The X-MatchLITE FPGA-Based Data Compressor.

José Luis Núñez, Claudia Feregrino, Stephen Bateman*, Simon Jones
Electronic Systems Design Group, Loughborough University
Loughborough, Leicestershire. LE11 3TU. England.
*GateField Corporation, 47100 Bayside Parkway, Fremont, CA.

Abstract

This paper introduces a hardware amenable algorithm for lossless data compression and a highly integrable architecture which enables Gbit/s compression using contemporary ASIC technology. An FPGA prototype of the architecture is presented. A comparison between this prototype and the full version of the system is made together with the details of the engineering decisions needed to successfully realize an ASIC compressor in FPGA technology.

1. Introduction

While there has been substantial interest in lossy data compression hardware for image and signal processing, there has been until quite recently, relatively less interest in lossless compression hardware where exact data restoration occurs. However, a combination of increased pressure on bandwidth and cost per bit in storage and data transmission, together with the requirements to improve power consumption by reducing data volume has resulted in an increased interest in lossless compression hardware. As part of this process we have been researching high performance lossless data compression hardware for a number of years [1], [2], [3]. A major outcome of this work has been the X-match approach [4] which utilizes an adaptive Content Adressable Memory (CAM) for the dictionary storage together with dynamic coding of the dictionary locations and associated bit packing . Such an approach appears to offer the possibility of Gbit/s operation with compression equivalent to the UNIX utility Compress [5].

Extensive algorithm and VHDL design has resulted in an X-Match soft core with a 110 Kgates complexity and operating at 25 MHz. . For many applications such high speed is not necessary. Furthermore, the complexity of the full X_Match system exceeds the achievable with contemporary FPGA technology thus missing out other benefits included with areprogramable system. As a consequence of this we have developed a simplified and redesigned version of X-Match which is suitable for FPGA implementation with only modest compression performance degradation.

The remainder of this paper is organized as follows: Section 2 describes the original X-Match algorithm while section 3 depicts its architecture. Section 4 deals with the details of the FPGA architecture. Section 5 compares the performance obtained by both designs. Section 6 presents our FPGA verification methodology. Finally section 7 concludes.

2. The X-Match algorithm

2.1. Algorithm description

The X-Match algorithm uses a dictionary of previously seen data and attempts to match the current data element with an entry in the dictionary. Each entry is 4 bytes wide and 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 'dont 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. Data expansion is then limited to 3.125% (32 bits -> 33 bits). The dictionary is maintained using a move to front (MTF) strategy [6] whereby a new tuple is place at the front of the dictionary while the rest move down one position. This strategy generates a LRU (Least Recently Used) replacement policy so the last dictionary_size tuples are used as a window of history information for the compression process. 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 three separate fields as follows:

- The match location. It uses Phased Binary Code (PBC) [7], a version of Huffman coding chosen for its suitability for hardware implementation. PBC is characterized by assigning smaller codes while the dictionary grows from 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 [8] based on the statistics obtained through extensive simulation.
- Any extra characters that did not match transmitted in literal form.

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 thetuples 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 much 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 two entries contain the same data. Several partial matches are possible simultaneously so the one that produces a shorter output is selected as valid. The algorithm is given as pseudo-code in Figure 1.

2.2. Algorithm performance

The compression ratio achieved by the X-Match algorithm is examined in this section. We define compression ratio as the ratio of output bits to input bits. All the results are obtained compressing 4Kbyte blocks. Details on the data set used for the experiments can be found in [4]. For comparison, we use three well-known algorithms as detailed below:

