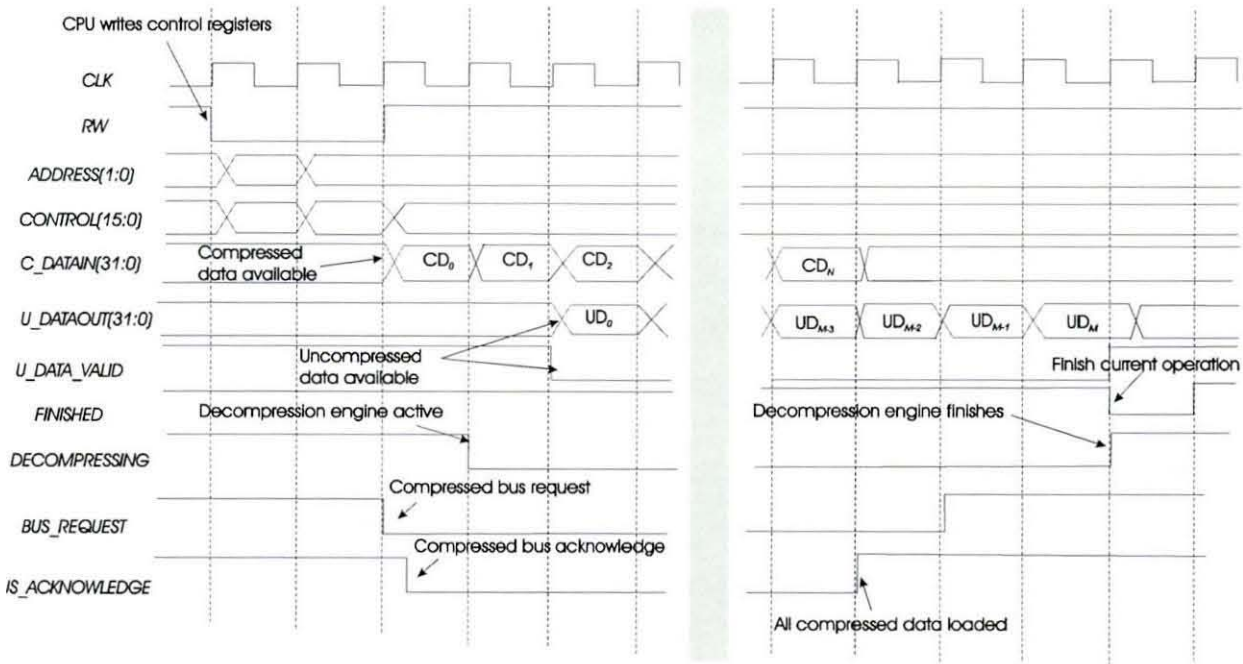


Figure 7.8. Decompression operation

7.8.3 X-MatchPRO special conditions

7.8.3.1 Buffer Coding Overflow

A coding overflow condition should never be encountered under normal operating conditions. It can never happen if these 2 conditions are met: the uncompressed block size is less than 32 Kbytes and once the compressed bus is requested during compression it is granted in less than $256 - \text{threshold_value}$ cycles. This value is obtained after solving the following 2 equations:

$$33xTe + 1xTb \leq 256x64 \leq 16384 \text{ bits} \tag{7.1}$$

$$32x(Te + Tb) \leq \text{block size} \leq 32768x8 \leq 262144 \text{ bits} \tag{7.2}$$

These 2 equations can be simplified to:

$$33xTe + Tb \leq 16384 \text{ bits} \tag{7.3}$$

$$Te + Tb \leq 8192 \text{ bits} \tag{7.4}$$

Where Te is the number of cycles the engine is compressing data but the buffer is not outputting data whilst Tb is the number of cycles the engine is compressing data and the

buffer is also outputting data. The block size is set to a maximum 32768x8 bits. The value of T_e is 256 after solving the equations.

If this condition is not met an overflow could take place in the coding buffer. This is an error condition detected by the *coding overflow* signal going active. This means that there is no room in the buffers to write new compressed data being produced by the compression engine and the operation fails. This situation could arise if the engine is continuously expanding the data instead of compressing the data because in that situation the engine produces 33 bits of data per cycle but only 32 bits of data can be read from the buffer per cycle.

7.8.3.2 Buffer Coding underflow

Coding underflow cannot be considered a special case because it is the normal consequence of compression. If compression is taking place the coding buffer outputs data faster than it receives data from the compression engine. With a typical compression ratio of 0.5 the engine writes on average 16 bits of data to the coding buffer per cycle and 32 bits of data are read from the buffer per cycle. This underflow condition is signal with the *bus request* signal going inactive and the *compressed data valid* signal going inactive. The bus will not be requested again until the number of valid 64-bit compressed words in the coding buffer is bigger than the threshold value.

7.8.3.3 Decoding Buffer Overflow

Buffer decoding overflow is an occasional condition that can take place when compression is very good. In this case the decompression engine consumes little data but the decoding buffer gets 32 bits of compressed data each cycle from the compressed bus. If a decoding overflow takes place the decompression engine keeps working at full speed unaware of the overflow condition in the buffer. The device stops requesting the bus (stops requesting data) and this is indicated by *bus request* signal going inactive. The buffer will request more compressed data once the gap between the write address and the read address is bigger than the threshold value. The system must stop putting compressed data in the compressed bus since this data will not be written to a buffer under an overflow condition.

7.8.3.4 Decoding Buffer Underflow

A decoding buffer underflow is an infrequent condition that could take place when the decompression engine requests compressed data to the decoding buffer but no data is

available in the buffer to satisfy this request. This condition can happen with data expansion. In this case the decompression engine consumes 33 bits of data per cycle but the buffer can only get 32 bits per cycle. After some cycles the decompression engine request data to the buffer but the buffer is empty. Under these circumstances the decompression engine empties its pipeline and maintains its current state and data request until compressed data is available from the buffer. The engine stops uncompressing data until more data is available. The uncompressed port will see a few cycles where no uncompressed data is available. The *uncompressed data valid* signal will go inactive to indicate this condition.

It is important to notice that a decompression engine underflow is a different condition from a decoding buffer underflow. A decompression engine underflow is a normal internal condition that could generate a decoding buffer underflow if the buffer is empty. A decompression engine underflow happens when fewer than 66 bits are valid in the 129 bit ($65+64 = 129$) decompression register. A special case is when the decompression engine underflow can not be satisfied from the buffer because all the compressed data in the block has been written in the buffer but it is now exhausted. This is a normal termination of the decompression operation and it does not generate a decoding buffer underflow. The decompression engine must continue to decode the last few bits of compressed data (<66 bits) remaining in the decompression register until 0 bits are valid. This termination condition is controlled by the *bus acknowledge* signal going inactive or by an internal counter reaching the value stored in the compressed block size register. Buffer decoding underflow generation is disabled when the device reaches this termination condition.

7.9 FPGA-based X-MatchPRO: complexity and performance

Table 7.4 shows a summary of the FPGA-Based X-MatchPRO family targeting Actel, Altera and Xilinx FPGA's. These data was obtained after mapping the design to each technology using a synthesis engine and then performing placing and routing using vendor-specific tools. The figures shown in Table 7.4 were extracted from the post-layout reports provided by the place&route tools.

The validity of these timing reports was verified using backannotated simulation and a full test vector data set formed by around 100k vectors. These vectors were obtained from a cycle accurate C++ model of X-MatchPRO and were specifically designed to test all the operating modes and special conditions of the device.

