Halvor Horvei

A Hardware Architecture for the Encoding and Decoding of the LZSS Compression Algorithm

Masteroppgave i Elektronisk systemdesign og innovasjon
Veileder: Bjørn B. Larsen, Halfdan Bechmann
Juli 2020

Halvor Horvei

A Hardware Architecture for the Encoding and Decoding of the LZSS Compression Algorithm

Masteroppgave i Elektronisk systemdesign og innovasjon
Veileder: Bjørn B. Larsen, Halfdan Bechmann
Juli 2020

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

NTNU
Kunnskap for en bedre verden

Abstract

This thesis aims to compress firmware to reduce the amount of necessary flash memory. In order to do this an architecture for encoding and decoding of the LZSS compression algorithm is presented. The encoding module is based on the use of an application-specific CAM design. The CAM allows for fast searching and matching of data by parallel access. By utilizing masking registers to remove redundant comparisons, the design can lower its power dissipation. The decoding process has been pipelined in order to minimize the time it takes for decompression.

The design has been evaluated based on area and compression time for different buffer sizes. Power consumption has also been qualitatively discussed. The results are based on benchmarks from the Calgary Corpus, which is a widely test set for data compression. The design has been tested and verified using Systemverilogs object-oriented testing and assertion-based verification.

Sammendrag

Denne oppgaven utforsker muligheten for å komprimere fastvare for å redusere den nødvendige størrelsen av ikke-volatilt minne. For å gjøre dette presenteres arkitektur for koding og dekoding av LZSS-komprimeringsalgoritmen. Modulen for koding av data er basert på bruken av en applikasjonsspesifikk variant av CAM. CAM er en minne-enhet som tillater rask søk og sammenligning av data ved gjennom parallell aksessering. Ved å bruke maskeringsregistere kan unødvendige sammenligninger av data reduseres, som igjen reduseres effektbruken til designet. Dekodingsprosessen bruker en ekstra buffer for å redusere tiden det tar å dekomprimere data. Designet har blitt evaluert basert på størrelse av designet og komprimeringstiden for ulike buffer-størrelser. Effektbruk har også blitt kvalitativt diskutert. Resultatene er basert på test-data fra Calgary Corpus. Designet har blitt testet og verifisert ved SystemVerilogs test- og verifikasjonsmetoder.

Preface

I would like to thank my academic supervisor Bjørn B. Larsen for guidance throughout the process of writing the thesis. I would also like to thank Halfdan Bechmann and Silicon Labs for the guidance and giving me the opportunity for this thesis. Cliff Cummings from Sunburst Design and Dave Rich from Mentor Graphics for their help and general contribution to the online Verilog/SystemVerilog community. Lastly, I want to thank my family and friends for their support and for giving me motivation throughout the semester.

Contents

List of Figures

List of Tables

Acronyms

CAM	Content-addressable Memory. 3, 7, 13, 14, 20, 30, 31
CPU	Central Processing Unit. 11
DUT	Design Under Test. 19, 25
EEPROM	Electrically Ereasable Programmable Read-only Memory. 6
FF	Flip-flop. 15, 20, 23
FIFO	First-In-First-Out. 20
HDL	Hardware Descriptive Language. 16
LSB	Least Significant Byte. 21
LUT	Lookup Table. 15
MSB	Most Significant Byte. 21
RAM	Random-access Memory. 13
SRAM	Static Random-access Memory. 13
SVA	SystemVerilog Assertion. 16

1 Introduction

The newest emergence of this computing era is the desire to connect all sorts of devices to the internet, known as the Internet-of-Things (IoT). With an ever-growing market, analysts from Ericsson estimate that the number of IoT devices with cellular connection has grown to 400 million in 2016, with projections up to 1.5 billion by 2022 [3].

IoT-systems range among home automation, transportation and manufacturing. The development of IoT-systems poses several challenges, like the increasing heterogeneity of devices, privacy and scalability[4].

Another challenge IoT-devices encounter is security and the ever-facing threat of being attacked by malicious software. These devices are especially vulnerable due to its nature of wireless communication and minuscule design. The IoT end nodes can e.g. be providing critical information about the operational characteristics and position of a car. Medical devices collect sensitive information about patients health care. Personal information about the users daily routines can be extracted from smart home installations. All examples where it is vital that there is no leakage of information to any unwanted third party.

Frequent firmware updates are a necessity to handle discovered exploits and vulnerabilities. The firmware is usually kept in Electrically Ereasable Programmable Read-only Memory (EEPROM) or flash memory. This requires twice the amount of storage, since you would need to store the new firmware before deleting the old one. IoT end nodes intend to be cost effective and obtain a low power consumption, and we want to avoid the need for unnecessary memory storage when doing firmware updates. A solution for this is to compress the firmware sent to the device and decompress on-chip. This decompression is costly, both in terms of processing power and energy consumption. To offload the processor of this, an accelerator can be used to compress and decompress the data. An accelerator is generally more energy-efficient in performing its specific task than a general-purpose processor, as it can perform the task quicker and sooner enter power-saving modes.

Additionally, the ability to compress and decompress data with a low energy cost could open new possibilities to save power by transmitting less data and save storage cost by compressing on-chip user data. Many of the modern compression programs, such as gzip and 7zip, uses the DEFLATE compression algorithm. This is again built up by the LZSS algorithm and Huffman coding. The LZSS algorithm is the most time-consuming part due to the need for searching operations through large buffers. The goal of this thesis is to analyze a hardware implementation of the LZSS compression algorithm for use in integrated circuits.

Necessary background information needed for the LZSS algorithm, the CAM architecture which the implementation is based on will be presented in chapter 2. This chapter also introduces the metrics which will be qualitatively discussed later in the thesis.
Chapter 3 contains the methods used for testing and verifying the implementation.
The discussion of different implementation techniques with different trade-offs will be given in chapter 4.
Chapter 5 shows the results based on the metrics presented in chapter 2.
Chapter 6 discusses the implementation based on the results from chapter 5 and further improvements on the testing and implementation of the design. Some thoughts of what can be done in the future will also be given.
Finally, chapter 7 will give some conclusive remarks.

2 Background

2.1 Data Compression Theory

Shannon's source coding theorem, introduced in the article "A Mathematical Theory of Communication" [5], states that the limit of compressing data using optimal coding is given by equation 1. This is known as Shannon's entropy. Shannon's entropy is a measure of how much information a string of symbols contains. It can give us an estimate of the minimum number of bits per symbol required to represent a given stream of symbols.

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_b P(x_i) \tag{1}$$

Where b is the base of the logarithm, X is a discrete random variable and P(X) is the probability mass function.

This is given an infinite data stream of independent and identically-distributed (i.i.d) random variables. You need to know the probability distribution of the data in order to find the entropy, and in practical data compression this is something you rarely know. We usually only observe a subset of the input stream at a time. The symbols are often dependent of each other as well, especially in text compression.

Data compression consists of reducing the number of bits required to represent some information. A compression algorithm which loses information about the original data is called lossy. Through different transformations these algorithms can exploit redundancy in the data due to lack of human perception. These algorithms have generally higher compression rate, but can not be used in for firmware compression as it is essential to be able to restore the original representation when decompressing again. Lossless compression identifies and eliminates statistical redundancy in the data. No information about the original data is lost and can be decoded back to its original form.

