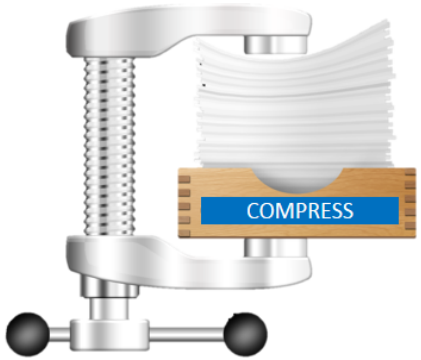
Evaluation of Fast-LZ Compressors for Compacting High-Bandwidth but Redundant Streams from FPGA Data Sources

Author: Luhao Liu
Supervisor: Dr. -Ing. Thomas B. Preußer
Dr. -Ing. Steffen Köhler

09.10.2014

Why needs compression?

- Most files have lots of redundancy. Not all bits have equal value.
- To save space when storing it.
- To save time when transmitting it.

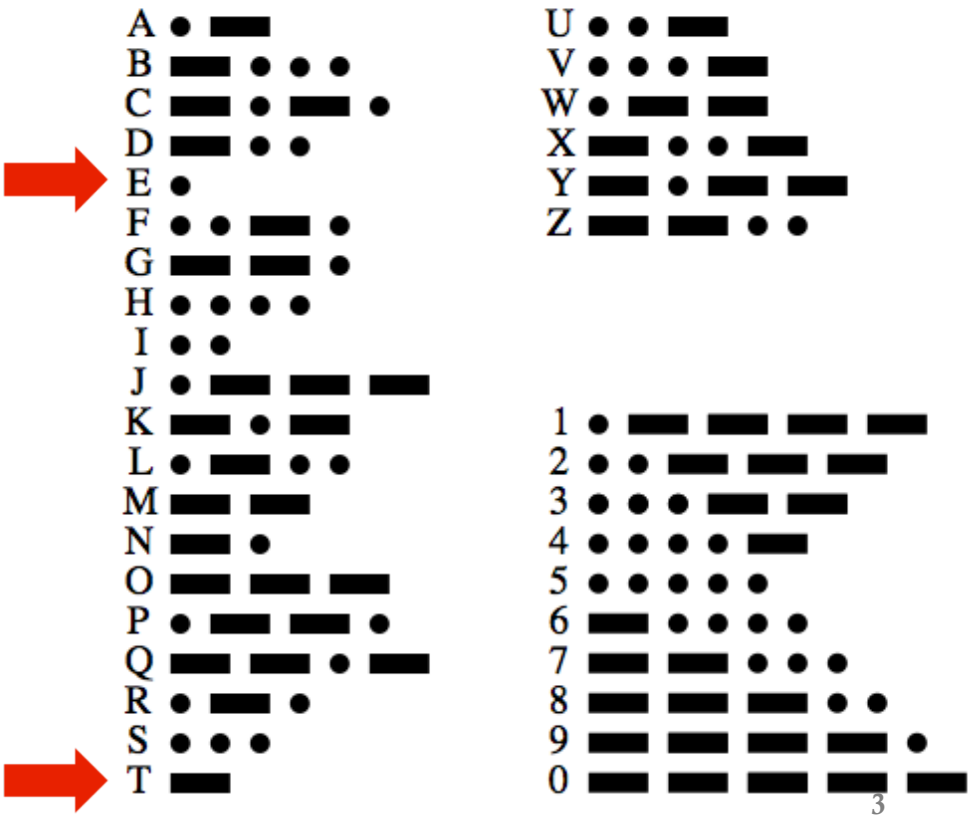


Who needs compression?

- Moore's law: # transistors on a chip doubles every 18-24 months.
- Parkinson's law: data expands to fill space available.
- Text, images, sound, video, …

International Morse Code [1]

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

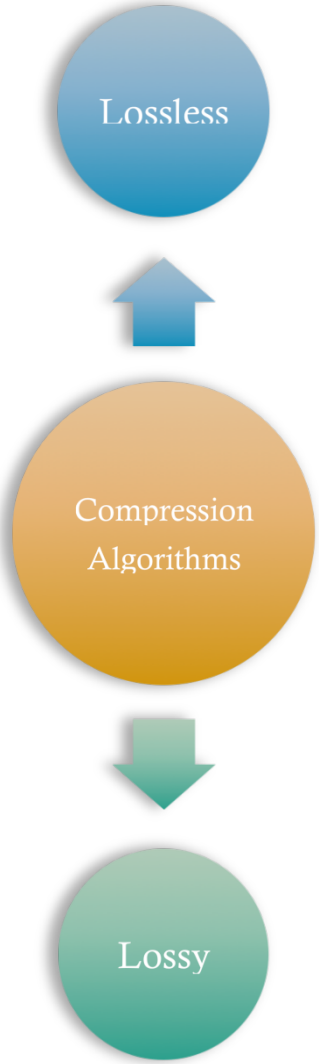


Morse code, invented in 1838, is the earliest instance of data compression in that the most common letters in the English language such as “e” and “t” are given shorter Morse codes.