- Arithmetic- zero-order adaptive arithmetic coder operating on a byte stream [9].
- Compress- the popular 'Compress' program available under UNIX.
- LZS- proprietary implementation of the Lempel-Ziv algorithm used in the STAC electronics data compressor chip [10] - a valid representative of current high performance compression hardware.

```
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 1. The X-Match algorithm

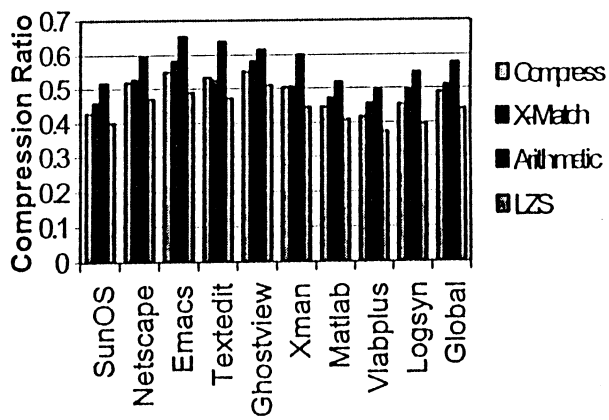

Figure 2. Algorithm performance

It is clear that the arithmetic coder is the worst performer while X-Match gives a reasonable compression ratio. LZS consistently outperforms X-Match but being a byte-oriented algorithm offers a poorer throughput, as we will see in section 5.

3. The X-match hardware

A fuller description of the X-match hardware can be found in [4] so here we will briefly describe its main characteristics. The architecture is based around a block of CAM to realize the dictionary [1]. This is necessary since the search operation must be done in parallel in all the entries in the dictionary to allow high throughput. The size of the CAM is 128 words times 4 bytes/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 vector produced by the dictionary maintenance functions.

3.1. Compressor architecture

An overview of the compressor architecture is presented in Figure 3. 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 PBC that depends on how many entries are valid in the dictionary as indicated by the Next-Free-Location counter (NFL) 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.

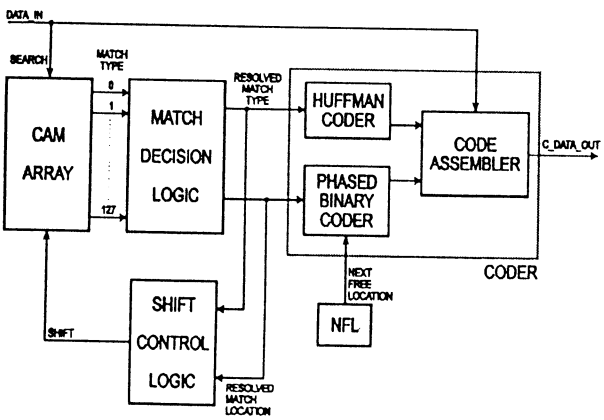

Figure 3. Architecture of the compressor

3.2. Decompressor architecture

Figure 4 shows the decompressor architecture. The compressed data enters the decoder to produce a match

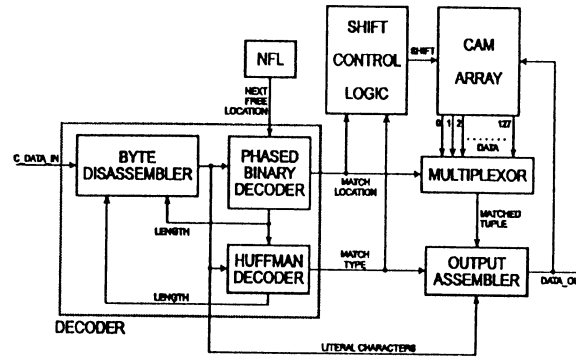

Figure 4. Architecture of the decompressor

location and a match type. The byte disassembler is used to shift the correct number of bits of the 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 decompressed tuple is also required to maintain the dictionary following the same pattern used during compression.

4. The X-match FPGA prototype

We now present the engineering decisions taken to successfully realize the ASIC compressor X-Match in FPGA technology.

4.1. Introduction to the FPGA prototype

Two main factors are responsible for the modifications introduced to the initial version:

- The use of a GF250F100 ProASIC™ device manufactured by GateField Corporation [11] that offers around 46K usable gates out of 100K physically present gates presents a hard limit in terms of the amount of logic available. This device is manufactured in a 0.6 μm FLASH-CMOS technology and has been selected for two reasons: Firstly, it uses the concept of flash-based programmable elements so the downloaded design stays in the chip when the power is off. Secondly, its ASIC style architecture facilitates the migration to high volume production technologies such as gate

arrays. This fine-granularity device groups its gates into small tiles each of them ready to realize any logic function of a maximum three inputs and one output. The GF250F100 contains 12800 of these tiles. Depending on the logic function the tile implements the utilization ratio ranges from 1 gates/tile to 8 gates/tile.