2.2 Lempel-Ziv compression

Abraham Lempel and Jacob Ziv published in 1977 a paper a new compression algorithm. [6] As an abbreviation of their names and publishing year, the algorithm is called LZ77. LZ77 is a dynamic dictionary encoding, meaning that the input stream is encoded by a position or data in a dictionary which updates based on the new input. For the LZ77 this dictionary consists of a search buffer and a look-ahead buffer. The search buffer contains previously encoded symbols. The encoding scheme can be seen in figure 1. The look-ahead buffer contains the next sequence of symbols to be encoded. To start encoding you search backwards through the search buffer for a match with the first symbol in the look-ahead buffer. If there is a match, check the second symbols following the symbols in each buffer and so on. The sequence of symbols in the look-ahead buffer that matched will now be encoded as the length and offset representing it, as well as the next character. This will from now on be referred to as a token and is denoted as $\langle length, offset, character\rangle$, or simply $\langle 3, 4, C(*)\rangle$ where * is substituted with the next character. Length is the number of matching symbols, offset is the distance between the two first two matching symbols and character is the following symbol in the look-ahead buffer after the matching sequence. If there are no matches for the symbol it will be outputted as an ”empty” token $\langle 0, 0, C(*)\rangle$. An uncoded symbol is known as a literal. Another thing to mention is that matches are not limited to the length of the search buffer in the original algorithm. They can extend into the look-ahead buffer. As for decoding, this does not introduce any problem as the encoded stream would look either way look equal, and the decoder simply extends the copying of symbols as far as the match length suggests. The algorithm does not require any prior knowledge of statistical properties of the symbols to be encoded. This is known as a one-pass algorithm and means that it, opposite of some statistical based algorithms, does not need to go through the input sequence before starting the encoding.

Now we have a stream of tokens and literals. The decoding scheme works as shown in figure 2. To decode the token $\langle 3, 4, C(b)\rangle$ we first look a the flag bit each word has been denoted by to tell whether it is to be interpreted as a literal or not. If it is a literal we simply output this symbol and move on to the next. Else we move a copy pointer into the previous decoded symbols, to the place indicated by the offset, and copy the number of symbols that the match length number tells us to the output. We need a buffer at the output with the same size as the look-ahead buffer in the encoding phase to be able to copy these symbols.

Storer and Szymanski [7] later improved on this algorithm. Instead of encoding a literal with an ”empty” token $\langle 0, 0, C(*)\rangle$, they simply set a flag indicating that this symbol was to be decoded as a literal. The empty token would consist of the offset and length being set to 0 and the character set to the current literal. This requires more bits to represent the symbol than just the literal and a flag.

Another improvement introduced by LZSS is to discard matches which would have produced more bits as an encoded token than just send the literals to the output. The

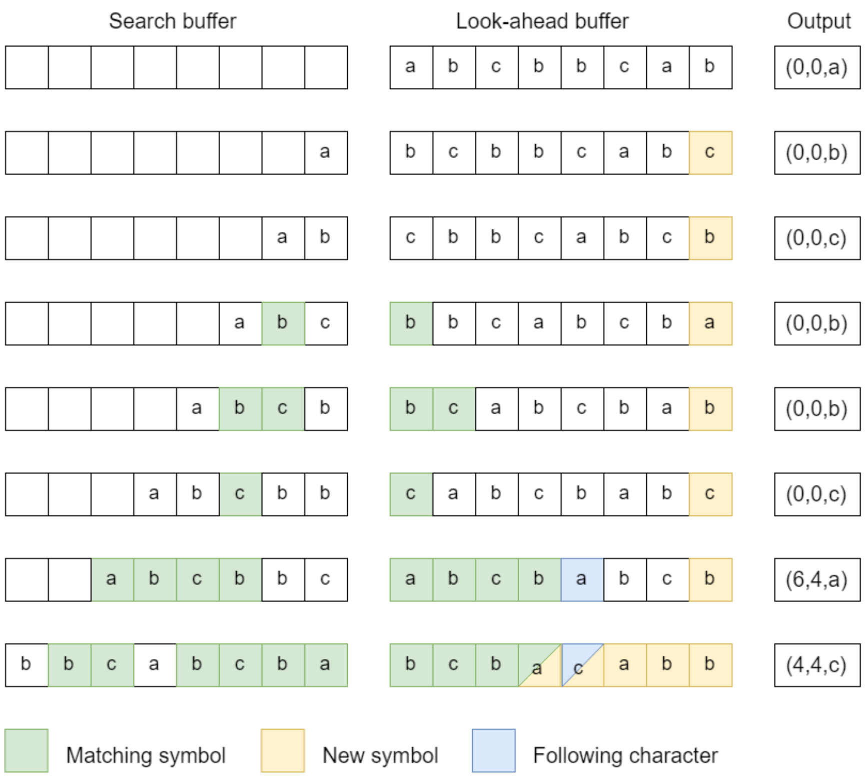


Figure 1: LZ77 encoding

optimal minimum length of the string which is being encoded will vary by the size of the sliding window, the symbol-frequency and the size of the alphabet. However, the usual minimum length is often chosen to be 4 when encoding ASCII-code, as coding strings with a size of 3 bytes or less will give us a compressed version which is the same size or larger than the literal. This will give us 1 byte for length and 2 bytes for offset. For ASCII text 1 symbol can be represented by 1 byte. The look-ahead buffer can then be $2^8 = 256$ bytes long, limited by the 1 byte representing the length. Now again, for the LZSS improvement the next character would be replaced by a flag bit indicating that the data should be interpreted as an offset/length-pair.

Higher compression rate does not give longer decompression time for LZSS as some other compression algorithms. In fact, higher compression levels gives a slightly faster decompression rate since there are fewer bits for the decompressor to process.

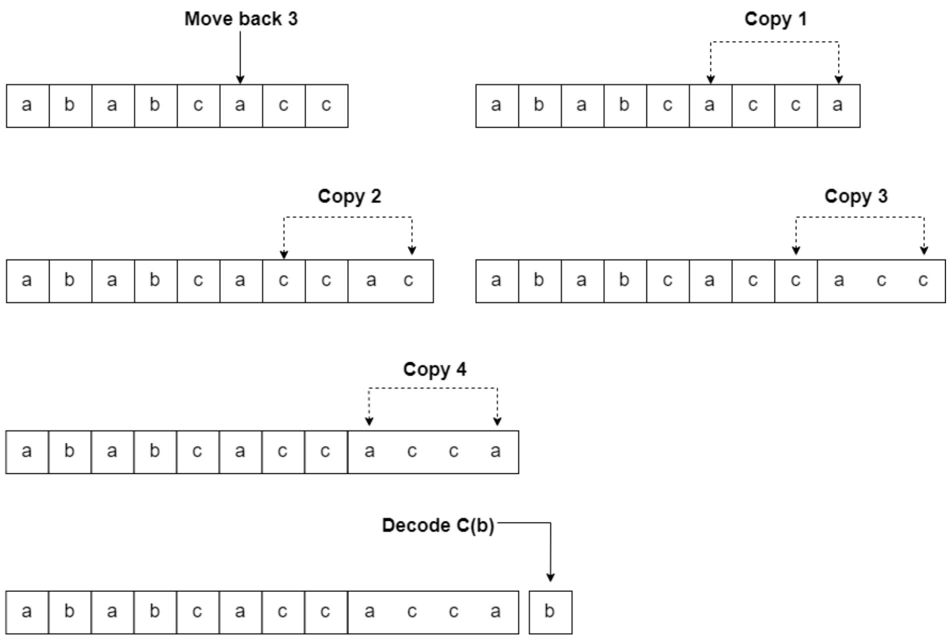


Figure 2: Decoding token $\langle 3, 4, C(b) \rangle$. Edited from [1].

2.3 Hardware Acceleration

Hardware acceleration is the performing a task on dedicated hardware rather than software running on a general-purpose Central Processing Unit (CPU). The intention is to decrease latency and increasing throughput.

A hardware implementation of an algorithm can be implemented in two different ways, either as custom instructions or as hardware peripherals. It's then either as an extension of the CPU or outside the CPU, as a hardware peripheral. Figure 3 shows the implementation of custom instructions and figure 4 shows the implementation of a hardware peripheral.To be able to completely relieve the load off the CPU, the compression accelerator can be given its own direct memory-mapping to the memory. Operations that could be performed in a few cycles should be implemented as a custom instruction as it creates less overhead. For a peripheral you typically have to execute at least a few instructions to write to the control registers, status registers, and data registers and one instruction to read the result. This takes of course longer time, but the CPU is then free to perform other tasks while the peripheral is working.