General Applications

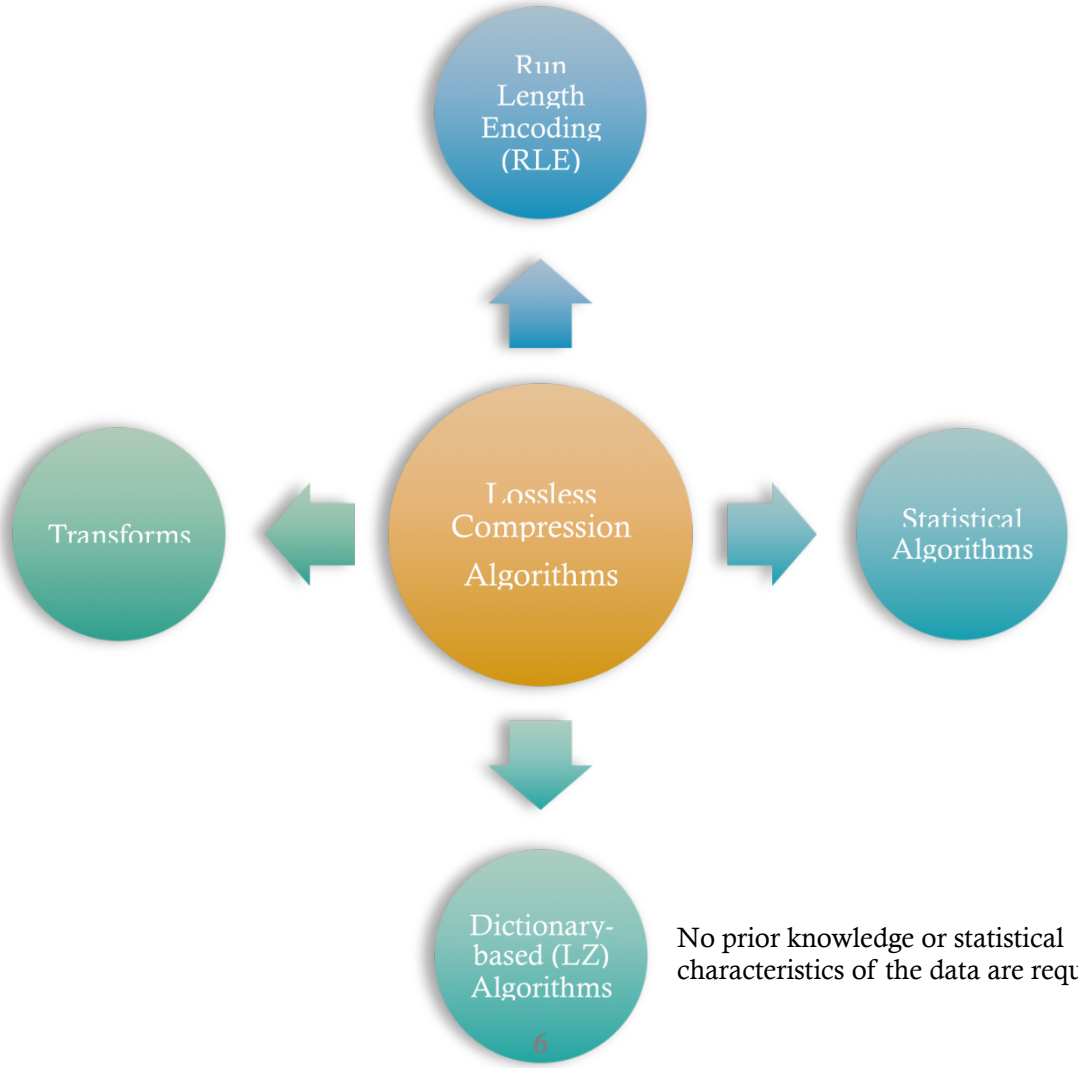
Generic file compression	• Files: GZIP, BZIP, BOA • Archivers: PKZIP • File systems: NTFS
Multimedia	• Images: GIF, JPEG • Sound: MP3 • Video: MPEG, DivX™, HDTV
Communication	• ITU-T T4 Group 3 Fax • V.42bis modem
Databases	• Google

[2]



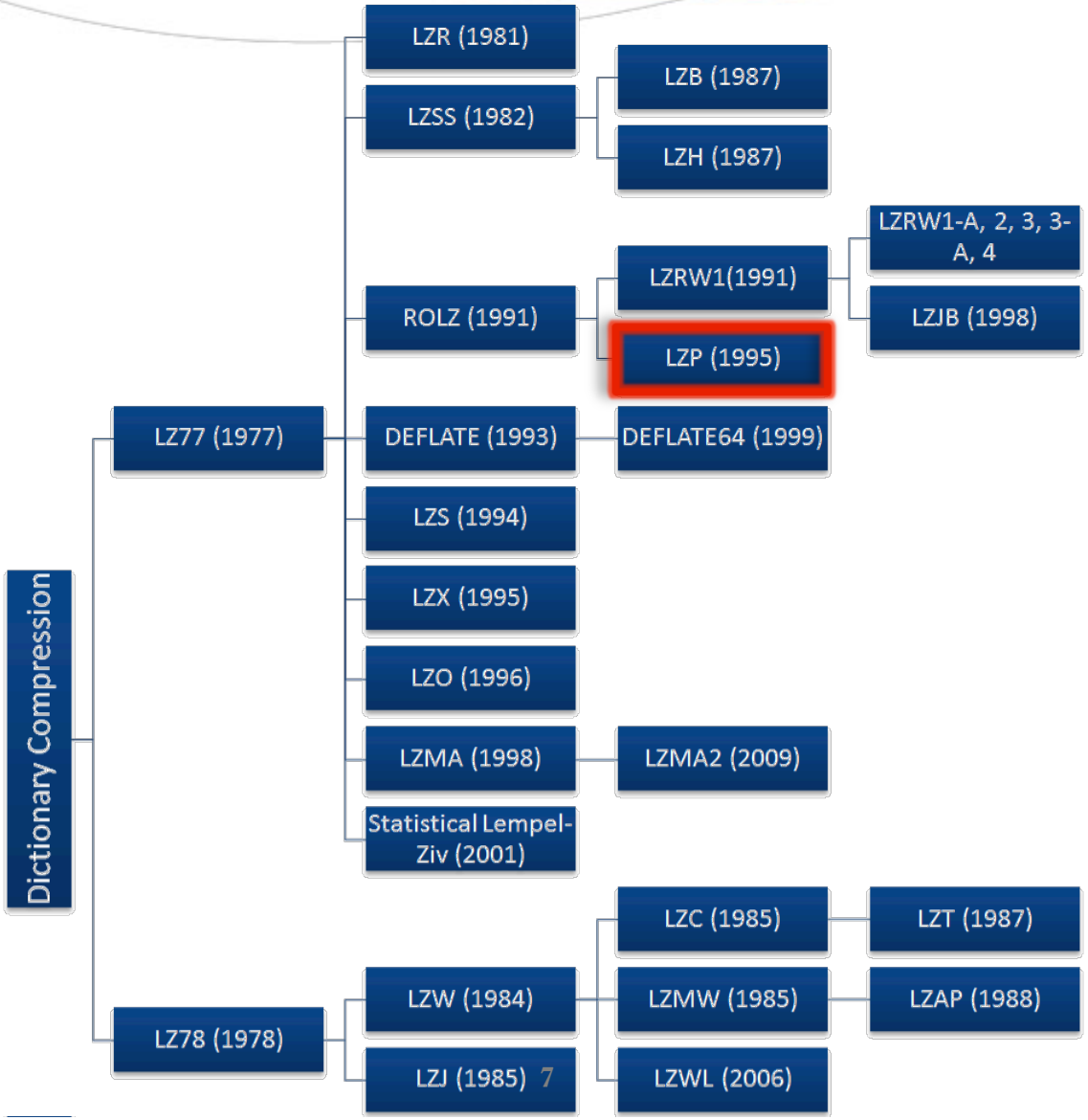
Used for text, file may become worthless if even a single bit is lost.