- The initial gate count of the gate array based X-Match (110 Kgates) is out of range for a single-chip implementation. Although it is possible to use a partitioner to split the design in several ProASIC devices timing performance degradation makes this option unfeasible and a cut-down single-chip version is unavoidable.

4.2. Architectural simplification

The first step needed to fit the design into the ProASIC device is to reduce the size of the CAM array. The original size of the CAM is 128x4 bytes. Table 1 shows a summary of a study done on what would be the best way of reducing it. Since a minimum of 2 bytes has to match, reducing the width below 3 bytes would not respect the partial match concept.

Table 1 presents the compression ratio using different CAM simplification alternatives. It is clear that better compression is obtained when decreasing the dictionary length leaving the width at 4 bytes. Our initial tests show that the design has an area utilization ratio comparable or slightly worse than the figure reported by GateField and that makes the 64x4 dictionary unfeasible. We select a 32x4 dictionary size that brings about further simplification in the coding technique for the match locations. The simplification consists of replacing PBC by Uniform Binary Coding (UBC). UBC allocates a fixed size code to each location using log(max_size) bits where max_size is the maximum number of entries in the CAM. The original design uses PBC which is effective when the dictionary is growing but once it is full there is no advantage. Table 1 shows that the compression ratio improvement is inferior to 0.4% when comparing PBC with UBC in the 32x4 implementation. The reason is that the smaller the dictionary, the faster it fills up losing the advantage that PBC provides during the growing stage. Therefore we decided to use UBC, reducing the amount of logic needed in the coding and decoding modules and to preload the now non-growing dictionary with common data. The device size in Table 1 is an estimation of the UBC implementation.

4.3. Architectural improvement

One of the major concerns when moving from gate arrays to FPGAs is the resultant slower speed. Therefore special attention has been paid to the critical paths trying to compensate for the performance migration cost to this technology. In both cases the overall 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. 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 operation is only local to some positions when a full match occurs and a full match can only happen in one position at a time since no two positions have exactly the same data. Therefore we do not need to resolve the best match to know how to shift the dictionary. If there is no full match the shift affects all the locations and if there is a full match this is known before accessing the best match decision logic.

Dictionary Structure	Location Coding Technique (CR)		CAM size (tiles)	Device Size (tiles)	FPGA utilization
	PBC	UBC			
128x4 (original)	0.511	0.520	25.6 K	30.8 K	240%
128x3	0.557	0.570	19.2 K	24.4 K	190%
64x4	0.527	0.533	12.8 K	18 K	140%
64x3	0.576	0.582	9.6 K	14.8 K	115%
32x4	0.544	0.546	6.4 K	11.6 K	90%
32x3	0.596	0.600	4.8 K	10 K	78%

Table 1 . CAM simplification design choices.