To be able to evaluate the effectiveness of an accelerator we need to look at its execution time compared to the software implementation. The total execution time of

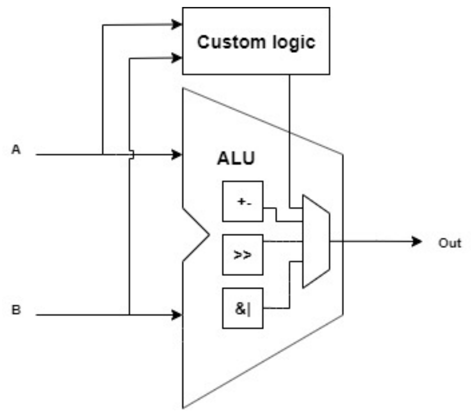


Figure 3: Accelerator implemented as custom logic

an accelerator can be written as

$$t_{accel} = t_{in} + t_x + t_{out} \tag{2}$$

where t_x is the execution time of the accelerator-core and t_{in} and t_{out} are the delays of fetching data from memory and storing back again. For an accelerator to be effective t_{accel} must be shorter than the time it takes for an equivalent software implementation. In addition, there are other constraints that need to be considered such as area, power usage and pin layout.

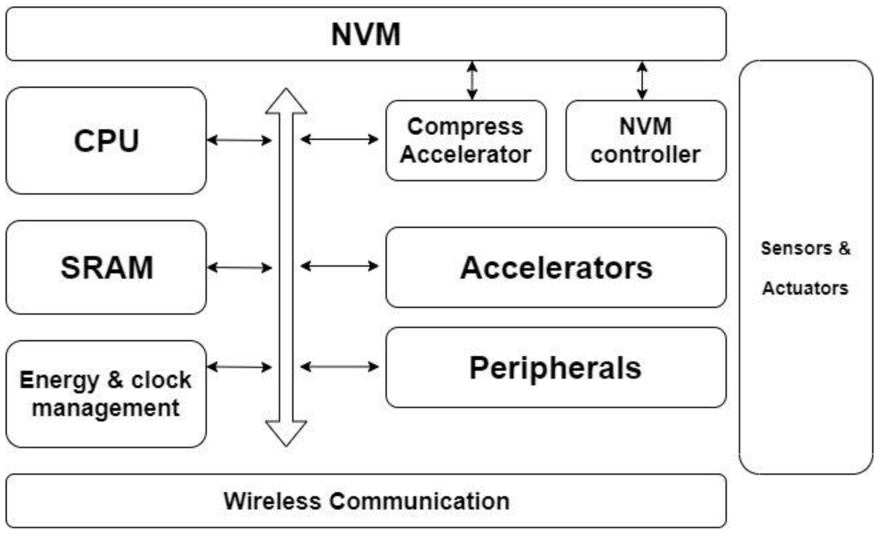


Figure 4: Accelerator implemented as hardware peripheral

There are accelerators within cryptography, graphics and artificial intelligence. A common approach to improve the throughput of an algorithm through hardware acceleration is to look for possibilities of exploiting parallelism. For lossless data compression, and especially the LZSS-algorithm, this is tricky as the encoding of each symbol is dependent on the previously encountered symbols. This creates data-dependencies which would result in incorrect encoding and a compressed file which is unreadable.

An idea could be to divide the file to be compressed into N blocks and then compress these blocks independent of each other. This would increase throughout by N. A drawback is that the area increases by each new block, since for each block it is necessary with control registers, counters, etc. to be able to detect matches. Another drawback is that the compression ratio decreases, since larger matches will not be found with smaller look-ahead and search buffers.

Even though each symbol cannot be encoded completely independently there are parts of the encoding, such as the searching and matching of symbols, which can be run in parallel. An efficient way to search through many symbols in parallel is to implement a CAM-based architecture.

2.4 Content-addressable Memory (CAM)

CAM is a memory unit suitable for high-speed searching applications. Different from the traditional Random-access Memory (RAM), CAM takes a data value as an input and compares it to the content its search lines. If there is a match with any of the search lines of the CAM, then it outputs the address of this search line as well as a valid bit indicating that it found a match. The most power consuming part of the CAM is the comparison mechanism of match lines. Figure 5 shows the architecture of a general-purpose CAM.

This memory unit can search through the entire memory and compare all search lines in a single clock cycle. This is independent of the buffer size and length of the search line. The drawback is the power dissipation.

Static Random-access Memory (SRAM)-based storage is most commonly used for CAMs cells. Each of the cells contain either a network of NAND or NOR logic. The design difference here impacts the speed and power dissipation of the design. The NAND-cell is slower, but uses significantly less power. In the following figures 6a and 6b D denotes the stored bit, SL the search line, and ML the match line.

The NOR-cell uses four transistors for comparison. In figure 6b these are marked as M_1-M_4. The M_1/M_3-pair and M_2/M_4-pair work as pull-down path from the match-line.

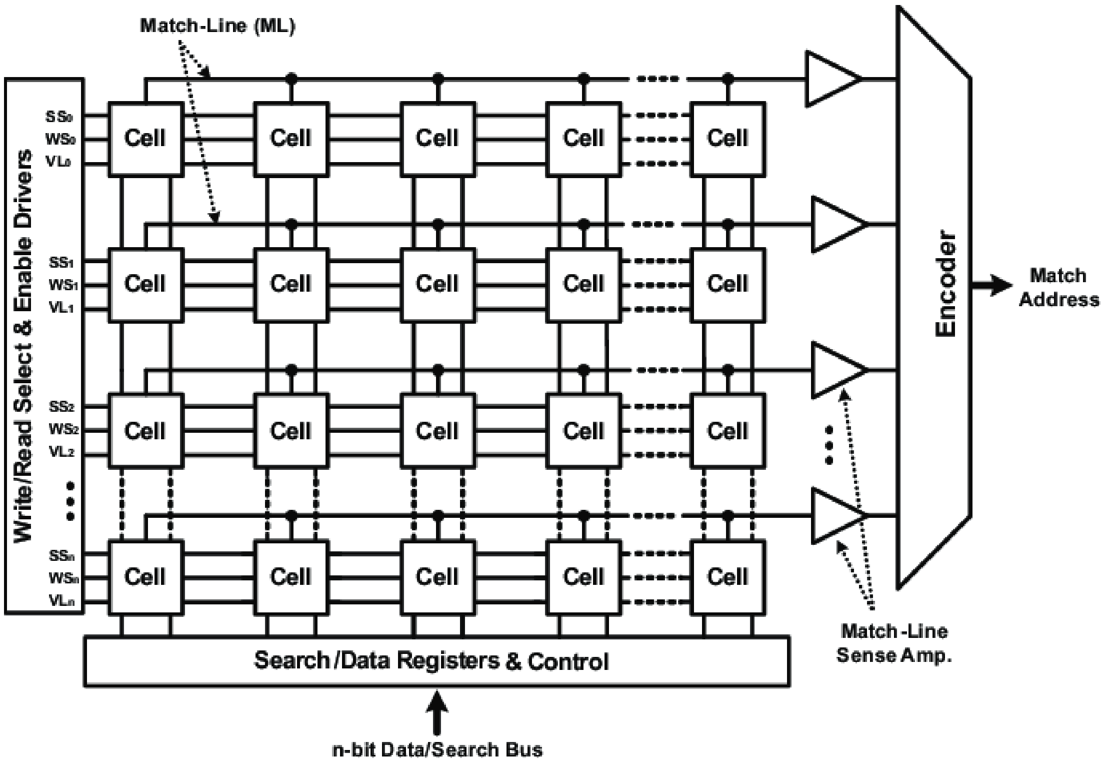
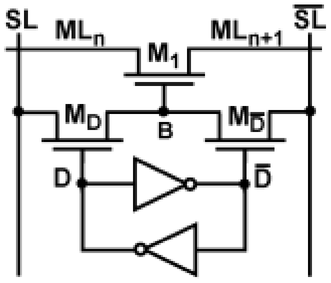
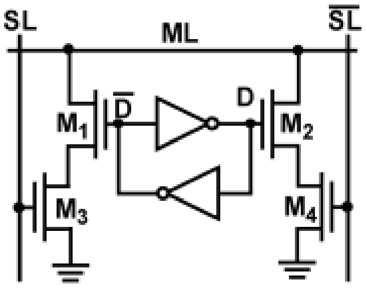


Figure 5: A standard CAM architecture for NAND cells.[2]



(a) Typical NAND-based CAM cell.[8]



(b) Typical NOR-based CAM cell.[8]

Figure 6: Cell architecture

If there is either no match with the stored bit or the search line is low, then the match line will be pulled to ground. Multiple NOR cells are connected together in parallel, and only if all of them matches will there be an asserted match line.