Used for for image, video and sound, where a little bit of loss in resolution is often undetectable, or at least acceptable.



No prior knowledge or statistical characteristics of the data are required.

A Hierarchy of Dictionary Compression Algorithms from 1977 to 2011

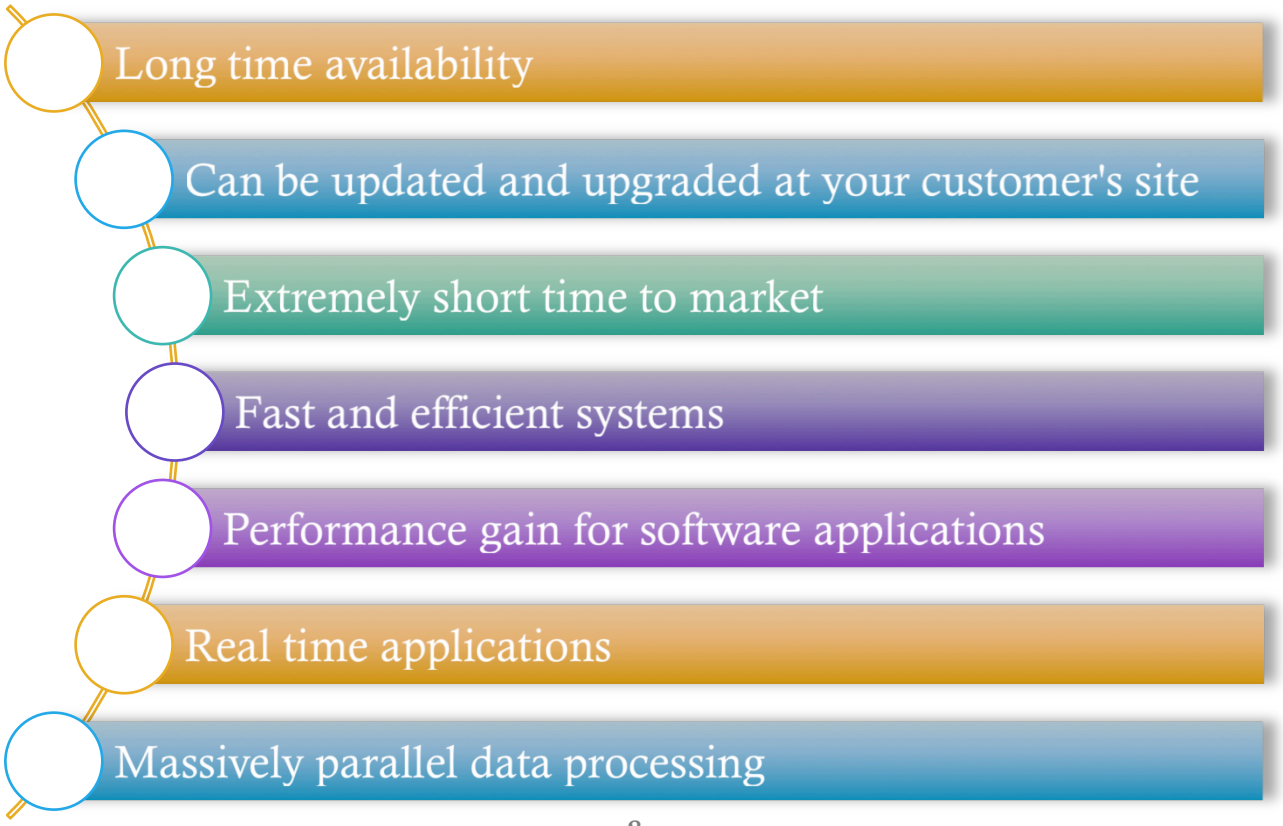


[3]

Benefits of FPGA Implementation

- Hardware implementation of data compression algorithms is receiving increasing attention due to exponential expansion in network traffic and digital data storage usage.

- FPGA implementations provide many advantages over conventional hardware implementations:



Helion LZRW3 Data Compression Core for Xilinx FPGA

Features:

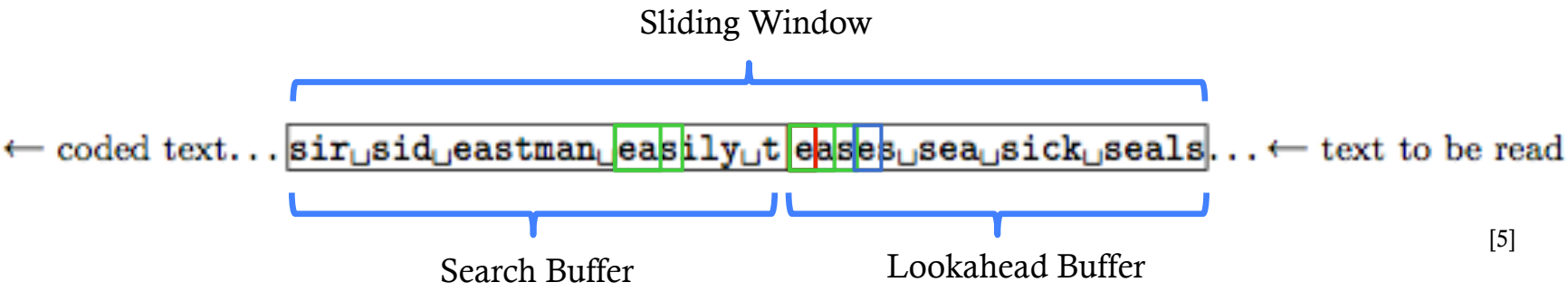
- Available as Compress only, Expand only, or Compressor/Expander core
- Supports data block sizes from 2K to 32K bytes with data growth protection
- Completely self-contained; does not require off-chip memory
- High performance; capable of data throughputs in excess of 1 Gbps
- Highly optimized for use in Xilinx FPGA technologies
- Ideal for improving system performance in data communications and storage applications [4]

Logic Utilisation and Performance

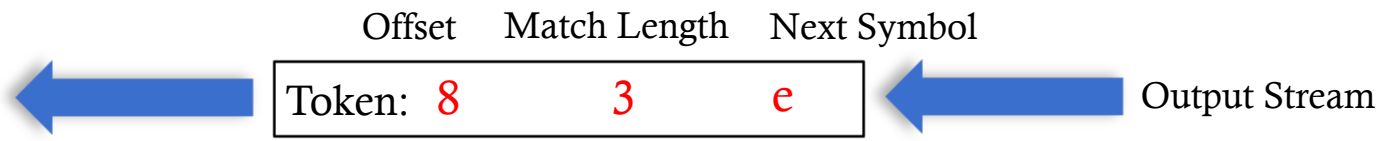
The tables below show typical logic area and performance figures for each version of the core covering two of the most popular Xilinx device families. All figures shown are for versions of the core supporting a maximum block size of 2K bytes.