This change together with moving the search multiplexer out of the way and embedding the movement generation logic into the priority logic leaves the architecture as shown in Figure 5. The new critical path is around 25% faster and a throughput of over 140 Mbytes/second should be expected in the ASIC technology.

frequency is limited to 6 MHz that implies a throughput of 24 Mbytes/second. This is considerably worse than the 25 MHz and 100 Mbytes/second of the original gate array pre-layout design. However a speed degradation factor of 3 is to be expected [12] and we should be able to improve this figure with the arrival of higher gate count devices so we can eliminate some of the constrains just mentioned.

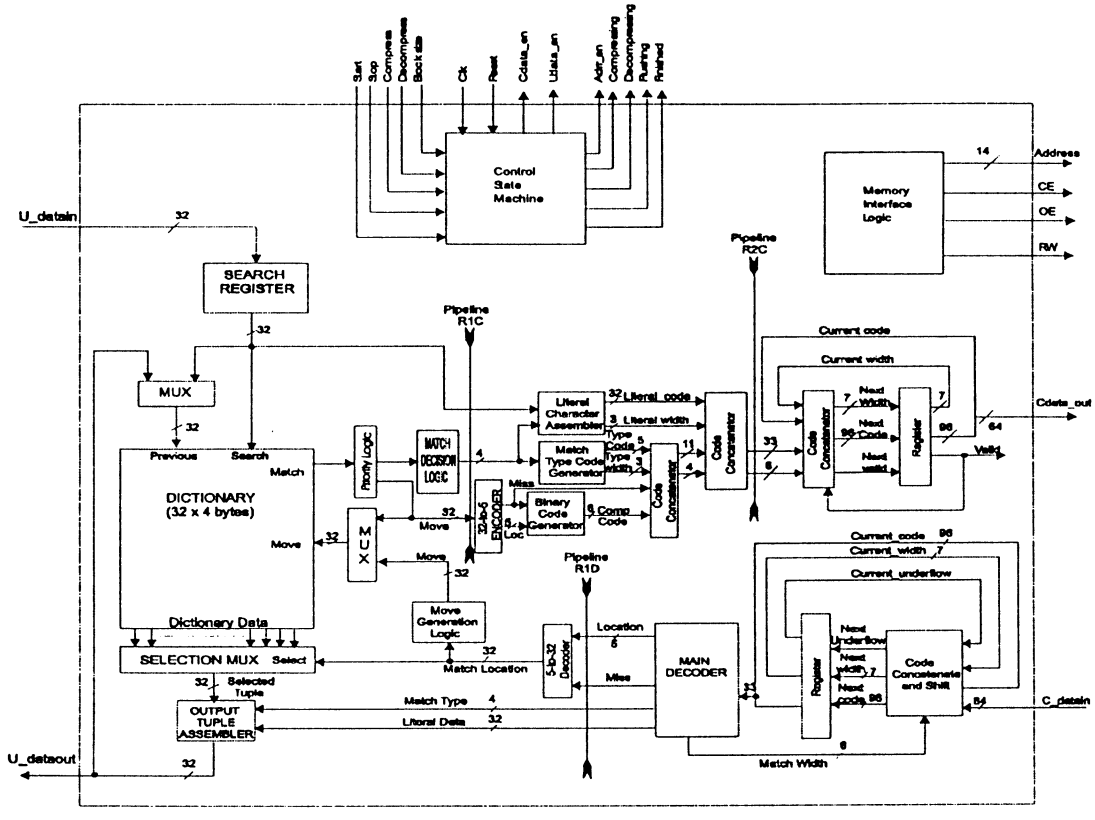

Figure 5. The ProASIC X-MatchLITE architecture

5. Hardware summary and performance comparison

The simplified design (32x4 and UBC) has been synthesized, placed and routed into the ProASIC device. The utilization ratio is over 90% with around 38Kgates (11640 tiles). Routing resources limitations imply that a trade-off has to be made in terms of the amount of buffering allowed in the high-load nets and pipelining. No buffering allows us to include an extra level of pipelining but the time required to drive the high-fanout nets becomes dominant preventing any benefit. Adding a buffering scheme forces us to eliminate the extra level of pipelining to be able to route the device. The clock

Table 2 summaries the characteristics of X-Match versus X-MatchLITE. The STAC Electronics device used for comparison in section 2.2 achieves 50 Mbytes/second using a 50 MHz clock and a 0.5 μm CMOS gate array technology. Although it clocks at a higher frequency the throughput is lower than X-Match because it operates at a byte level while both X-Match and X-MatchLITE operate at a tuple level.

The global compression ratio obtained by X-MatchLITE is around 5% worse than X-Match using the same data set already shown in figure 2

6. Hardware verification

In order to be able to verify the correct functionality of our prototype we built a test board so we could apply the same stimulus files to the software-based X-Match and to the ProASIC device. The interface to the test board is done through a virtual test-bench written using the Labview software and running on a conventional PC. The virtual test-bench interfaces to a 96-bit parallel digital I/O DAQ card that it is hardwired to the test-board. The same identical vector file is applied to each stage of the design flow allowing us to compare the output from four different sources, namely: the original algorithm description written in C, the RTL description of the device written in VHDL, the back-annotated netlist and finally the real silicon.

A simple text file comprising 16 tuples (64 bytes) forms the test vector file for compression. Although limited by the number of pins available in the board and the difficulties associated with routing the signals under verification to those fixed pins the tests show no differences among the output of each stage of the design flow. To verify the decompression mode it is necessary to execute again the place and routing process so the signals associated to this operational mode can be assigned to the pins in the board physically connected to the DAQ card. Then, the output of the compression mode is used as the test vector file to correctly regenerate the test vector file used for compression.