The NAND-cell only uses three transistors for the comparison, marked as M_1, M_D and $M_{\bar{D}}$ in figure 6a. M_D and $M_{\bar{D}}$ work as pass transistors to the node B. For either SL = 0, D = 0 ($M_{\bar{D}}$ is on) or SL = 1, D = 1 (M_D is on) the M_1 transistor is turned on. Any other combination results in M_1 turned off. All the NAND-cells are then connected serially. In both of these figures the transistors used for read and write operations are omitted.

2.5 Metrics

2.5.1 Power

Dynamic power consumption is given by

$$P_D = \alpha \cdot f \cdot 1/2C \cdot V_{DD}^2 \tag{3}$$

where α is the switching factor, f is the frequency, C is the capacative load and V_{DD} is the supply voltage.

The switching factor α and how to lower this to reduce dynamic power consumption will have the largest focus during the discussion of the design. Other techniques such as block-level clock gating, must be evaluated in compliance with the larger system, and is out the scope of this thesis.

2.5.2 Compression Ratio

The compression ratio is given as shown in equation 4.

$$\texttt{Compression ratio} = \frac{\texttt{Uncompressed size}}{\texttt{Compressed size}} \tag{4}$$

Higher compression ratios are desired, as this give lower output file sizes.

2.5.3 Area

The area of the design implemented will be measured in amounts of Lookup Tables (LUTs) and Flip-flops (FFs) and number of muxes used.

3 Methodology

3.1 Design Methodology

The design will be written in the Hardware Descriptive Language (HDL) SystemVerilog. The design will be simulated in Vivado 2018.2 using the Vivado Simulator. The Vivado Simulator does not fully support SystemVerilog Assertion (SVA). Questa Simulator from Mentor Graphics will be used instead for verification. This is only for the formal verification of the design and will not affect the synthesized design.

3.2 Testing Methodology

3.3 Test and Verification Methodology

3.4 Benchmarking

3.4.1 Calgary Corpus

Calgary corpus was created in 1987 by Ian Witten, Tim Bell and John Cleary from the University of Calgary.[9] The corpus consists of 14 files totaling 3,141,622 bytes. Calgary Corpus is normally not widely used today due to its small size. However, this applies to testing developing compression programs for personal computers. For microcontrollers, where the firmware size is generally below 1 MB, Calgary Corpus works great. There are other corpora such as Canterbury [10] and Silesia [11]. As shown in table 1 Calgary Corpus consists of several different test models, including English text, programming source codes and numbers. While this thesis mainly focuses on compression of firmware source code, the accelerator is general-purpose and can be used for e.g compression of sensor data or miscellaneous data sent to the main memory of the microcontroller.

To be able to compare to other implementations of compression algorithms it is essential that the testing data is the same. If this was not the case then the test input could be manipulated to fit the algorithm under test, and we would only achieve a non-comparable compression rate.

The results from the benchmark are usually added to a weighted average of all the models.

The generator consists of clock generation and drivers for reset signals and valid data valid signals.

Table 1: Table over Calgary Corpus benchmarks

Model	Size(bits)	Description
BIB	111,261	ASCII text in UNIX ”refer” format - 725 bibliographic references.
BOOK1	768,771	unformatted ASCII text - Thomas Hardy: Far from the Madding Crowd
BOOK2	610,856	ASCII text in UNIX ”troff” format - Witten: Principles of Computer Speech.
GEO	102,400	32 bit numbers in IBM floating point format - seismic data.
NEWS	377,109	ASCII text - USENET batch file on a variety of topics.
OBJ1	21,504	VAX executable program - compilation of PROGP.
OBJ2	246,814	Macintosh executable program - ”Knowledge Support System”.
PAPER1	53,161	UNIX ”troff” format - Witten, Neal, Cleary: Arithmetic Coding for Data Compression.
PAPER2	82,199	UNIX ”troff” format - Witten: Computer (in)security.
PIC	513,216	1728 x 2376 bitmap image (MSB first): text in French and line diagrams.
PROGC	39,611	Source code in C - UNIX compress v4.0.
PROGL	71,646	Source code in Lisp - system software.
PROGP	49,379	Source code in Pascal - program to evaluate PPM compression.
TRANS	93,695	ASCII and control characters - transcript of a terminal session.

The stimulus is divided into two parts. First is the Calgary Corpus presented in section 3.4.1 which is needed to be able to benchmark the design under test against other compression methods. This is a large testbench covering multiple input data scenarios, but it’s also static and we have no explicit control over the input. That is why we also need directed stimulus for testing corner cases. This tests scenarios we know often contain bugs, such as asserting signals when they are not suppose to, under- and overflow of array contents, and etc. These tests are also dangerous as they often just confirm that your thought-process is right, and not trying to actually make the system fail.

The best way to find bugs in a system is to use random testing. However, it is impractical to write every possible data value for any scenario. For increasingly large system the simulation time grows to the point where it is impossible to test everything. So we need constraints to limit the test input. This is where we can utilize SystemVerilogs object-oriented features to make test classes for directed random testing.

One thing to be aware of is that SystemVerilogs randomize()-method is only pseudo-random and that the seed for each ”process” based on each simulation stays the same. In this context is the program-block, every thread, object, function or task call each has a separate process. So when we use the randomize-method on the stimulus objects the initial results from the simulation are random, but then every new simulation iteration from the same testbench will produce the same results. This method is also implementation dependent, meaning that the stimulus generated can vary between different simulators.[12]. This is to ensure random stability. Random stability make sure that the errors are easily reproduced, which lowers debugging time.

The standard setup for testing and verify a design in SystemVerilog is shown in figure 7.