	LZRW3 Comp		LZRW3 Exp		LZRW3 Comp/Exp	
technology	Spartan3E -5	Virtex5 -3	Spartan3E -5	Virtex5 -3	Spartan3E -5	Virtex5 -3
logic resource	787 slices 4 RAMB16	190 slices 4 RAMB18	794 slices 4 RAMB16	166 slices 4 RAMB18	885 slices 4 RAMB16	213 slices 4 RAMB18
max clock	103 MHz	215 MHz	110 MHz	226 MHz	106 MHz	205 MHz
typ throughput	659 Mbps	1376 Mbps	704 Mbps	1446 Mbps	678 Mbps	1312 Mbps

LZ77 Algorithm



[5]



- Match with smallest offset is always encoded if more than one match can be found in the search buffer.
- However, if no match can be found, a token with zero offset and zero match length and the unmatched symbol are written.

Improvements on LZ77 Algorithm

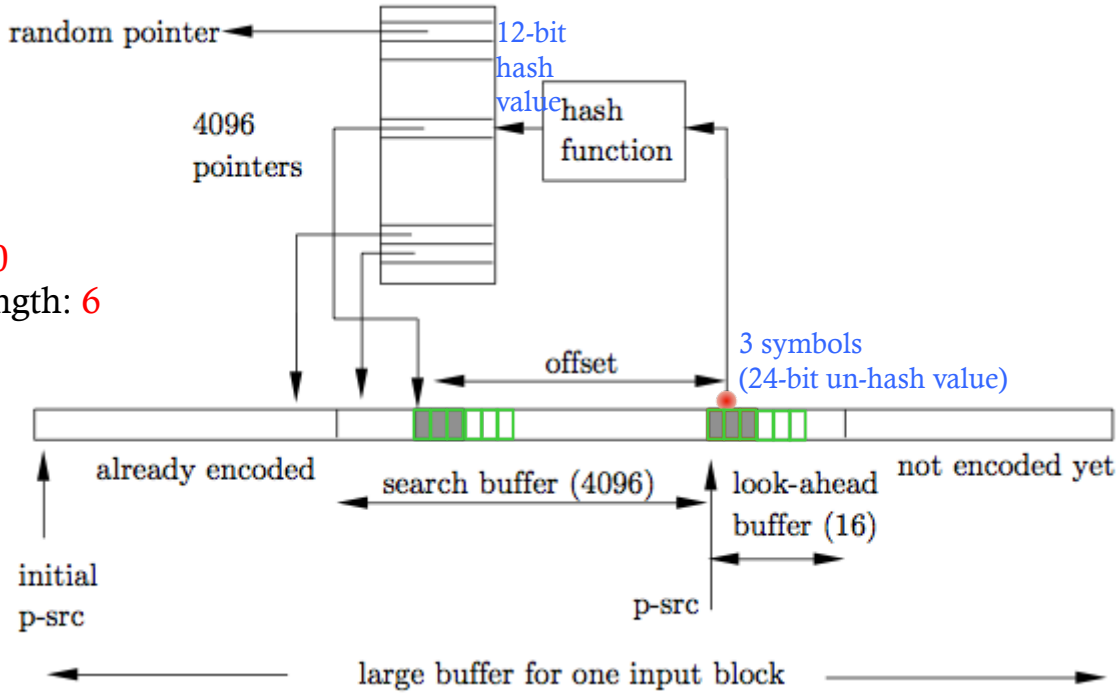
- Encode the token with variable-length codes rather than fixed-length codes, e.g. LZX uses static canonical Huffman trees to provide variable-size, prefix codes for the token.
- Vary the size of the search and look-ahead buffers to some extent.
- Use more effective match-search strategies, e.g. LZSS establishes the search buffer in a binary search tree to speed up the search.
- Compress the redundant token (offset, match length, next symbol), e.g. LZRW1 adds a flag bit to indicate if what follows is the control word (including offset and match length) for a match or just a single unmatched symbol.

Disadvantages of LZ77 and its Variants

- If a word occurs often but is uniformly distributed throughout the text. When this word is shifted into the look- ahead buffer, its previous occurrence may have already been shifted out of the search buffer, then no match can be found even though this word has appeared before.
- A big trade-off: size L of the look-ahead buffer. Longer matches would be possible and also compression could be improved if L is bigger, but the encoder would run much more slowly when searching and comparing for longer matches.
- A big trade-off: size S of the search buffer. A large search buffer results in better compression because more matches may be found, but it slows down the encoder, because searching takes longer time.