Technology												
Manufacture	Process	Complexity						FPGA details			Speed (Gbit/s)	Clock (MHz)
		Full-duplex X-MatchPRO										
		Dictionary length	Logic units	% Elements	Memory bits	% Memory	Gate count equivalent (K)	Part	Max system gates (K)	Logic units (K)		
Actel	0.25 um Flash-CMOS FPGA	16	9039 Tiles	70	40960	100	214	Pro ASIC/ A500K1 30BG456	287	12.8 Tiles	0.8	25
Altera	0.18 um SRAM-CMOS FPGA	16	5063 LE's	60	40960	38	N/A	Apex/ 20K 200EFC4 84-1	526	8.32 LE's	1.6	50
Xilinx	0.18 um SRAM-CMOS FPGA	16	5295 LC's	55	40960	25	210	Virtex/ XCV 400EBG 432-8	570	9.6 LC's	1.6	50

Table 7.4. X-MatchPRO technology.

Logic unit in column 4 is the basic logic unit in the architecture of the selected technology. The complexity of this logic unit varies among the different technologies. Actel ProASIC devices [Actel00] use a logic unit call Tile, Altera call these units logic elements (LE's) [Altera01] and Xilinx call them logic cells (LC's) [Xilinx01]. Actel ProASIC tiles are simple blocks that can implement a logic function with 3 inputs and 1 output such as an AND gate or a flip-flop. Actel ProASIC architecture is very flat and tiles are repeated across the device forming a matrix of identical logic elements. Dedicated memory blocks are grouped in one the sides of the device. Each memory block can implement 2304 bits of fully-synchronous dual port RAM. Xilinx Virtex architecture uses a more complex LC that includes a 4 input Look-Up Table (LUT), a carry function and a storage element. A Configurable Logic Block (CLB) is formed by 4 of these LC's plus some extra logic (1 CLB is equivalent to 4.5 LC's). The CLB's are repeated across the device forming a matrix of logic. Additionally dedicated

memory blocks are inserted at different positions in the CLB matrix. Each Virtex memory block can implement 4096 bits of fully-synchronous dual port RAM. Altera Apex LE's are quite similar to Xilinx LC's. Each Apex LE has a 4 input LUT, a storage element, carry chain logic and some extra functions. A group of 10 LE's forms a Logic Array Block (LAB) that also includes an Embedded System Block (ESB). Each ESB can be used to construct a variety of memory functions and includes up to 2048 memory bits. A group of 16 LAB's forms a MegaLAB that are repeated across the device.

FPGA vendors use different measurements to translate their technology components into gate equivalents and, in general, it is more accurate to give complexity in terms of FPGA elements and usage percentage. When available the gate count equivalent obtained from the place&route tool is given for reference in column 8. The total gate count includes memory and logic but only around 14% of the gates (30 K) correspond to logic whilst the rest 86% (180 K) correspond to the gate count equivalent of the 40 Kbits of memory. The percentage of column 5 measures how much of the logic available in the selected FPGA part given in column 9 is used by the design. This measurement only refers to utilization of logic elements and not embedded RAM. The amount of embedded RAM used by the designs measure in bits is given in column 6 whilst the percentage of memory utilization is available in column 7. The throughput of column 12 measures the raw uncompressed data throughput of the device. These are the number of bits of raw uncompressed data that will be consumed by the device during compression or produced during decompression. These figure is obtained multiplying the clock frequency of column 13 times the number of bits processed by cycle (32). The throughput performance of the Altera and Xilinx devices is comparable because they are similar SRAM-based technologies using the same feature size. The number of logic elements used in both technologies is very similar. This means that LC's and LE's perform very similar functions. Table 7.4 shows that the complexity of the Xilinx Virtex device and Altera Apex devices are comparable. More logic elements are needed in the Actel ProASIC devices because their Tiles are smaller than LC's or LE's and they can implement only a simple logic function each of them. The most efficient implementation in terms of area is achieved in the Actel device since this device is roughly half the size of the other 2 FPGA's in terms of maximum system gates and it would not be possible to fit the design in the Altera or Xilinx chips if only half of the current resources were available. The performance of the Actel implementation is lower than the other 2 devices due to 2 main reasons: Firstly the feature size is bigger which degrades performance, secondly the routing complexity increases in these fine granularity devices because they lack the architectural hierarchy of the other 2 FPGA's [Betz98]. ProASIC architecture is flat so the routing scheme is more complex.

Table 7.5 presents a summary of the X-MatchPRO features compared against the ASIC compressors selected in section 4.3. The table compares different ASIC technologies but these correspond to the fastest silicon currently available from the respective manufactures. The ASIC figures have been obtained from the data sheets provided by the manufactures while our own figures are based on post-layout reports. The Hi/fn 9600 device is implemented in the most advance technology in the list and offers the highest throughput among the ASIC devices. This architecture is able to process one byte per clock cycle and its throughput in bits per second can be readily obtained multiplying the clock frequency times 8. This is also true in the IBM device but in the case of the AHA devices the previous value has to be divided by two because their less efficient internal architecture needs two clock cycles to process each byte.

It is possible to perform a direct comparison of the Hi/fn 9600 that is based on a 0.35 um ASIC technology an clocks at 80 MHz with the Hi/fn 9610 (see table 2.2). The Hi/fn 9600 implements the same LZS algorithm but it is based on an older 0.5 um technology and clocks at 50 MHz. Therefore, an increase in throughput of 60% is achieved migrating from 0.5 um to a 0.35 um feature size. Further reductions in feature size should increase the clocking frequency of the device but it is also important to take into account that interconnect overheads and deep sub-micron effects mean that the speed-up factor is not linear. The IBM device can achieve a similar clocking frequency of 100 MHz if mapped to a comparable 0.35 um technology as reported in IBM literature [Craft98]. In general, these two LZ1 derivatives achieve a similar throughput because they are based on the same LZ1 algorithm and they are limited by the fact the only 1 byte is processed per cycle. The main advantage of X-MatchPRO is that 4 bytes and not 1 byte are processed in each clock cycle.

The table shows that X-MatchPRO exceeds by a factor of 2 the throughput of the other ASIC compressors. It is expected that X-MatchPRO throughput will improve by a factor of 2-3 [Betz98] if replacing the FPGA technology for an ASIC technology with a similar feature size. This means that X-MatchPRO based on an ASIC should be able to match the clock frequency of any of the other previous ASIC's if implemented in the same technology. A throughput gain of a factor of 4 will be obtained by X-MatchPRO under these circumstances thanks to its ability to process 4 symbols per clock cycle.

It is also interesting to compare the X-MatchPRO design with the previous X-Match design in terms of throughput. The original X-Match design has a critical path in the search and adaptation process that limits its performance to 6 MHz (192 Mbits/s throughput) in a 0.6 um ProASIC FPGA technology as seen in our paper [Nunez99]. This is a direct implementation

of X-Match in an FPGA with little architectural enhancements. Our research reveals that this performance improves to 14 MHz (448 Mbits/s throughput) when targeting the X-Match design to the more up-to-date 0.25 um ProASIC FPGA used by X-MatchPRO in table 7.4. X-MatchPRO does not contain a critical path in the search and adaptation process thanks to ODA as seen in section 6.3 but it is limited by the unpacker/decoder feedback loop as seen in section 6.5. The new critical path limits the performance of X-MatchPRO in a ProASIC technology to 25 MHz (800 Mbits/s). This is approximately twice the throughput of the original X-Match architecture (448 Mbits/s → 800 Mbits/s) .