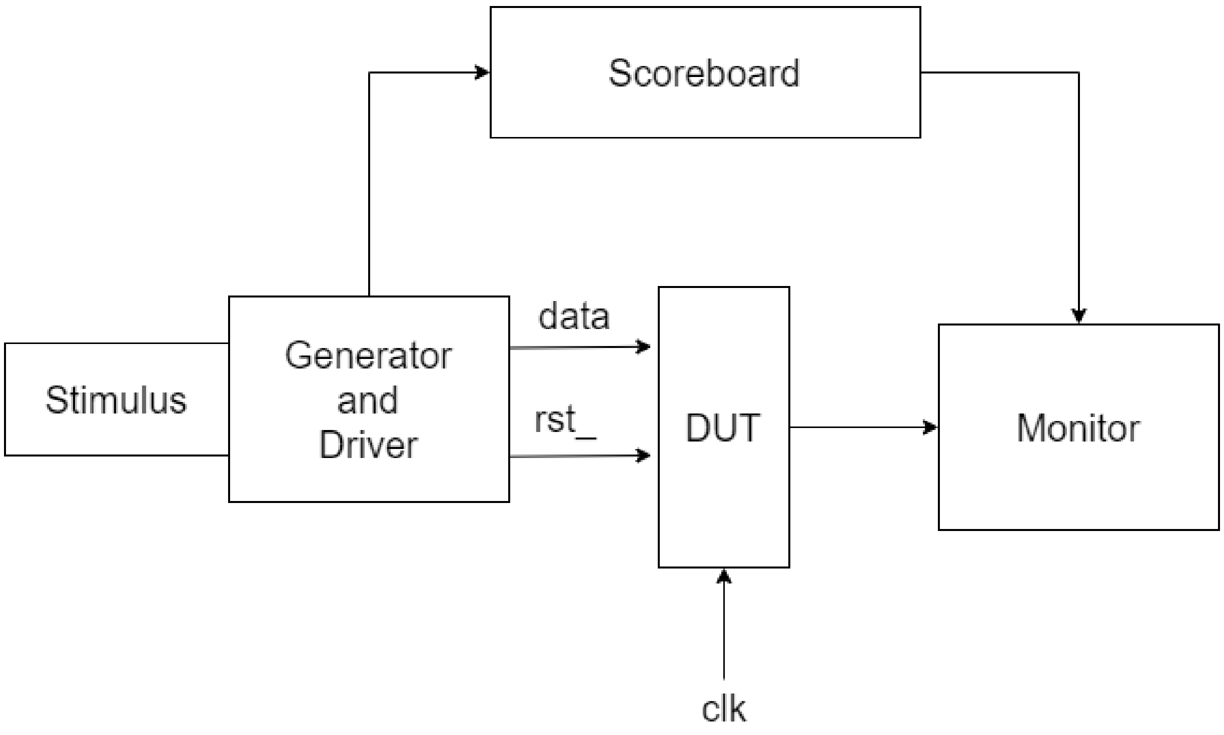


Figure 7: Block diagram of the test and verification method.

Generator

This module generates the stimulus.

Driver

The Driver applies the generated stimulus to the Design Under Test (DUT).

DUT

This is the current module that is being tested. For our case it is either the encoding or decoding module. The submodules have not been tested separately.

Scoreboard

The scoreboard receives the expected and actual values and see if they match.

Monitor

The monitor contains all the immediate assertions used to verify the DUT.

The code for testing and verifying the design is given in the appendix.

4 Implementation

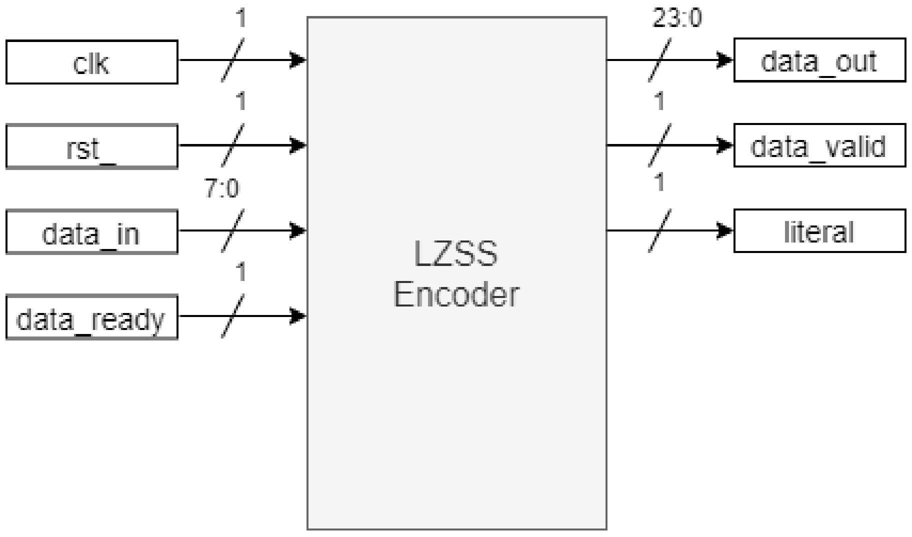


Figure 8: Top module of the LZSS encoding.

The design is based of a reference code written in C by Haruhiko Okumura[13]. Storer and Szymanski never implemented their algorithm, so this is the closest to a reference code that is available. LZSS encoder top module is shown in figure 8.

4.1 Search buffer

The search buffer is implemented as a byte-addressable packed 2D array. This 2D array has a row-major matrix order, this applies obviously to both the shifting direction and the matching direction. Every array in the design is implemented as little-endian. It could be an idea to infer the general-purpose CAM template XAPP1151[14] from Xilinx for this project. There are several restrictions for our design to rather create a special-purpose CAM. The CAM used in this design has some modifications to the standard design presented in chapter 2.4. First off the replacement policy is First-In-First-Out (FIFO). Using the template CAM we could keep track of the least recently written address and then overwrite the data on this address with the next symbol data. This write operation takes 16 clock cycles. Using a shift register instead enables us to write the new data in 1 clock cycle. The operating speed of the shift register is constant regardless of the buffer size, if we discard the clock growth load. The drawback is that a shift register uses a lot more power since it can possibly toggle all FFs in one shift-operation.

There will never be a need to read from the CAM, except for matching, but the sequential order of the symbols in the CAM is essential for the LZSS encoding to work. Since we are not going to write to specific parts of the CAM there is no need for a decoder to decide where to write.

The whole CAM must also be shiftable in order for the FIFO replacement policy to work. One thing to be aware of then is the boundary conditions of each row. Since the whole CAM is to be shiftable we must shift from the last column in each row into the first column in the next row. The same applies to the matching and in particular the implementation of the `match_matrix`.

Before every match, barring the first one, we check whether the symbol to the left got a match in the previous matching cycle. If it did not, we simply do not compare this word as it would not result in a matching string anyway. By doing this for every word we can create a mask for the entire CAM. This reduces number of word necessary to compare drastically and we can save a lot of power. For the situation where there are more than one matches with the longest length we need an priority encoder to choose which offset to output. This is discussed closer in chapter 4.2.1.

4.2 Encoding

One way to encode the input stream is to find the first matching string and then output this and shift the number times equal to the offset. This would however only be marginally faster performing the matching sequentially. Its still necessary to count the number of consecutive matches to be able to determine the length of the matching string. This would take a number of clock cycles equal to the match length to count. What becomes obvious then is that a long string quickly requires a large tree-structure of AND-ports to check if the string matches. A better alternative is only look for matches for one look-ahead buffer symbol at a time. The search for a match for the symbol can be done in parallel for every symbol in the search buffer. To reduce the amount of comparisons required, which reduces switching activity, we can use a masking registers. This is implemented as following. Every time a match between the look-ahead buffer and the search buffer is found, the representing bit in the `match\_matrix` is set. When we in the following clock cycle try to match the next symbol in the look-ahead buffer, only the symbols following a previous match, i.e. `search_buffer[i][j-1]`, will be checked. If the match matrix shows a match on the Least Significant Byte (LSB) of the search buffer rows, then we should check for a match on the Most Significant Byte

(MSB) of the next row. The first symbol in the search buffer must be checked by itself, since there is no previous symbol that could have been matched. This is only necessary to do when the match length is 0, as this is the only time when this symbol can match. Another special case is if there is a match at the end of the search buffer. We can not abort the comparing and say that this is the longest match, as we might find longer matches simultaneously at other positions in the search buffer. The matching procedure is shown in figure 9.