LZRW1 Algorithm

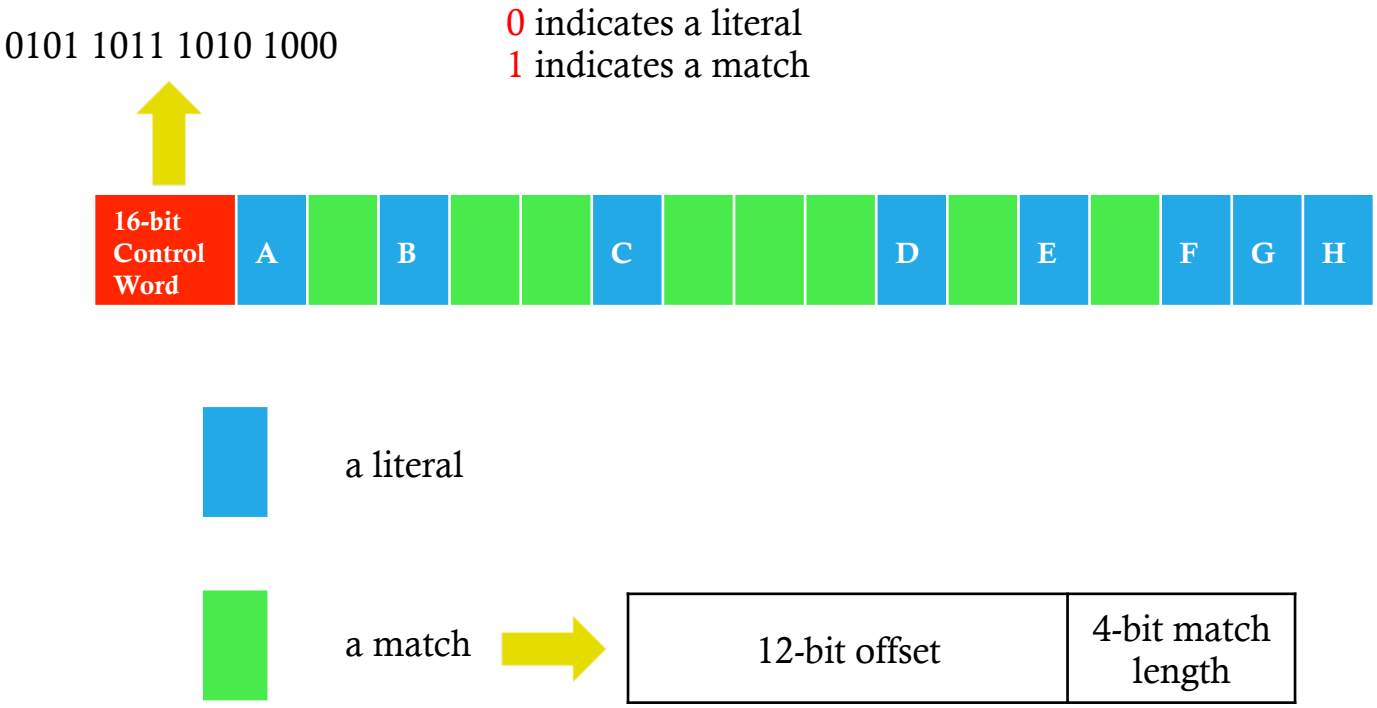
e.g. Offset: 100
Match Length: 6



[5]

- On the basis of LZ77, a hash table is used to help search a match faster. However it is fast but not very efficient, since the match found is not always the longest.

Output Format of LZRW1 Encoder

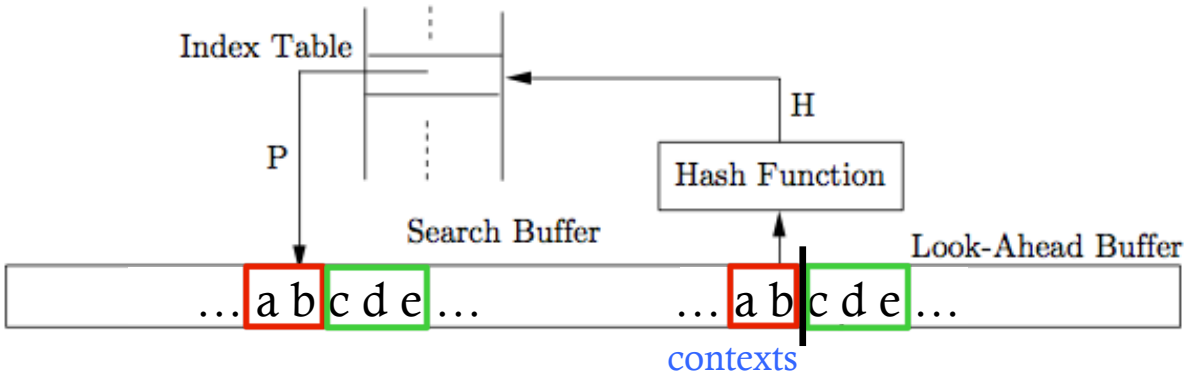


- Obviously, groups have different lengths. The last group may contain fewer than 16 items.

Disadvantage of LZRW1

- Ideally, the hash function will assign each different index to a unique phrase, but this situation is rarely achievable in practice. Usually some even different phrases could be hashed into the index, which is called **Hash Collision**. Therefore, LZRW1 has such drawback that the use of hash table can lead to a little worse compression ratio because of lost matches, even though hash table turns out to be more efficient than search trees or any other table lookup structure.

LZP Algorithm

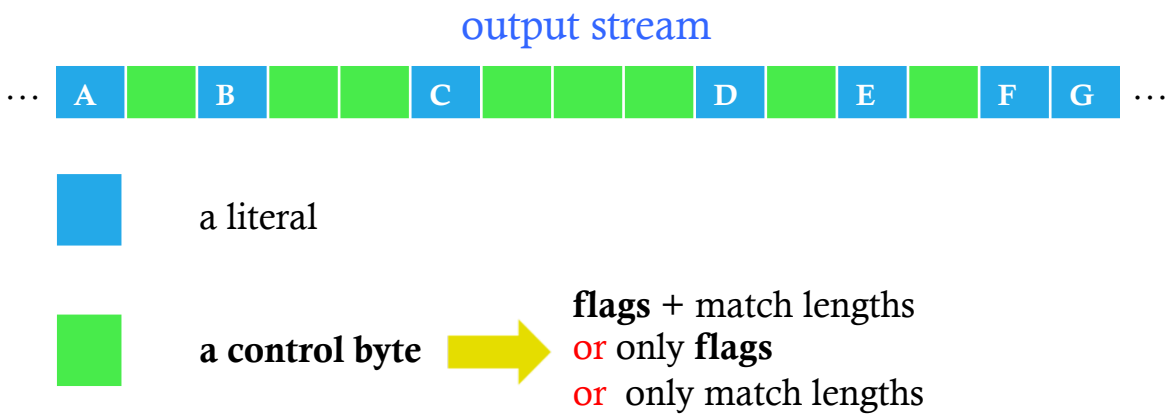


[5]

Match Length: 3

Output Format of LZP1

- LZP has 4 versions, called LZP1 through LZP4, where LZP1 is mainly used in my implementation.