DEVELOPERS		IBM	Advance Hardware Architectures (AHA)		Hi/fn	System Design Group Loughborough University		
CHIP		ALDC1-40S	AHA 3521	AHA 3231	Hi/fn 9600	X-MatchPRO		
TECHNOLOGY DETAILS	PROCESS	IBM CMOS 0.8 micron triple-level gate array/std cell	0.5 micron CMOS	0.5 micron CMOS	0.35 micron gate array/std cell	0.18 micron SRAM-CMOS FPGA Xilinx VIRTEX-E	0.18 micron SRAM-CMOS FPGA Altera APEX20KE	0.25 micron FLASH-CMOS FPGA Actel A500K ProASIC
	COMPLEXITY (16-word dictionary)	70 Kgates	N/A	N/A	100 Kgates	5367 LUT's 55 % of a XCV400EB G432-8	5040 LC's 60 % of a EP20K200 EFC484-1	9039 TILE's 70% of a A500K130-BG456
	CLOCK (MHz)	40	40	40	80	50	50	25
THROUGHPUT (Mbits/s)		320	160	160	640	1600	1600	800
FULL-DUPLEX PERFORMANCE (Mbits/s)		N/A	N/A	N/A	1280	3200	3200	1600
ALGORITHM		ALDC	ALDC	DCZL	LZS	X-MatchPRO	X-MatchPRO	X-MatchPRO

Table 7.5. X-MatchPRO comparison.

7.10 Conclusions

This chapter has extended the compression/decompression engine of chapter 6 by adding a suitable system interface and a buffering function. Moreover, a highly compact full-duplex implementation has been obtained by mapping the decoding dictionary to embedded RAM instead of distributed flip-flops so the complexity of the half-duplex and full-duplex devices is comparable in terms of logic gates. The resulting design has been implemented and its functionality proved to be correct using timing simulation in 3 different FPGA technologies. The multiple technology implementation qualifies the design as portable. The performance figures of the FPGA-based X-MatchPRO exceed those of other ASIC compressors and match the requirements of chapter 3.1.

Chapter 8

Conclusions

8.1 Objectives of chapter

This chapter concludes this thesis with a summary of the research objectives, an evaluation on how well we achieve those objectives, the limitations of the current work and finally a proposed path for future research.

8.2 Summary of the objectives and the research flow

As stated in chapter 1 this thesis aimed to advance the field of lossless hardware data compression by providing higher throughputs and better compression ratios. The motivation for this research was found in that current solutions do not provide the levels of performance required in high-speed communication and storage applications. Lossless data compression is currently a tool commonly used to double the bandwidth and storage capacity of systems running in the order of Mbits/s such as wide area networks in communication applications and tape drives in storage applications. Its usage in systems that involve higher transfer rates is not as popular because of the performance impact that the compression process introduces. The same benefits should be expected if properly deployed in applications where data movement is measured in Gbit/s such as RAID drives and local area networks.

After establishing the usefulness of Gbit/s lossless data compression hardware in chapter 1 the research continued with an analysis of the current state of lossless data compression in chapter 2. Chapter 2 reviewed recent advances in software and hardware compression analysing the benefits and limitations of each method. Chapter 3 continued with the selection

of the X-Match hardware-friendly algorithm because it exhibited high-performance features including parallelism, single cycle execution and low latency. Chapter 3 was used as a pivotal point that clearly specified the starting point of our investigations. Chapter 4 described the experimental framework as a set of tools to be used to carry out the investigations. This set of tools included the data sets to be used in the compression efficiency measurements and a selection of lossless data compression methods representative of high performance software and hardware-based compression. Chapter 5 focused on compression efficiency analysis and optimisation using the data sets and methods of Chapter 4. It studied ways of increasing model and coder efficiency without affecting throughput. A dictionary-based approach was used because of its inherent simplicity and hardware amenability. Chapter 6 focused on increasing the performance throughput of the hardware architecture without affecting the compression ratio. Chapter 6 produced a new core architecture for the compression and decompression engines. The architecture was mapped and verified in ProASIC FPGA technology, selected as a silicon test-bench, to prove the high performance characteristics of the design. Chapter 7 extended the core developed in chapter 6 to a full-duplex self-contained coprocessor architecture named X-MatchPRO. X-MatchPRO was efficiently mapped to 3 FPGA devices from 3 different manufactures. Post-layout backannotation was used to obtain exact data on performance and complexity.

8.3 Summary of the X-Match compression method

The X-Match design of chapter 3 describes the basic architecture of a high-performance lossless data compressor based on storing data commonly seen in a dictionary and matching incoming data with data present in the dictionary. A move-to-front adaptation policy is used to maintain dictionary efficiency avoiding storing duplicated data words. The dictionary is based on a CAM circuit that allows single cycle search and adaptation. The CAM feature that enables configuring its columns as selectively shiftable registers implements the move-to-front technique. The data words called tuples are fixed in width with 4 bytes per data word. The width of 4 bytes is found to be optimal generating more compression than other alternatives whilst it naturally maps to a parallel high-throughput architecture. The dictionary length grows from an initial value of 0 to a maximum value of 128 each time a tuple is not fully matched in the CAM. A partial matching (X-matching) strategy is used to improve compression so only 2 bytes out of maximum of 4 are required to match for the dictionary hit to be considered valid. A match is coded as a single bit set to 0 followed by a PBC (Phased Binary Code) indicating the match location followed by a Huffman code indicating a match type and any non-matching characters in literal form. A miss is coded as a single bit followed

by the 4 non-matching characters in literal form. Combinatorial searching strategies (where the byte at location n in the search tuple is allowed to match a dictionary byte located at a position different to n in the CAM) do not improve compression but affect complexity and throughput. The CAM circuit supports compression and decompression but not simultaneously. Pre-layout results after mapping the design in a 0.6 μm gate array technology shows a data independent throughput of 100 Mbytes/s clocking at 25 MHz with complexity around 100 Kgates.

8.4 Main contributions achieved in this research:

The X-MatchPRO hardware

The X-MatchPRO chip developed in chapters 5, 6 and 7 describes a dual-channel full-duplex high-performance lossless data compressor coprocessor with enhanced compression and throughput features.

X-MatchPRO enhances compression ratio by adding an internal run length coding technique named RLI. In its original configuration described in Chapter 5 RLI combines with Phased Binary Coding (PBC) to obtain a compression improvement between 3-10 % depending on data sets. Chapter 5 addresses the best location for a run-length coder in X-Match with 2 options being investigated internal and front. Although the compression performance of both solutions is very similar RLI adds a very neat solution from a hardware point of view because it integrates in the architecture and shares the dictionary logic keeping complexity to a minimum. RLI is particular effective in a hardware implementation because it is not target to code repetitions of a particular data pattern but repetitions of matches in data location 0. RLI can effectively code any repeating 32-bit pattern without any data identification information because the move-to-front adaptation policy places repeating data in location 0. The last dictionary codeword is reserved to indicate RLI events which can code up to 255 4-byte repetitions in a single code. The last dictionary codeword varies in a PBC-based coder because dictionary length is variable but it is fixed in a UBC-based coder because all the dictionary locations are active after the first cycle. The maximum compression ratio enable by the combination of PBC and RLI is $10/(255*4*8) = 0.00122$ when 1020 repetitions of the same byte are found after a dictionary reset.

The move-to-front technique used in model adaptation generates a non-uniform distribution of matches that a more complex technique than uniform binary coding can used to increase compression. PBC offers slightly better performance than Huffman coding derivatives. PBC

is useful when compressing small data packets using dictionaries larger than 64 locations otherwise with dictionary sizes of 16, 32 and 64 locations simpler UBC suffices. This is because small dictionaries fill up with data quickly and PBC loses its advantage once the dictionary becomes full.