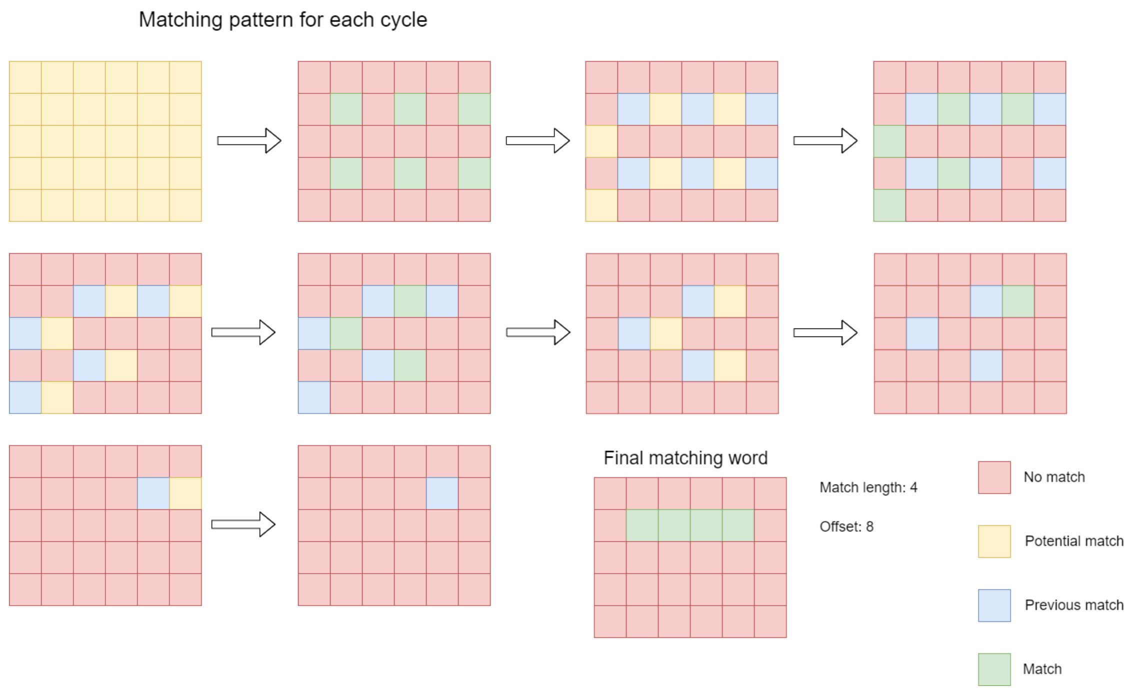


Figure 9: Matching scheme for the LZSS encoding.

In order for the for-loops to be synthesized into multiple parallel assignments of the match matrix they need to be able to be statically unrolled.

As mentioned in chapter 2 the lower limit of a matching sequence resulting in a token is 4 symbols. The upper limit is the fact that the match length is stored in 8 bit array, which would result in a max length for the look-ahead buffer of 256 symbols.

4.2.1 Priority encoder and offset calculation

For the case where there are multiple matches for the longest match length we have to decide which to choose. As long as the match length and offset can be represented by 1 and 2 bytes, respectively, it does not matter which match to choose. However, for simplicity we choose the match with the lowest match offset. For this we need a priority encoder which gives us the row and column in the first set bit in the match matrix. It is important that the priority encoder is parameterizable in order to be able to change the sizes of the search buffer at configuration time. This is implemented by traversing through the match matrix until we find the first set bit. This is implemented as a combinational circuit, and shown as code in the appendix. Since SystemVerilog does not support `break` in nested loops we need to set a flag in the inner loop which activates a `break` in the outer loop. A problem is that the method for checking that we have found the longest match is to see if all the bits in the match matrix are 0. Then the information of where the matches end is lost. There are two possibilities for solving this and compute the offset of the match. We can either use an extra copy of the match matrix on the previous clock cycle and then compute the offset from this matrix. The other option is to compute the offset for every match when match length is above 3. When the `match_matrix` then is 0, we use the previous computed offset. The advantage of this method is that we reduce the number of FFs used compared to the other method, but it requires many unnecessarily computations and the critical path becomes longer.

The LZSS decoder reads the offset as distance into the decode buffer so we need to translate the (row,col)-position into a distance into the search buffer. This is simply done by multiplying the row number with the search buffer width and then adding the number of columns in the match is. This matching symbol is the last in the sequence, and the offset is given by the first matching symbol. So we must subtract the offset of the last matching symbol by the match length to get the correct offset. For timing purposes this is done by subtracting the column position given by the priority encoder with the match length. This will give the correct offset, but we need to take in consideration the case where a match stretches over 2 rows. It is important to remember that the search buffer rows are little-endian and that the offset will be given as if it is big-endian. This means that an offset of 1 is the upper left-most symbol in the search buffer.

This gives us the shortest offset, but it will not give any improvement on speed during decoding. As long as the offset is under 256, and we can represent it with one byte there will not be any difference in compression ratio either.

4.2.2 Finite-state-machine

The state machine used for control signals of the encoding module is a Moore type. It is using one-hot state encoding for lower switching activity. This gives lower power consumption, and smaller likelihood for glitches. It does however use more flip flops than a binary encoding. A diagram of the different states and their transitions is shown in figure 10.

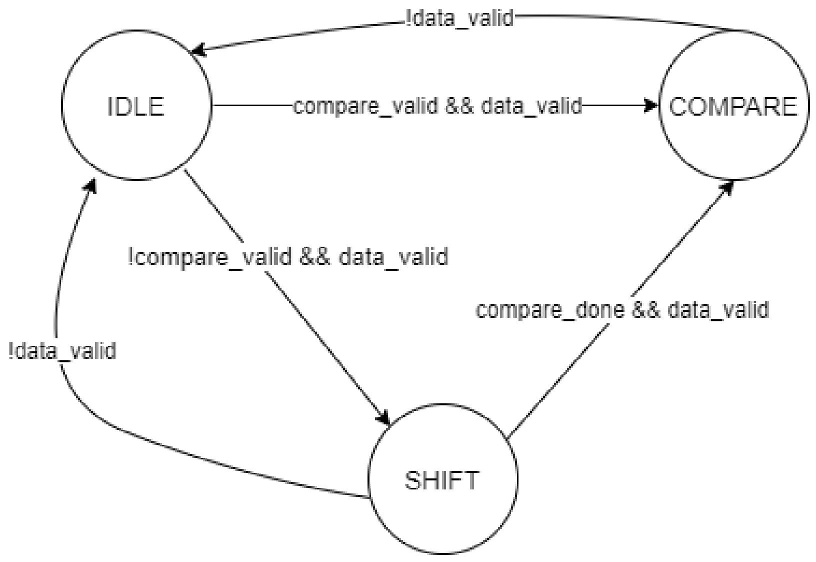

!data_valid
IDLE
compare_valid && data_valid
COMPARE
!compare_valid && data_valid
compare_done && data_valid
!data_valid
SHIFT

Figure 10: State machine for the encoding module.

4.3 Decoding

The LZSS decoding top module is shown in figure 11. In the decode module we need a decode buffer for storing the previous encountered symbols. When there is a token to decode, we copy ”length” number of symbols in the decode buffer starting from the position indicated by the offset. The sequence that is copied will be sent to the output one symbol at a time. The sequence will also be copied to the MSB of the decode buffer. The decode buffer therefore needs to be a Serial-In-Parallel-Out (SIPO) buffer to be able to output from the buffer without shifting through it. By having another buffer for before the output to store the copied symbol sequence we can pipeline the design. We can then shift in the sequence to the decode buffer and output one symbol per clock cycle. The alternative would be to output directly from the decode buffer and then shift the sequence back to the beginning of the decode buffer again. This saves

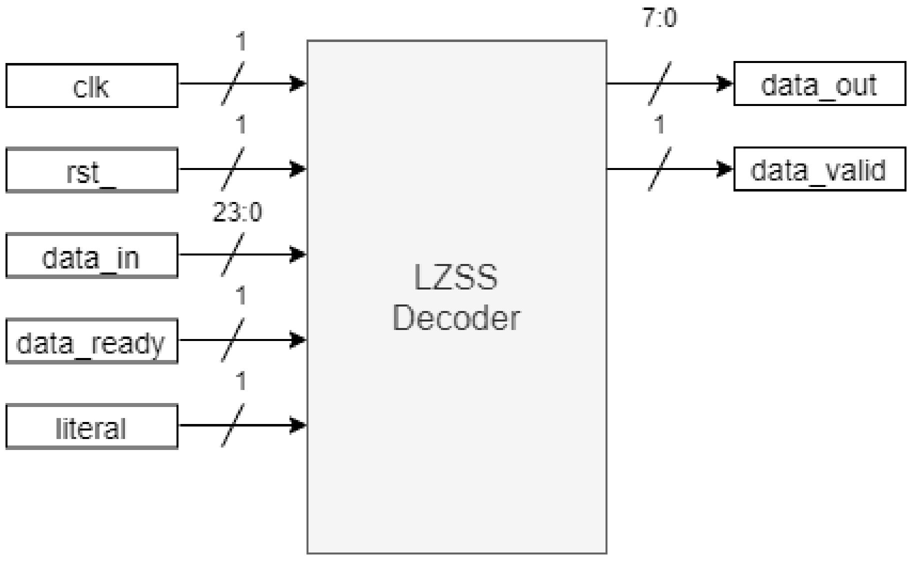


Figure 11: Top module of the LZSS decoding.

area, but uses twice as long time to decode and output a token. The literals will be sent directly to the output and shifted in the decode buffer. This is shown in figure 12.