- An "average" input stream usually results in more literals than match length values, so it makes sense to assign **a short flag (less than one bit) to indicate a literal**, and **a long flag (a little longer than one bit) to indicate a match length.**

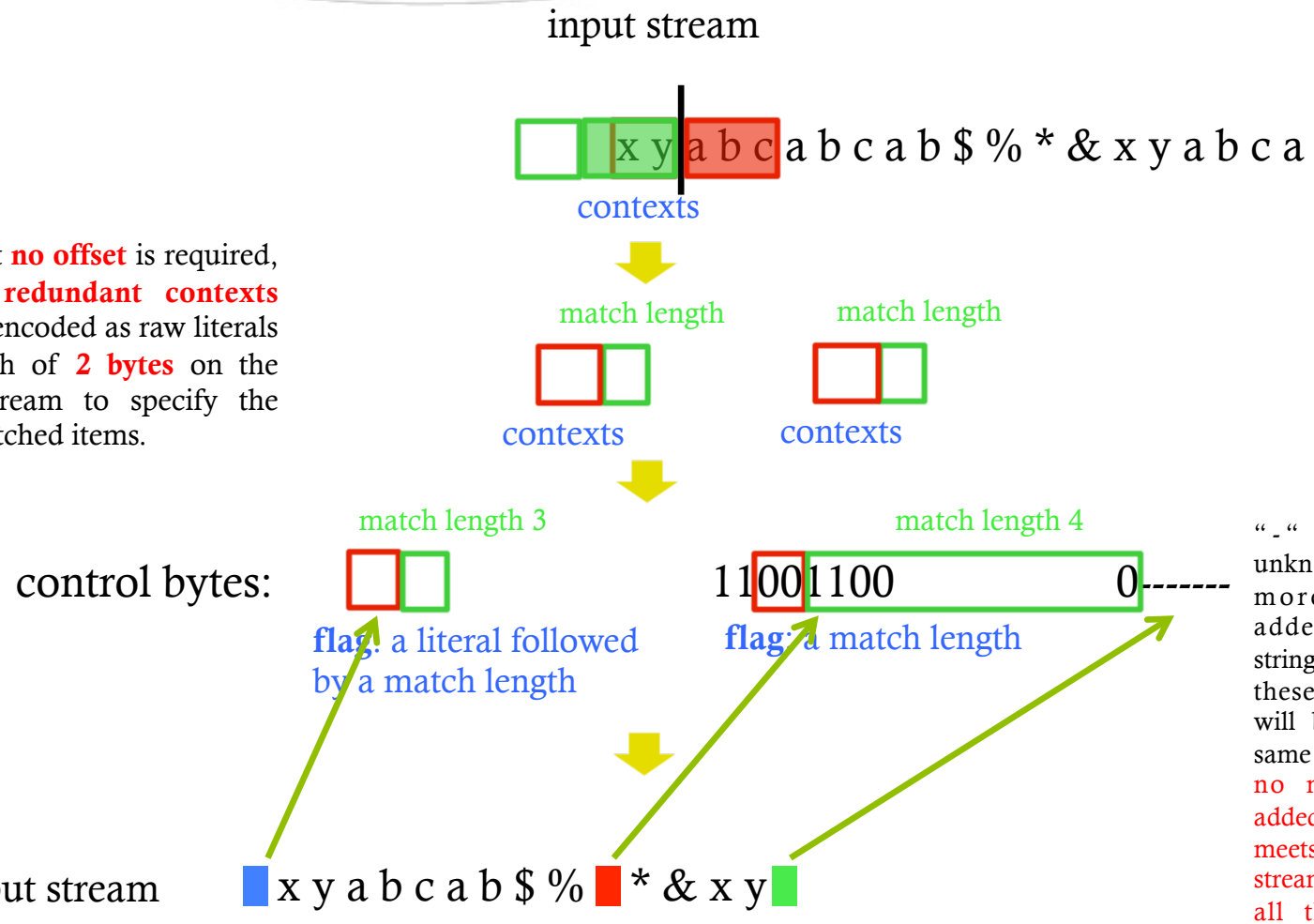
flag "1"	two consecutive literals
flag "01"	a literal followed by a match length
flag "00"	a match length

Output Format of LZP1

- The scheme of encoding match lengths is shown below. The codes are 2 bits initially. When these 2 bits are all used up ("11"), 3 bits are added. When these are also all used up ("11111"), 5 bits are added. From then on another group of 8 bits is added when all the old codes have been used up.

Length	Code	Length	Code
1	00	11	11|111|00000
2	01	12	11|111|00001
3	10	:	:
4	11|000	41	11|111|11110
5	11|001	42	11|111|11111|00000000
6	11|010	:	:
:	:	296	11|111|11111|11111110
10	11|110	297	11|111|11111|11111111|00000000