X-MatchPRO can adapt its complexity requirements to the available hardware resources trading dictionary length for compression efficiency. X-MatchPRO does not require a large dictionary because it maintains a highly efficient history state by quickly eliminating any data duplication in a single cycle. It obtains compression with dictionaries as small as 16 locations whilst a 256 locations dictionary offers the best trade-off between complexity and performance.

X-MatchPRO compression improves gradually increasing block size from 256 bytes to 4 Kbytes but remains largely invariant with further increases in block size due to dictionary saturation. Small block sizes increase the effect of locality of reference and periodically activated a technique like PBC so they suit well X-MatchPRO.

X-MatchPRO enhances throughput with a new redesigned architecture that includes an Out of Date Adaptation (ODA) policy. A critical feedback loop is identified in the search and adaptation circuitry because after a search operation, the best match must be solved and an adaptation vector generated in time t before the dictionary is ready to start a new cycle in time $t+1$. ODA breaks the critical feedback loop in the search and adaptation circuitry so the dictionary adapts at time $t+1$ with match information generated at time $t-1$. ODA does not affect compression negatively because dictionary elements are unique at all times except the dictionary element at the top of the dictionary that can be duplicated. Dictionary data duplication is restricted to location 0 and duplicated data is eliminated in a single cycle maintaining dictionary efficiency. The complexity impact of ODA is very small, requiring only a few hundred gates.

A second feedback loop is identified in the bit disassembly logic because a variable-length codeword must be decoded before new data can be added and old data eliminated from the active part of the buffer. This is characteristic of data compression methods based on mapping a fixed length symbol to a variable length codeword. Packing and unpacking is trivial when the codewords have the same length and their position in the compressed stream can be easily identified. This loop is optimised increasing the level of parallelism during the concatenation of new data and the shifting out of old data as described in chapter 6.

X-MatchPRO is a dual-channel full-duplex coprocessor architecture that includes simple interfacing, buffering functions and 2 independent compression and decompression channels that can operate simultaneously. Full-duplex functionality adds an useful feature to the design because it is becoming a characteristic of high performance networks to carry information in both directions simultaneously. The challenge is to design 2 independent channels keeping complexity to a minimum. The RAM-based decompression model achieves this by eliminating the need for an expensive shift register file to store the data. The higher priority given to matches closer to the top of the dictionary is a key technique in the full duplex architecture. It enables the RAM-based decompression dictionary to have only location 0 initialised in the first cycle as long as the same value is used to initialise in a single cycle all the locations of the CAM-based compression dictionary. A pointer array stores addresses to the dictionary locations following the same move to front strategy used by the data in the CAM-based compression model. The pointer array is a fraction of the size of the dictionary because the basic pointer word width varies from 4 to 8 bits depending on dictionary length whilst the dictionary word width is 32 bits. The elimination of the multiplexors associated to the CAM for decompression in the half-duplex implementation provides enough resources to implement the pointer array and maintains logic complexity almost constant. The decompression circuitry avoids interference between ODA and the pointer array to enable both compression and decompression dictionaries to be in synchrony at all times.

A buffering function is introduced in the packing and unpacking logic to fulfil a dual purpose. It smoothes the data flow out and in the chip in the compressed port and it allows a width adaptation from the 64 bits used out and in the compression and decompression engines respectively to a more manageable 32 bits out and in the chip.

X-MatchPRO FPGA-based hardware proves the high-performance features of the design in silicon. The FPGA-based hardware is based on a 16-location dictionary using UBC coding for the match locations to reduce the resource requirements on the FPGA prototype. A detailed post-layout verification of the compressor/decompressor core is done in chapter 6 using a Actel ProASIC FPGA as the silicon test-bench. The core is extended to a full-duplex coprocessor architecture and mapped to FPGA devices from Actel, Xilinx and Altera corporations. These multiple technologies validate the portability of the design and make use of alternative FPGA architectures with different strong and weak points. Actel ProASIC devices provide an excellent prototyping platform because they are reprogrammable and non-volatile and their high granularity technology offers a smooth migration path to ASIC technology with predictable results. Xilinx Virtex and Altera Apex devices offer an advance

process, very high densities and a sophisticated tool set to obtain very high performance. Both Altera and Xilinx implementations are very similar in logic cell count and performance.

8.5 Other contributions achieved in this research

A systematic analysis and classification of the lossless compression methods is done in Chapter 2 with special emphasis in hardware. The classification is based on dividing lossless data compression in 3 independent stages, namely: modelling, coding and packing and using the first 2 stages to structure the review. Modelling and coding separation is usually reserved to statistical methods but it can be applied successfully to dictionary-based methods as well. The following conclusions can be drawn from the first part of this research:

Compression improves by:

1. The use of high-order statistical modelling. The optimal maximum order increases with increases in symbol granularity: 1st order for word alphabets, 4th order for byte alphabets and 10th order or higher for binary alphabets.
2. The use of arithmetic coding as an optimal method to extract the redundancy identified by the model. Arithmetic coding is optimal for a given model because no other coding method can improve on it. On the other hand if the model feeding the coder is inaccurate the global performance will be poor. Arithmetic coding needs accurate modelling. If this is not the case simpler and therefore faster coding could be a better alternative.
3. PPM is one of the best compression methods currently available and it combines points 1 and 2 in a complex algorithm made possible in the last couple of decades with the arrival of powerful general-purpose processors and plentiful memory resources.
4. The use of an algorithm granularity compatible with data granularity. For example text is clearly byte oriented or word oriented and compresses better with algorithms where the basic input is 8 bits or with methods that parse the input data stream into natural words.

Throughput improves by:

1. The use of hardware amenable algorithms that do not required too many memory or logic resources to run. Application specific hardware chips based their power in single cycle

execution and high clock ratios obtained from efficiently mapping an algorithm to silicon. LZ derivatives are dominant in the field of hardware compression.

2. The use of algorithms that can offer a constant data independent throughput in the uncompress port. Throughput in the compressed port depends on the instantaneous compression ratio but the uncompressed port should be able to consume or produce the same number of uncompressed bits per cycle. Otherwise a worst case throughput measurement should be used when throughput depends on data type.
3. The use of wider symbols like bytes or words instead of bits. The definition of word can change from natural words to 4-byte words like in X-Match. Increasing the level of parallelism by widening the basic input symbol improves throughput but finding the redundancy becomes a more difficult task.
4. The use of CAM-based circuit to store the history data so fast single cycle searching can be done during compression. Systolic architectures based on pipelined CAM's where the input symbol is compared with a different position of the dictionary in each cycle can obtain higher throughputs and they have excellent scalability properties. On the other hand they suffer from high latency and this makes them unattractive in many real-time application environments.
5. The elimination of dependencies between the modelling, coding and packing processes so deep pipeline architectures can be implemented. Algorithms that map fixed length symbols to variable length codewords such as X-Match suffer from a dependency between the decoding and the unpacking process difficult to improve. LZ algorithms map variable length symbol sequences to fixed length codewords and avoid this problem.
6. Model adaptation tends to be a typical performance bottleneck in many compression algorithms because in statistical methods a set of cumulative frequencies must be incremented or a tree must be reconstructed and in dictionary methods the dictionary must be rearrange introducing new symbols and deleting old symbols.