7. Conclusions

We have shown how a complex design can be practically implemented into an FPGA. The use of a fine granularity device where each block defines a very simple logic function has proven to be well suited to an unbalanced design like the X-Match compressor where most of the sequential logic (dictionary) and combinatorial logic (coding and decoding functions) are clearly separated on the silicon. Other FPGA architectures where the building blocks implement mixed combinatorial and sequential functions offer poorer utilization ratios. Only routing resource limitations have prevented us from incorporating the functionality required by a more powerful algorithm.

The FPGA prototype while still offering good compression ratio and speed shows that a full implementation of X-Match would be a very useful data compressor. We expect to be able to tackle these issues when higher density chips become available. The device is currently available to be incorporated into a functional system.

Acknowledgements
The authors acknowledge with gratitude the funding provided by the UK's EPSRC under grant number GR/L 54530 and also the support provided by GateField Corporation and the Consejo Nacional de Ciencia y Tecnologia (CONACyT, Mexico).

		X-Match	X-MatchLITE
Implementation Details	Granularity	4 bytes	4 bytes
	Dictionary size (locations x bytes/location)	128 x 4	32 x 4
	Partial Match Coding Technique.	Huffman Coding	Huffman Coding
	Location Coding Technique	Phased Binary Coding	Uniform Binary Coding
Process Details	Feature Size	0.6 µm CMOS	0.6 µm FLASH-CMOS
	Technology	Gate Array	FPGA
Gate Count		110 Kgates	38 Kgates
Maximum Clock Frequency		25 MHZ	6 MHZ
Throughput		100 Mbytes/s	24 Mbytes/s

Table 2. Comparison of X-Match with X-MatchLITE

References

[1] S.Jones, '100Mbit/s Adaptative Data Compressor Design Using Selectively Shiftable Content-Addressable Memory', Proceedings of IEE (part G), vol.139, no.4, pp.498-502, 1992.

[2] M. Kjelsø, 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.

[3] J. Jiang, S.Jones, 'Parallel Design of Arithmetic Coding', Proceedings IEE, PartE, Vol 141, pp 327-333, November 1994.

[4] 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.

[5] S. W. Thomas, J. McKie, S.Davies, K. Turkowski, J. A. Woods, and J. W. Orost, Compress program and documentation available from wuarchive.wustl.edu/ /packages/compression/compress-4.1.tar.

[6] J.L.Bentley at all, 'A Locally Adaptative Data Compression Scheme', Communications of the ACM, vol.29, no.4, pp.320-330, 1986.

[7] T.Bell, J.Cleary, I. Witten, 'Text Compression', (Section A.2), Published by Prentice Hall, 1990.

[8] D.Huffman, 'A method for the construction of Minimun Redundancy Codes', Proceedings of the I.R.E, pp.1098-1101, 1958.

[9] I. H. Witten, R. M. Neal, and J. G. Cleary, 'Arithmetic Coding for Data Compression', Communications of the ACM, Vol. 30, pp. 520-540, June 1987.

[10] Information available from http://www.hifn.com

[11] GateField Corporation., Fremont, CA. The GF250F ProASIC Products DATAbook, 1997

[12] 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.