In order to implement this we need to assign the content of the decode buffer to the output buffer. This buffer will act as Parallel-In-Serial-Out (PISO) shift register.

4.4 Handshaking

Between the encoding module and decoding module it has been implemented a simple single clock-domain handshaking protocol consisting of a `data_valid` and `data_ready` signal. Only when both signals are asserted will there be a transfer of data. `data_ready` can be set independently, but `data_valid` is only set when the `data_ready` is asserted and data is ready. If `data_valid` is asserted for more than one clock cycle it is to be considered a new packet. This is shown in figure 13.

The handshaking protocol is implemented between the encoding and decoding modules and between the DUT and the testbench. This is to ensure we encode every symbol in the input stream.

4.5 Testing and benchmarking

The benchmarking is done through the files mention in chapter 3.4.1. The encoded symbols are written to new files. These can then be read by the decoding module for

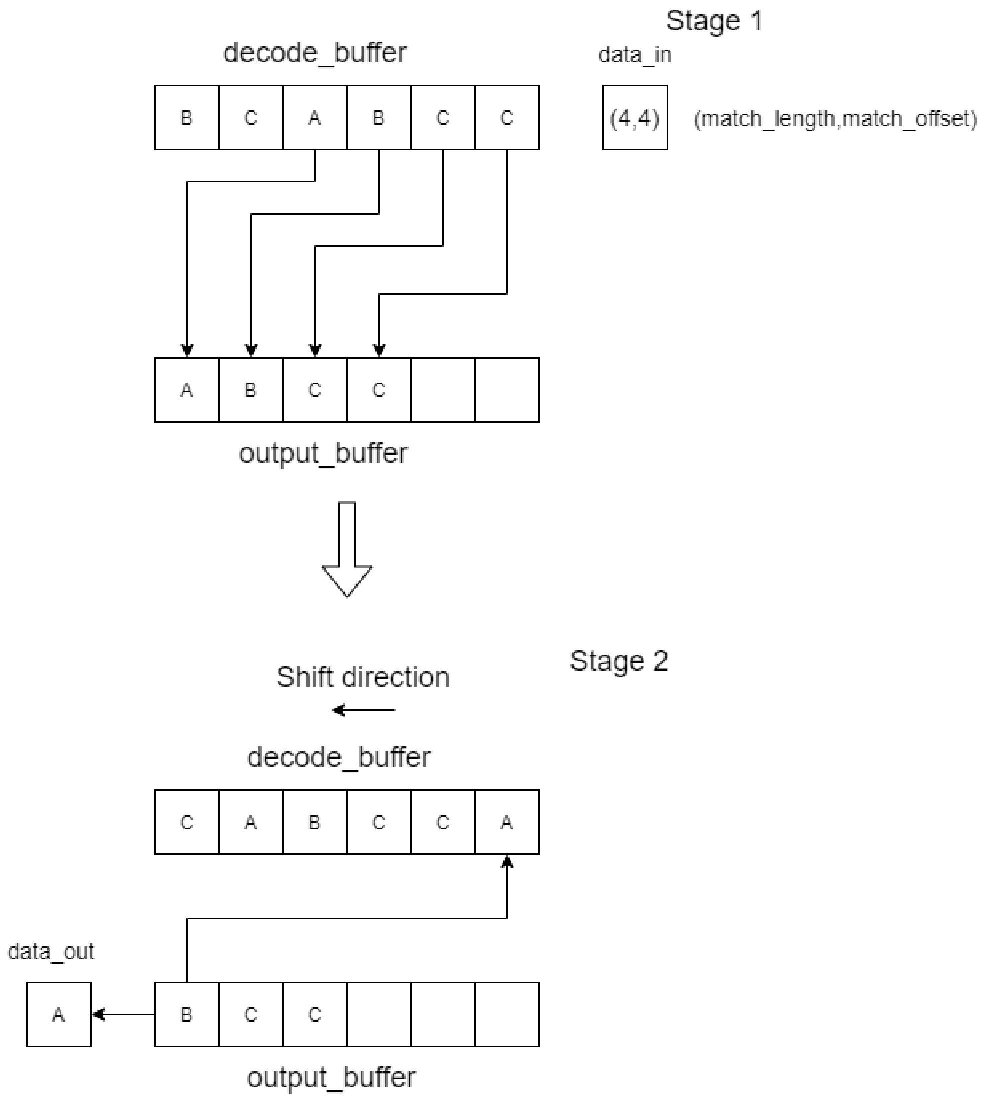


Figure 12: LZSS decoding.

benchmarking and functional verification. The compression ratio can not be computed from the sizes of these files, as they contain meta data about line breaks and etc. The encoded symbols are also written to the file as binary where each of the 1s and 0s and coded as 8-bit number. Instead the compression ratio is computed by ratio of symbols read from the benchmark files and symbols outputted from the encoding module plus the literal bit per symbol.

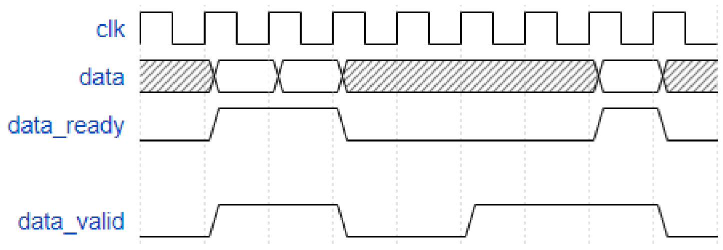
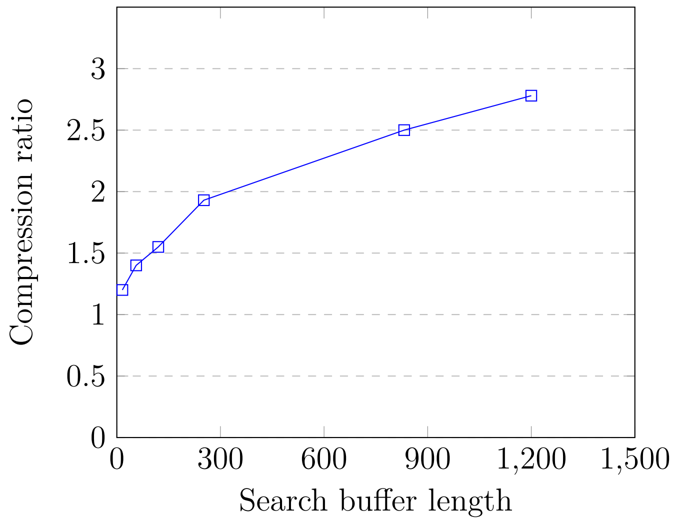


Figure 13: Ready/Valid handshaking protocol.

5 Results

For the remaining results the look-ahead buffer length will be fixed to the search buffer width. Figure 14 shows the different compression ratios for the different buffer sizes. The look-ahead buffer size is here fixed at the size of the search buffer width.



[ht]

Figure 14: Encoding: Compression ratio vs. search buffer length

Figure 15 shows the different compression speeds for the different buffer sizes.

Table 2 includes the area of each synthesized design, as well as the information presented in the two graphs above.