Another contribution is the development of a compression performance database using software and hardware algorithms that correspond to state-of-the-art technology. A total of 3 different data sets are used to represent data commonly found in computer systems: the memory data set, the disc data set and the Canterbury data set. It is common in this type of research to do comparisons using obscure or out-of-grade algorithms and data sets with the

negative effect that further analysis becomes very difficult. We decided to choose state-of-the-art lossless data compression algorithms implemented in both hardware and software. The LZS, ALDC and DCLZ hardware-based algorithms are commercially successful chips used in many networking and storage applications. The PkZIP software-based algorithm is known by anybody who has downloaded a file compressed in ZIP format from the Internet and routinely used for archiving and distribution of data. HA and PPMZ software-based algorithms define the current limits of lossless data compression and illustrate how the diminishing returns rule makes any significant compression improvements in the future a big challenge.

8.6 Measurement of success

The first objective was an identification of the factors that limit or improve the performance of lossless data compression methods. The concepts of compression speed and compression ratio were used to define the performance of a method. An analysis of current compression solutions was done in Chapter 2 where it was identified that the highest throughput combined with lower latency was achieved in hardware using CAM circuits and single cycle operation. The higher compression was found in software in methods based on variable-order statistical modelling. Limitations in speed were mainly due to small symbol width like in systems based on binary alphabets. Limitations on compression were due to poor modelling or coding. It was also clear that compression and speed were highly dependent on each other with better compression done by the slowest algorithms and vice-versa.

The second and third objectives were to find solutions to these limitations and to prove them in real silicon to advance the field of lossless data compression. Our work was based on hardware and it naturally stressed the point of speed over compression. The developed X-MatchPRO lossless data compression chip offers Gbit/s full-duplex data compression performance and improved compression using the X-Match method. It can handle the data streams found in Gbit/s applications where no other solution is currently available. It achieves its objectives using low-cost FPGA technology while a custom solution is expected to obtain a typical increase in throughput of a factor of 3. It, therefore, advances the field of lossless data compression hardware and achieves the main objective of this work.

8.7 Limitations of research

Our initial research revealed that statistical modelling based on variable-order models and arithmetic coders is a compression methodology able to achieve a performance close to the

entropy limit. It seems reasonable to investigate how some of these statistical concepts can be introduced in X-Match. This path of research involves the development of parallel arithmetic coders. Parallel arithmetic coding offering incremental transmission does not have a current satisfactory solution because of the data dependencies that exist between 2 consecutive symbols. This is particularly true when analysing the decompression stage of the algorithm. Limitations in time prevented a thorough investigation of parallel arithmetic coding since it constitutes a PhD on its own.

Compression performance of the X-MatchPRO method is somehow limited mainly when targeting data textual in nature. The reason is that this data exhibits single-byte granularity and it maps badly to the 32-bit granularity of X-MatchPRO. Redundancy in this type of data is easily picked by a byte-based LZ derivative but it fails to be found by X-MatchPRO because bytes are not aligned in groups of 4. The alternative of increasing compression performance in X-MatchPRO by coding the literal characters part of partial-match codewords or misses was found to be unfeasible because of its direct impact on complexity and more important throughput.

8.8 Future work

The fabrication of a custom ASIC solution will have a positive impact on speed typically improving throughput by a factor of 3 if compared with a similar feature size FPGA. A much more compact device is possible because FPGA gates scale down considerable when translated into ASIC gates.

The integration of the FPGA-based X-MatchPRO in a real application such as Gbit Ethernet will prove an invaluable tool to verify the benefits of high-speed lossless data compression.

The development of a form of parallel arithmetic coding will open the way to a variable-order X-Match model that could achieve the best of both worlds: high speed and excellent compression.

It is also interesting the idea of a variable-width X-MatchPRO dictionary extending the concept of variable-length. This means that the algorithm would be able to adapt its internal granularity to the data granularity. For example, text compression will improve significantly if the data word width could be adjusted to the natural word width.

Further increases in throughput are possible if several X-MatchPRO are combined into a single chip. The challenge is to design a multiple compressor chip that uses the same interface as a single compressor chip so the application only sees a significant increase in performance.

8.9 Summary

This thesis has addressed the problem of high-speed lossless data compression in hardware. It has produced the X-MachPRO chip that with a combined compression and decompression performance of 3.2 Gbit/s in a Xilinx or Altera FPGA's can outperform any other ASIC chips currently available.

References

- [Acorn92] 'Is v42 the Answer?', Acorn User Magazine Comms, September, 1992.
- [Actel00] 'ProASIC™ 500K Family', Data sheet, Actel corporation, 955 East Arques Avenue, Sunnyvale, CA, 2000.
- [AHA95] 'AHA Data Compression and Forward Error Correction Standards', Application Brief, Advanced Hardware Architectures Inc, 2635 Hopkins Court, Pullman, WA, 1995.
- [AHA96] 'Primer: Data Compression (DCLZ)', Application Note, Advanced Hardware Architectures Inc, 2635 Hopkins Court, Pullman, WA, 1996.
- [AHA97a] 'AHA3521 40 Mbytes/s ALDC Data Compression Coprocessor IC', Product Brief, Advanced Hardware Architectures Inc, 2635 Hopkins Court, Pullman, WA, 1997.
- [AHA97b] 'AHA3211 20 Mbytes/s DCLZ Data Compression Coprocessor IC', Product Brief, Advanced Hardware Architectures Inc, 2635 Hopkins Court, Pullman, WA, 1997.
- [AlliedTelesyn00] 'Link Compression Facilities in the Network iQ Router', AlliedTelesyn Corporation, Technical Notes, 960 Stewart Drive, Sunnyvale, CA, 2000.
- [Altera98] 'JAM Programming & Test Language Specification Version 2.0', Altera Corporation, Altera corporation, 101 Innovation Drive, San Jose, CA 1998.
- [Altera01] 'APEX20K Programmable Logic Device Family', Data sheet, Altera corporation, 101 Innovation Drive, San Jose, CA, 2001.
- [Arnold97] R. Arnold, T.Bell, 'A Corpus for the Evaluation of Lossless Compression Algorithms', Data Compression Conference, pp. 201-210, 1997.
- [Arps88] R. Arps, T.Truong, D. Lu, R. Pasco, T. Freidman, 'A Multi-Purpose VLSI Chip for Adaptive Data Compression of Bilevel Images', IBM Journal of Research and Development, Vol.32, No. 6, pp-775-795, 1988.

- [Bell89] T. C. Bell, I. H. Witten, J. G. Cleary, 'Modelling for Text Compression', ACM Computing Surveys, Vol. 21, No. 4, pp. 557-591, 1989.
- [Bell90] T. C. Bell, J. G. Cleary and I. H. Witten, 'Text Compression', Prentice-Hall, NJ, 1990.
- [Bentley86] J. L. Bentley, D. D. Sleator, R. E. Tarjan, V. K. Wei, 'A Locally Adaptive Data Compression Scheme', Communications of the ACM, Vol. 29, No. 4, pp. 320-330, 1986.
- [Betz98] V. Betz, J. Rose, 'How Much Logic Should Go in an FPGA Logic Block?', IEEE Design & Test of Computers, pp. 10-15, January-March, 1998.
- [Bianchi89] M. Bianchi, J. Katto, D. Van Maren, 'Data Compression in a Half-inch Reel-to-reel Tape Drive', Hewlett-Packard Journal, Vol. 40, No. 6, pp. 26-31, 1989.
- [Bloom98] C. Bloom, 'Solving the Problems of Context Modelling', <http://www.cbloom.com/papers/index.html>, 1998.
- [Boo98] M. Boo, J.D. Bruguera and T. Lang, 'A VLSI Architecture for Arithmetic Coding of Multilevel Images', IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing, Vol. 45, No. 1, pp. 163-168, January 1998.
- [Bunton92] S. Bunton, G. Borriello, 'Practical Dictionary Management for Hardware Data Compression', Communications of the ACM, Vol. 35, No. 1, pp. 95-104, 1992.
- [Burton95] R. C. Burton, 'Fiber Channel', www.cis.ohio-state.edu/~jain/cis788-95/fiber_channel/index.html, 1995.
- [Caruso99a] J. Caruso, 'Gigabit Ethernet Ventures Into the Land Beyond LAN', Network World, 118 Turpike Rd, Southborough, MA, October, 1999.
- [Caruso99b] J. Caruso, '10G Ethernet WANs', Network World, 118 Turpike Rd, Southborough, MA, August, 1999.
- [Cheng95] J.M.Cheng and L.M.Duyanovich, 'Fast and Highly Reliable IBMLZ1 Compression Chip and Algorithm for Storage', Hot Chips VII Symposium, August 14-15, pp. 155-165, 1995