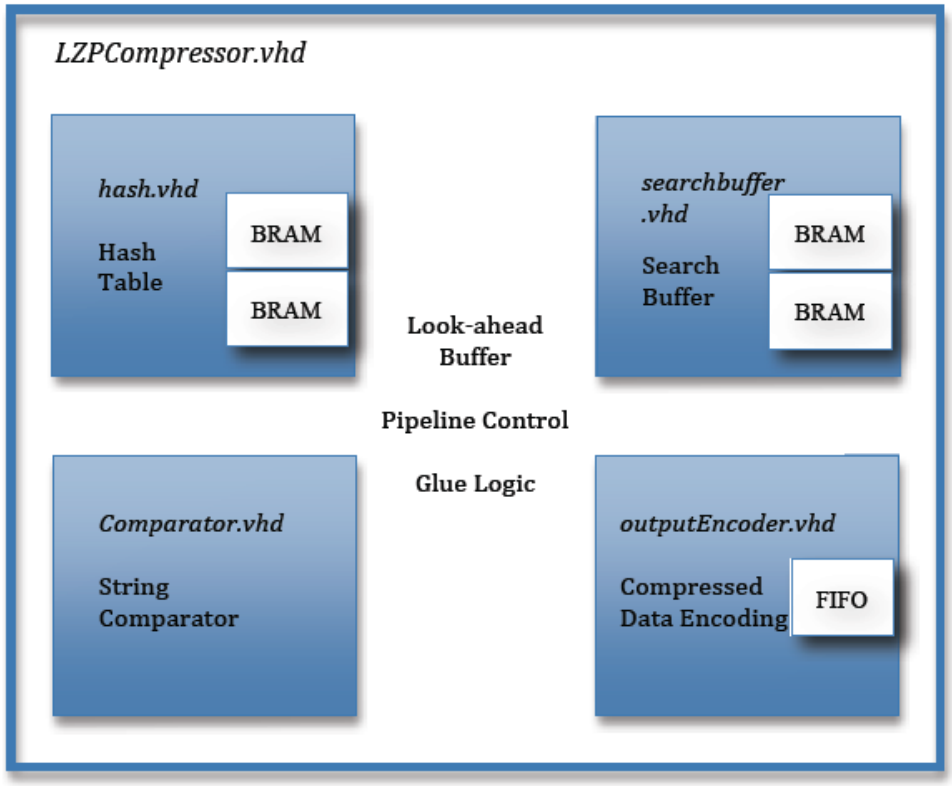
Example



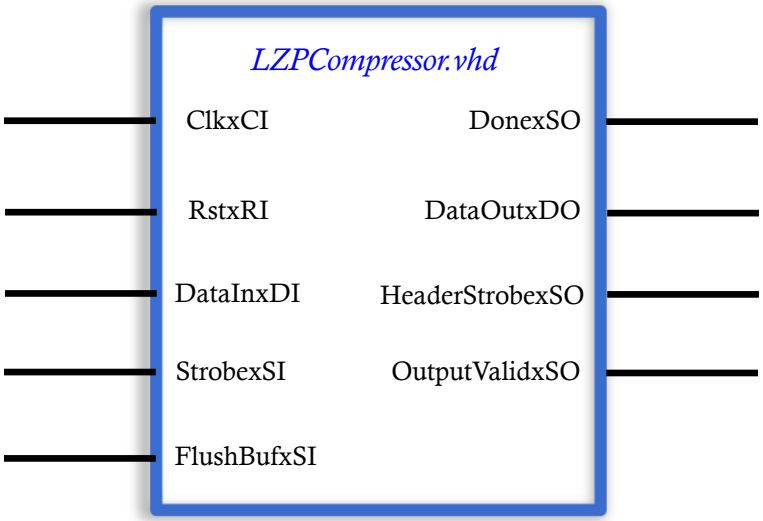
!!It is true that **no offset** is required, however the **redundant contexts** **"ab"** must be encoded as raw literals with the length of **2 bytes** on the compressed stream to specify the position of matched items.

"-" indicates an unknown flag bit. If more symbols are added to the input string in the following, these unknown flags will be written in the same way as before. If no more symbol is added, i.e. the encoder meets the end of input stream, in my design, all the "-"s will be replaced by "1".

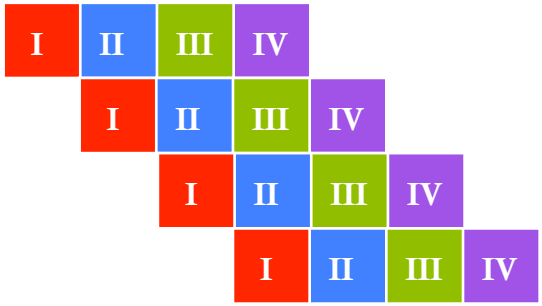
LZP Compressor in FPGA Implementation



Top-Level Module



- 16-byte look-ahead buffer, 4-stage pipeline operations, some glue logic



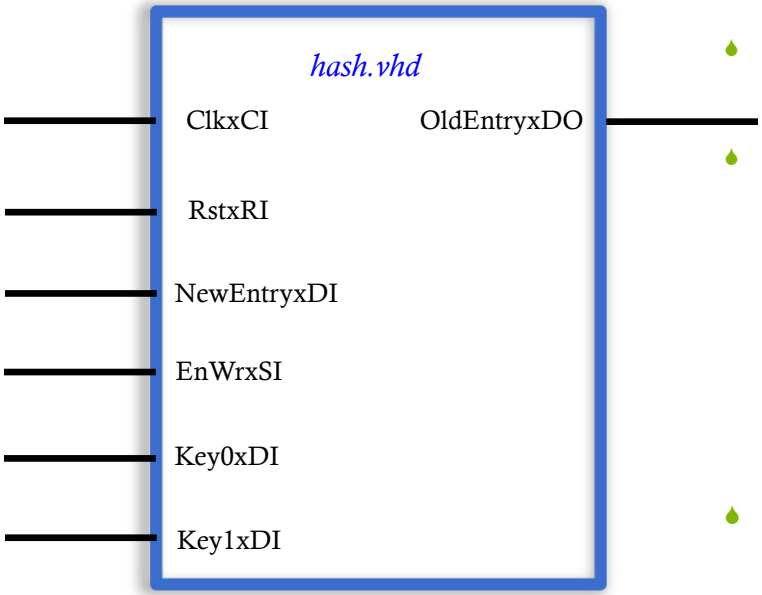
I contains:
FSM controlling look-ahead buffer to start reading input stream until all input data is shifted out;
A actual shift register for look-ahead buffer with the variation of its effective length;
Storing input stream in search buffer;
Calculating address of pointer to the beginning of look-ahead buffer and storing it into a certain position in hash table;
Counting the number of processed bytes in search buffer.

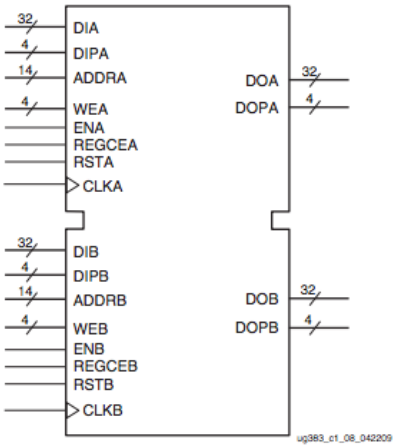
II waits for the read-back data from search buffer.

III attains a candidate string of 16 bytes from the search buffer according to the search pointer stored in the hash table.