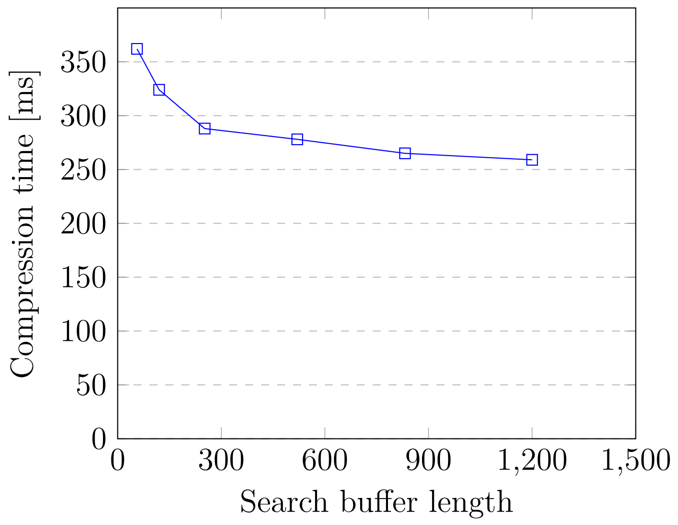


Figure 15: Encoding: Compression time vs. search buffer length

Table 2: Table over compression ratio and time for different buffer sizes

Search buffer width	Search buffer depth	Look-ahead buffer length	Compression time [ms]	Compression ratio	Area
7	8	8	324	1.40	472 LUTs, 682 FFs and 8 muxes
10	12	12	324	1.63	982 LUTs, 1290 FFs and 14 muxes
14	18	18	288	1.86	1799 LUTs, 2526 FFs and 24 muxes
26	32	32	265	2.50	5422 LUTs, 7858 FFs and 32 muxes
30	40	40	259	2.78	8910 LUTs, 11234 FFs and 44 muxes

6 Discussion

As expected, figure 14 shows that the compression ratio increases with a larger search buffer. The maximum match length for a token is limited by the length of the look-ahead buffer. The way the design is implemented there can not be a token with length longer than the search buffer width. That is why the look-ahead buffer length and the search buffer width have been kept equal during the benchmarking. An increase of the search buffer depth makes it more likely to find a matching token. An increase of the search buffer width gives the possibility of finding tokens with longer lengths.

Figure 15 shows that the compression speed slightly decreases as larger buffers are used. This is unexpected since the initial thought would be that the encoder would use longer time searching for longer matches. However, for any match or literal the encoder uses two clock cycles to enter and exit the COMPARE-state. One to assign matching symbol positions in the `match_matrix` and then one to check if the `match_matrix` is all 0's. Then it enters the SHIFT-state to send out the token or literal to the output. This process is the same no matter how long of a match has been found. The number of times we have to switch state per symbol outputted decreases the longer the buffers are. So long matches will have fewer of these transition cycles per encoded symbol, and will therefore encode the symbols faster. Whenever there is a match length between 0 and 3 the design will enter the SHIFT state and shift the buffers one slot to the left. Then the next symbol will be compared. An improvement might be to check if the first symbol matched with over half of the symbols in the search buffer. If it did then it would not be worth checking the next symbol in the look-ahead buffer as we have already checked it with most of the symbols in the search buffer. We would then output the literal while in the SHIFT state. This way there would be no need to enter the COMPARE state again for the next symbol and we would save 3 or 6 clock cycles, depending on if it is 2 or 3 symbols.

Another slight improvement is to stop matching if there is only a match in one of the last three symbols of the search buffer. This only applies if this is the first matching try, i.e. trying to match the first symbol in the look-ahead buffer. At this point there can not be any long enough matches to be found. The returns of any attempt to avoid this are diminishing as the search buffer size increases.

Both look-ahead buffer and search buffer is initialized with unknown values "X". If they had been initialized to 0's we would have created a false symbol history which would not concur with the input stream. `compare_valid`,which enables the COMPARE state, is only set when all values of the look-ahead buffer are known. For simulation purposes this is fine, but for synthesis this must be fixed. This can be done by having a initialization period with a counter that keeps track of valid input in the search buffer. This is to avoid trying to match the content of the look-ahead buffer with unknown values. When the counter reaches the size of the search buffer the initialization period is over. Another solution would be to have an invalid symbol which we know would not match. This would however limit the alphabet size.

6.1 Further work

A naturally step further to expand on this project would be to implement further encoding with e.g. Huffman coding. This is how the DEFLATE program compresses. Implementing the corresponding file format is important to ease the use of the design. Another interesting idea would be to compare this design to a design using the general-purpose CAM which outputs addresses. It would be necessary to keep track of the how long each symbol has been in the search buffer, and implement a Least Recently Used (LRU) replacement policy. If the same method of only checking a symbol at a time is to be used there have to be a way to ensure the sequential order is kept. Say you have a matching symbol and want to check the next one for a longer sequence. The next symbol you would need to check then is the one that has been in the search buffer one stage longer. Opposite of the sequential locality of this design there would be a time-sequential order. Since there would not be a need to use a shift register for the buffers the switching activity drastically drop which decreases the dynamic power consumption.

Another parameter which will the interesting to inspect is to increase the symbol length of our design. Multiple symbols will then be matched at the same time, which will increase the throughput. The backside of this method is the decrease in resolution of finding matches for each increase in symbol length. A symbol length of 5 bytes could potentially miss the longest match by 4 symbols. The design is parameterized to utilize this, but it has not been tested or verified its functionality.

An idea would be to combine the decode and encoding module together. This way we could reuse the the look-ahead buffer to be utilized as the decode buffer as well. This would increase the complexity of the design, and could there could be some floorplanning issues related to this design.

This design could be split into multiple parallel instantiations of the encoding/decoding modules as discussed in chapter 2.3. How this would compare to just one instantiation with the same area constraint could also be interesting test.

7 Conclusion

This thesis presents a hardware implementation of the encoding and decoding of the LZSS compression algorithm. The encoding module is based on an application-specific CAM. By implementing a the `match_matrix` to keep track of previous symbols that matched several redundant comparisons can be removed. This lowers the power dissipation due to the fall of switching activity in the CAM. The results show a decrease in compression time for higher buffer sizes. This is surprising, but can be explained by excessive state transition when outputting literals. The compression ratio increases as buffer sizes increases. The is due to the possibility of finding longer matches, which in turn increases compression ratio. The decoding module is implemented using an extra buffer before the output to pipeline the decoding. This is to reduce the time it takes to decompress. Both the encoding and decoding module have been tested and verified to work as intended.

Bibliography

[1] Khalid Sayood. Introduction to Data Compression - 5th Edition. https://www.elsevier.com/books/introduction-to-data-compression/sayood/978-0-12-809474-7.

[2] Kamran Eshraghian, Kyoung-Rok Cho, Omid Kavehei, Soon-Ku Kang, Derek Abbott, and Sung-Mo Kang. Memristor MOS Content Addressable Memory (MCAM): Hybrid Architecture for Future High Performance Search Engines. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19, May 2010.

[3] Ericsson. Ericsson Mobility Report November 2016. Technical report, November 2016.

[4] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, Fourthquarter 2015.

[5] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948.

[6] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[7] James Storer and Thomas Szymanski. Data compression via textual substitution. *J. ACM*, 29:928–951, October 1982.

[8] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, March 2006.

[9] Timothy Bell, Ian Witten, and John Cleary. Modeling for Text Compression. *ACM Comput. Surv.*, 21:557–591, December 1989.

[10] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings DCC '97. Data Compression Conference*, pages 201–210, Snowbird, UT, USA, 1997. IEEE Comput. Soc. Press.

[11] Sebastian Deorowicz. Silesia Compression Corpus. http://sun.aei.polsl.pl/˜sdeor/index.php?page=silesia, 2003.

[12] Subbdue. Systemverilog.io. https://www.systemverilog.io.

[13] Haruhiko Okumura. LZSS encoder-decoder. https://oku.edu.mie-u.ac.jp/~okumura/compression/lzss.c, 1989.

[14] Kyle Locke. Parameterizable Content-Addressable Memory. page 31, 2011.

A Appendix

The code for the design, testbench and verification can be found at `https://github.com/halvho/LZSS-hardware-implementation`.