- [Chen98] Jin-Ming Chen, Che-Ho Wei, 'A Novel VLSI Design for Ziv-Lempel Data Compression', The 1998 IEEE Asia-Pacific Conference on Circuits and Systems, pp. 739-742, 1998.
- [Cisco00] 'Data Compression AIM for the Cisco 2600 Series', Cisco Product Catalogue, Cisco Systems Inc, 170 West Tasman Drive, San Jose, CA, December 2000.
- [Cleary84] J. Cleary, I. Witten, 'Data Compression Using Adaptive Coding and Partial String Matching', IEEE Transactions on Communications, Vol. 32, No. 4, pp. 396-402, 1984.
- [Cleary95a] J. G. Cleary, W. J. Teahan, 'Experiments on the Zero Frequency Problem', Department of Computer Science, University of Waikato, 1995.
- [Cleary95b] J. G. Cleary, W. J. Teahan, I. H. Witten, 'Unbounded Length Contexts for PPM', Data Compression Conference, pp. 52-61, 1995
- [Cormack87] G. V. Cormack, 'Data Compression Using Dynamic Markov Modelling', The Computer Journal, Vol. 30, No. 6, pp. 541-549, 1987.
- [Craft98] D. J. Craft, 'A Fast Hardware Data Compression Algorithm and Some Algorithmic Extensions', IBM Journal of Research and Development, Vol. 42, No. 6, pp. 733-745, 1998.
- [Cressman94] D. C. Cressman, 'Analysis of Data Compression in the DLT2000 Tape Drive', Digital Technical Journal, Vol. 6, No. 2, 1994.
- [Cyclades00] 'Data Compression & Frame Relay', White Paper, Cyclades Corporation, 41829 Albrae Street, Fremont, CA, 2000.
- [DCP95] 'DCP816', Data sheet, DCP Research Corp, 8923 - 148 St, Edmonton, Alberta, 1995.
- [Dell99] 'RAID Technology', White Paper, Dell Computer Corporation, One Dell Way, Round Rock, Texas, March, 1999.
- [Dickson00] K. Dickson, "Cisco IOS Data Compression", White paper, Cisco Systems Inc, 170 West Tasman Drive, San Jose, CA, 2000.

- [Djumin97] S. Djumin, 'Gigabit Networking: High Speed Routing and Switching', www2.cis.ohio-state.edu/~jain/cis788-97/gigabit_nets/index.htm, 1997.
- [Drach95] N. Drach, A. Sez nec, D. Windheiser, 'Direct-Mapped Versus Set-Associative Pipelined Caches', Proceedings of PACT' 95 (Parallel Architectures and Compiler Techniques), Chypre, June 1995.
- [Elias75] P. Elias, 'Universal Codeword Sets and Representation of Integers', IEEE Transactions on Information Theory, Vol. 21, pp. 194-203, 1975.
- [Fano49] R. M. Fano, R. M., 'Transmission of Information'. Cambridge, MA: M.I.T. Press, 1949.
- [Fiala89] E. R. Fiala, D. H. Greene, 'Data Compression with Finite Windows', Communications of the ACM, Vol. 32, No. 4, pp. 490-505, 1989.
- [Franaszek96] P. Franaszek, P. Robinson, J. Thomas, ' Parallel Compression With Adaptive Dictionary Construction', Proceedings of the Data Compression Conference, pp. 200-209, 1996.
- [Gatefield99] 'Using the Gatefield JAM Player', Gatefield corporation, 47436 Fremont Blvd, Fremont, CA, 1999.
- [GEA97] 'Gigabit Ethernet Overview', White Paper, Gigabit Ethernet Alliance, http://www.gigabit-ethernet.org/technology/whitepapers/gige_97/papers97_toc.html, 1997.
- [Golomb66] S. W. Golomb, 'Run-Length Encodings', IEEE Transactions on Information Theory, IT-12, pp. 399-401, July 1966.
- [Gooch96] M. Gooch, 'High Performance Lossless Data Compression Hardware', PhD Thesis, Loughborough University, UK, 1996.
- [Guazzo80] M. Guazzo, 'A General Minimum-Redundancy Source Coding Algorithm', IEEE Transactions on Information Theory, IT-26, pp. 15-25, Jan 1980.

- [Halfill94] T. H. Halfill, 'How Safe Is Data Compression?', Byte Magazine, Vol. 19, No. 2, pp.56-74, 1994.
- [Hallnor00] E. G. Hallnor, S. K. Reinhardt, 'A Fully Associative Software-Managed Cache Design', Proceedings of the 27th International Symposium on Computer Architecture, pp. 107-116, 2000.
- [Hi/fn96] 'How LZS Data Compression Works', Application Note, Hi/fn Inc, 750 University Avenue, Los Gatos, CA, 1996.
- [Hi/fn97] 'Data Compression Analysis in Data Communications', Application Note, Hi/fn Inc, 750 University Avenue, Los Gatos, CA, 1997.
- [Hi/fn98a] 'The First Book of Compression Encryption', Hi/fn primers, 750 University Avenue, Los Gatos, CA, Hi/fn Inc, 1998.
- [Hi/fn98b] '9610 Data Compression Processor', Data Sheet, Hi/fn Inc, 750 University Avenue, Los Gatos, CA, 1998.
- [Hi/fn99] '9600 Data Compression Processor', Data Sheet, Hi/fn Inc, 750 University Avenue, Los Gatos, CA, 1999.
- [Howard92] P. G. Howard, J. S. Vitter, ' Analysis of Arithmetic Coding for Data Compression', Information Processing and Management, Vol. 28, No. 6, pp. 749-763, November 1992.
- [Howard93a] P.G. Howard, 'The Design and Analysis of Efficient Lossless Data Compression Systems', Technical report CS-93-28, Brown university, Providence, Rhode Island, 1993.
- [Howard93b] P. G. Howard, J. S. Vitter, ' Design and Analysis of Fast Text Compression Based on Quasy-arithmetic Coding', Proceedings of the IEEE Computer Society, Data Compression Conference, Snowbird, Utah, pp. 98-107, March 30-April 1 1993.
- [Huffman51] D. A. Huffman, 'A Method for the Construction of Minimum Redundancy Codes', Proceedings of IRE, Vol. 40, pp. 1098-1101, 1951.

- [Hsieh98] M. Hsieh, C. Wei, 'An adaptative Multialphabet Arithmetic Coding for Video Compression', *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 8, No. 2, pp 130-137, April 1998.
- [IBM94] 'ALDC1-40S-M', Data sheet, IBM Microelectronics Division, 15080 Route 52, Bldg 504 Hopewell Junction, NY, 1994.
- [Intel00] 'Intel Express 9500 Routers', Data Sheet, Intel corporation, 2200 Mission College Blvd, Santa Clara, CA, 2000.
- [Intel01] '56K Modem Technology: Faster Communication Over Standard Telephone Lines', Intel Networking White Papers, Intel Corporation, 2200 Mission College Blvd, Santa Clara, CA, 2001.
- [JAM98] 'JAM Programming & Test Language Specification Version 2.0', Altera corporation, 101 Innovation Drive, San Jose, CA, 1998.
- [Jiang94] J. Jiang, S. Jones. 'Parallel Design of Arithmetic Coding', *Proceedings IEE, Part E*, Vol 141, pp 327-333, November 1994.
- [Jiang95] J. Jiang, 'Novel design of Arithmetic Coding for Data Compression', *IEE Proc.-Comput. Digit. Tech.*, Vol. 142, No. 6, pp 419-424, November 1995.
- [Jiang96a] J. Jiang, 'A Novel Parallel Design of a Codec for Black and White Image Compression', *Signal Processing : Image Communication*(8), No. 5, pp. 465-474, 1996.
- [Jiang96b] J. Jiang, 'Design of Neural Networks for Lossless Data Compression', *Optical Engineering*, Vol. 35, No. 7, pp. 1837-1843, 1996.
- [Jones92] S.Jones, '100Mbit/s Adaptive Data Compressor Design Using Selectively Shiftable Content-Addressable Memory', *Proceedings IEE (Part G)*, vol.139, no.4, pp.498-502, 1992.
- [Jones00] S. Jones, 'Partial-matching Lossless Data Compression Hardware', *IEE Proc.-Comput. Digit. Tech.*, Vol. 147, No. 5, pp.329-334, 2000.

- [Jou90] N. P. Jouppi, 'Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers', Proceedings 17th Symposium on Computer Architecture, 1990.
- [Jou99] J. M. Jou, P. Y. Chen, 'A Fast and Efficient Lossless Data Compression Method', IEEE Transactions on Communications, Vol. 47, No. 9, pp. 1278-1283, September 1999.
- [Jung98] B. Jung, W. P. Burlleson, 'Performance Optimization of Wireless Local Area Networks Through VLSI Data Compression', Wireless Networks, Vol. 4, pp. 27-39, 1998.
- [Kampf98] F. A. Kampf, 'Performance as a Function of Compression', IBM Journal of Research and Development, Vol. 42, No. 6, pp. 759-766, 1998.
- [Wilson97] K. M. Wilson, K. Olukotun, 'Designing High Bandwidth On-Chip Caches', Proceedings of the 24th international symposium on Computer architecture, Denver, CO, pp. 121-132, 1997.
- [Kjelso95] M. Kjelso, M. Gooch, U. Simm, S. Jones, 'Hardware Data Compression and Memory Management for Flash-Memory Disks', Proceedings ISIC-95, 6th International Symposium on IC Technology, Systems and Applications, IEEE Press, pp 161-165, 1995.
- [Kjelso96] M.Kjelso, M.Gooch, S.Jones, 'Design & Performance of a Main Memory Hardware Data Compressor', Proceedings 22nd EuroMicro Conference, pp. 423-430, September 1996, Prague, Czech Republic.
- [Knuth85] D. E. Knuth, 'Dynamic Huffman Coding', J.Algorithms, Vol. 6, pp. 163-180, June 1985.
- [Kuang98] S. Kuang, J. Jou, Y. Chen, 'The Design of an Adaptive On-Line Binary Arithmetic- Coding Chip', IEEE Transactions on Circuits and Systems-I: Fundamental Theory and Applications, Vol. 45, No. 7, pp 693-706, July 1998.
- [Langdon84] G. G. Langdon, 'An Introduction to Arithmetic Coding', IBM Journal of Research and Development, Vol. 28, No. 2, pp. 135-149, 1984.

- [Lee96] Horn-Yeon Lee et al, 'A Parallel Architecture for Arithmetic Coding and its VLSI Implementation', IEEE 39th Midwest Symposium on Circuits and Systems, Vol. 3, pp. 1309-1312, 1996.
- [Lei95] S. M. Lei, 'Efficient Multiplication-Free Arithmetic Codes', IEEE Transactions on Communications', Vol. 43, No. 12, pp. 2950-2958, 1995.
- [Lelewer87] D. A. Lelewer, D. S. Hirschberg, 'Data Compression', ACM computing surveys, Vol. 19, No. 3, pp. 261-297, 1987.
- [Liu95] Y. Liu et al, 'Design and Hardware Architectures for Dynamic Huffman Coding', IEE Proc.-Comput.Digit.Tech., Vol. 142, No. 6, pp 411-418, November 1995.
- [Mace98] S. Mace, 'Faster SCSI and Fibre Channel SANs set the stage for servers that run and run', Byte magazine, January 1998.
- [Marks98] K. M. Marks, 'A JBIG-ABIC Compression Engine for Digital Document Processing', IBM Journal of Research and Development, Vol. 42, No. 6, pp. 753-758, 1998.
- [Mitel00] 'Data compression', white paper, Mitel Remote Access Solutions, Mitel Networks, 350 Legget Drive, Kanata, Ontario, 2000.
- [Moffat89] A. Moffat, 'Word-based Text Compression', Software-Practice and Experience, Vol.19, No. 2, pp. 185-198, 1989.
- [Moffat90] A. Moffat, 'Implementing the PPM Data Compression Scheme', IEEE Transactions on Communications, Vol. 38, No. 11, pp. 1917-1921, 1990.
- [Moffat94] A.Moffat, N.Sharman, I.Witten, T.Bell, 'An Empirical Evaluation of Coding Methods for Multi-symbol Alphabets', Information Processing & Management, Vol. 30, No 6, pp. 791-804, 1994.
- [Moffat97] A. Moffat, 'Critique of the paper 'Novel Design of Arithmetic Coding for Data Compression'', IEE Proc-Comput. Digit. Tech, Vol 144, No. 6, pp 394-396, 1997.

[Mogul97] J. C. Mogul, F. Douglis, 'Potential Benefits of Delta Encoding and Data Compression for HTTP', In Proceedings of the ACM SIGCOMM '97 Symposium, Cannes, France, 1997.

[Mukherjee93] A. Mukherjee, N. Ranganathan, J. Flieder, T. Acharya, 'MARVLE: A VLSI Chip for Data Compression Using Tree-Based Codes', IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 1, No. 2, pp 203-213, June 1993.

[MPEG-2] MPEG-2 video, Draft Int. Standard ISO/IEC DIS 13818-2.

[Nelson91] M. Nelson, 'The Data Compression Book', Prentice Hall, 1991.

[Nelson96] M. Nelson, 'Data Compression with the Burrows-Wheeler Transform', Dr. Dobb's Journal, September 1996.

[Nunez99] J.L. Núñez, C. Feregrino, S. Bateman, S. Jones, 'The X-MatchLITE FPGA-Based Data Compressor', Proceedings of the 25th EUROMICRO Conference, Digital Systems Design: Architectures, Methods and Tools, pp. 126-132, September, 1999.

[Nusinov94] E. Nusinov, J. Pasco-Anderson, 'High Performance Multi-channel Data Compression Chip', IEEE Custom Integrated Circuits Conference, pp. 203-206, 1994.

[Pennebaker88] W.B. Pennebaker et al, 'An overview of the Basic Principles of the Q-coder Adaptive Binary Arithmetic Coder' IBM J. Res. Develop, Vol 32, No. 6, pp 717-725, November 1988.

[Peon97] M. Peon, R.R Osorio, J. D. Bruguera, 'A VLSI Implementation of an Arithmetic Coder for Image Compression', Proceedings of the 23rd EUROMICRO Conference, pp. 591-598, 1997.

[Pivotal97] 'LAN & WAN Technologies Overview', White Paper, Pivotal Corporation, 300-224 West Esplanade, Vancouver, BC, 1997.

[Printz93] H. Printz, P. Stublely, 'Multialphabet Arithmetic Coding at 16 Mbytes/sec', Proceedings of the 3rd Data Compression Conference, Snowbird Utah, pp. 128-137, 1993.

[Quantum99] 'Quantum DLT 8000', Data Sheet, Quantum Corporation, 500 McCarthy Blvd, Milpitas, 1999.

[Rice83] R. F. Rice, 'Some Practical Universal Noiseless Coding Techniques', Jet Propulsion Laboratory, JPL, publication 83-17, Pasadena, California, 1983.

[Rice91] P. S. Yeh, R. F. Rice, W. Miller, 'On the Optimality of Code Options for a Universal Noiseless Coder', Jet Propulsion Laboratory, JPL, Publication 91-2, Pasadena, California, Feb 1991.

[Rissanen89] J. J. Rissanen, K. M. Mohiuddin, 'A Multiplication-Free Multialphabet Arithmetic Coder', IEEE Transactions on Communications, Vol. 37, pp. 93-98, February 1989.

[Sakanashi98] H. Sakanashi et al, 'Evolvable Hardware Chip for High Precision Printer Image Compression', Lectures Notes in Computer Science, Vol. 1478, pp 106-114, 1998.

[McFarling91] S. McFarling, 'Cache Replacement with Dynamic Exclusion', Technical Note TN-22, Western Research Laboratory, 250 University Avenue, Palo Alto, California, 94301 USA.

[Seagate97] 'Seagate Technologies 2000,4000, 8000 Series DAT Drives', Product Description Manual, Seagate Technology, 920 Disc Drive, Scotts Valley, CA, 1997.

[Shannon48] C. E. Shannon, 'A Mathematical Theory of Communication', Bell. Sys. Tech. J, Vol. 27, pp. 398-403, July 1948.

[Slattery98a] M. J. Slattery, J. L. Mitchell, 'The Qx-coder', IBM Journal of Research and Development, Vol. 42, No. 6, pp. 767-784, 1998.

[Slattery98b] M. J. Slattery, F. A. Kampf, 'Design Considerations for the ALDC Cores', IBM Journal of Research and Development, Vol. 42, No. 6, pp. 747-752, 1998.

[Storer82] J. A. Storer, T. G. Szymanski, 'Data Compression via Textual Substitution', Journal of ACM, Vol. 29, No. 4, pp.928-951, October 1982.

[Storage00] 'FibreRAID 2000 200 MB/sec Highest Performance RAID', Data Sheet, Storage Concepts Inc, 14352 Chambers Road, Tustin, CA, 2000.

- [Surk97] Y. Surk, T. Young, K. Park, 'A Novel PE-Based Architecture for Lossless LZ Compression', *IEICE Trans. Fundamentals*, Vol. E80-A, No. 1, pp. 233-237, January 1997.
- [Tanenbaum90] A. S. Tanenbaum, 'Structure Computer Organization', Prentice Hall International Editions, pp.209-215, 1990.
- [Tanenbaum96] A. S. Tanenbaum, 'Computer Networks', Third edition, Prentice-Hall Inc, 1996.
- [Thomborson92] C. Thomborson, "The V.42bis standard for data-compression modems", *IEEE Micro*, pp. 41-53, October, 1992.
- [Trillium97] 'Comparison of IP-over-SONET and IP-over-ATM technologies', White Paper, Trillium Digital Systems Inc, 12100 Wilshire Blvd, Los Angeles, CA, November, 1997.
- [Vandalore95] B. Vandalore, 'Gigabit Networking Survey', www.cis.ohio-state.edu/~jain/cis788-95/gigabit/index.html, 1995.
- [VanDuine00] R. VanDuine, 'Integrated Storage' Technical Paper, IBM Corporation, 3605 North Highway 52, Rochester, MN, 2000.
- [Vitter87] J. Vitter, 'Design and Analysis of Dinamic Huffman Codes', *Journal of the Association for Computing Machinery*, Vol.34, No. 4, pp. 825-845, 1987.
- [Wallace91] G. K. Wallace, "The JPEG still picture compression standard", *Communications of the ACM*, Vol. 34, No. 4, pp. 30-44, April, 1991.
- [Welch84] T. A. Welch, 'A Technique for High Performance Data Compression', *IEEE Computer*, Vol 17, No.6, pp. 8-19, June, 1984.
- [Witten87] I. H. Witten et al, 'Arithmetic Coding for Data Compression', *Communications of the ACM*, Vol. 30, No. 6, pp. 520-540, 1987.
- [Witten91] I. H. Witten, 'The Zero-Frequency Problem', *IEEE Transactions on Information Theory*, Vol. 37, No. 4, pp. 1085-1094, 1991.

[Wolf91] M. E. Wolf, M. S. Lam, 'A data locality optimizing algorithm', Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, pp. 30-44, 1991.

[Xilinx01] 'VIRTEX™-E 1.8 v Field Programmable Gate Arrays', Data sheet, Xilinx Inc, 2100 Logic Drive, San Jose, CA, February 2001.

[Xiong97] J. Xiong, W. Zhang, J. Cao, 'A new type of Dynamic Data Compression Chip', International Symposium on Test, and Measurement Conference Proceedings, ch 184, pp 243-245, 1997

[Ziv77] J. Ziv, A. Lempel, 'A Universal Algorithm for Sequential Data Compression' IEEE Trans. Inf. Theory, Vol. IT-23, pp. 337-343, 1977

[Ziv78] J. Ziv, A. Lempel, 'Compression of Individual Sequences Via Variable Rate Coding', IEEE Transactions on Information Theory, IT-24, pp. 530-536, 1978.

Publications

José Luis Núñez, Claudia Feregrino, Stephen Bateman, Simon Jones, 'The X-MatchLITE FPGA-Based Data Compressor', Proceedings of the 25th EUROMICRO Conference, Digital Systems Design: Architectures, Methods and Tools, pp. 126-132, September, 1999.

José Luis Núñez, Simon Jones, 'The X-MatchPRO 100 Mbytes/second FPGA-Based Lossless Data Compressor', Proceedings of Design, Automation and Test in Europe, DATE Conference 2000, pp.139-142, March, 2000.

José Luis Núñez, Simon Jones, Stephen Bateman, 'X-MatchPRO: A high performance full-duplex lossless data compressor on a ProASIC FPGA', to appeared in Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications IDAACS'2001.

Riad Stefo, **José Luis Núñez**, Claudia Feregrino, Sudipta Mahapatra, Simon Jones, 'FPGA-based modelling unit for high speed lossless arithmetic coding', to appeared in 11th International Conference on Field Programmable Logic and Applications FPL'2001.

Patent Applications

Simon Jones, **José Luis Núñez**, 'Data Compression Having Improved Compression Speed', UK Application No. GB0001711.1, Jan. 25, 2000.

Simon Jones, **José Luis Núñez**, 'Data Compression Having More Effective Compression', UK Application No. GB0001707.9, Jan. 25, 2000.