IV compares this candidate string of 16 bytes with the 16 bytes data in the look-ahead buffer to calculate the match length…

Hash Table Module





- Two 2 KB Xilinx Block RAMs with RAMB16BWER configuration are used to implement the 4 KB long hash table.
- Two-byte-wide key is hashed using the hash function written in VHDL as follow:

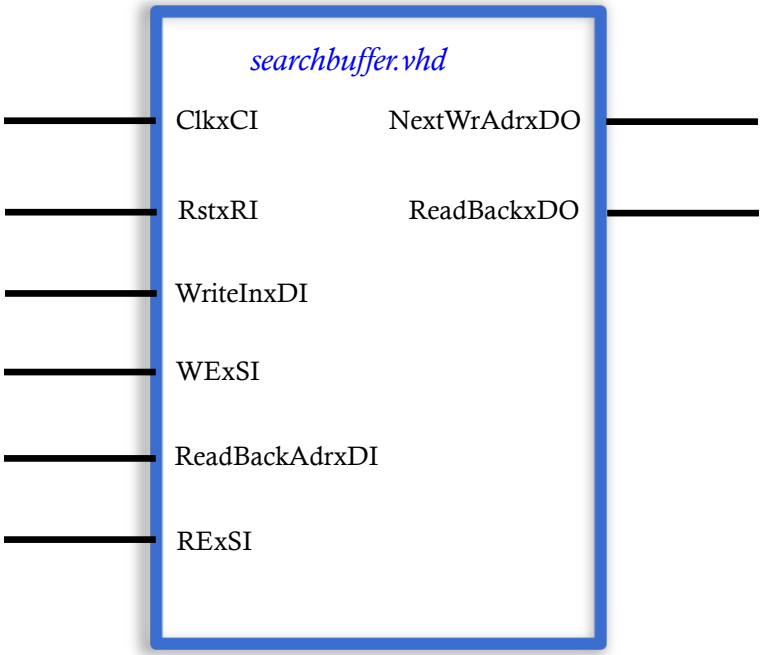
```
Stage0xS <= Key1xDI(7 downto 4) & (Key1xDI(3 downto 0) xor Key0xDI(7 downto 4)) & Key0xDI(3 downto 0) & Key0xDI(7 downto 4);

Stage1xS <= Key0xDI & (Key0xDI(3 downto 0) xor Key1xDI(7 downto 4)) & Key1xDI(3 downto 0);

Stage2xS <= ('0' & Stage0xS(15 downto 1)) xor (Stage1xS(14 downto 0) & '0');

BRamAddrxD <= Stage2xS(13 downto 0);
```

- The second hash function is a modified method based on the first one. It can reduce some more hash collisions, which leads to better compression.

```
constant SEED : integer := 40543;

Stage0xS <= Key0xDI(7 downto 4) & (Key0xDI(3 downto 0) xor Key1xDI(7 downto 4)) & Key1xDI(3 downto 0);

Stage1xS <= Stage0xS(11 downto 4) & (Stage0xS(3 downto 0) xor Key1xDI(7 downto 4)) & Key1xDI(3 downto 0);

ProductxS <= SEED * to_integer(unsigned(Stage1xS));

RawHashxS <= std_logic_vector(to_unsigned(ProductxS, 32));

BRamAddrxD <= RawHashxS(13 downto 0);
```

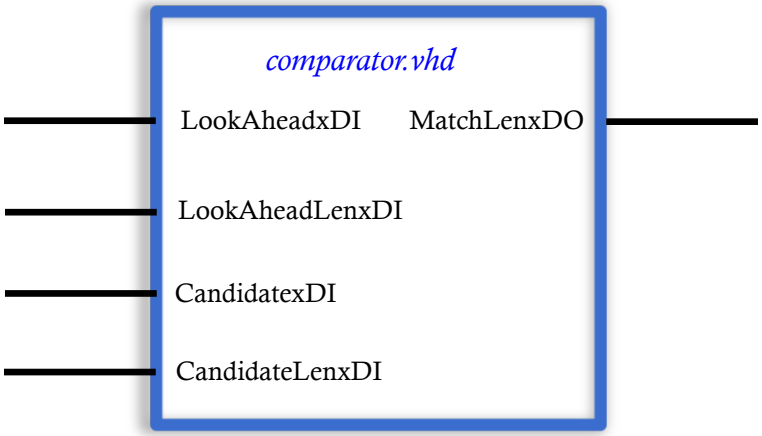

RAMB16BWER

Search Buffer Module



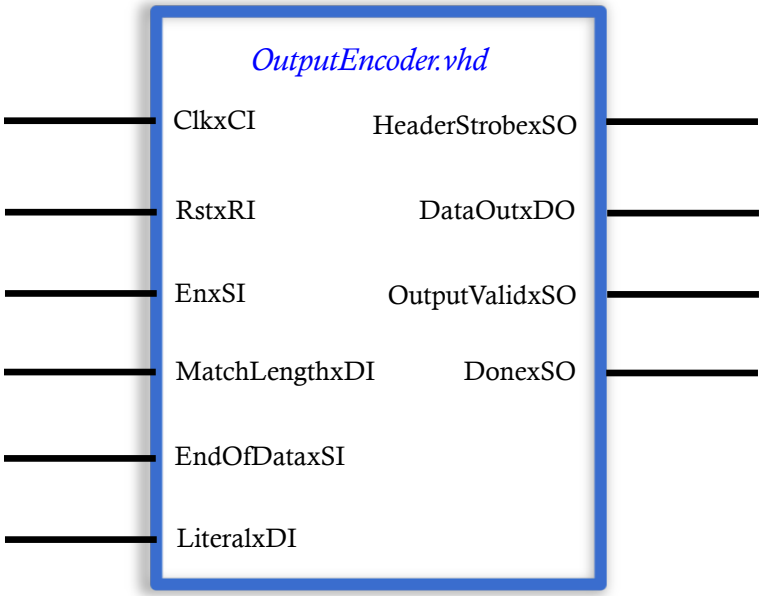
- It is implemented with two 2 KB Xilinx Block RAMs. It can store the input stream and also read back the candidate string for match length checking.

Comparator Module



- Once a candidate has been loaded from the search buffer, this unit compares it to the current look-ahead buffer and determines how many bytes match.
- For convenience of implementation, the first 2 symbols in the look-ahead buffer are treated as the contexts.
- When a match is found, the comparator firstly checks if the contexts are the same to avoid hash collision, then compares the following 14 symbols.
- So actually the maximum match length is allowed to be 14. Due to utilization of 2 Block RAMs in the search buffer, the length of the read-back candidate string is restricted to 16.

Output Encoder Module



- The output encoder encodes the literals and match items with control bytes, and writes them in serialized form in a simple synchronous circular FIFO that is 20 bytes long.
- Once a frame including a control byte followed by a group of literals is finished, FIFO starts to read out the encoded data in this frame. If read pointer overlaps with write pointer at same position, FIFO can stop reading out for a moment.
- At the first try, the simple 4-bit fixed-size method was used to encode match length. At the second try, the similiar variable-size method of LZP1 was used in order to improve compression ratio, but the maximum match length is restricted to 14.

Length	Code	Length	Code
1	00	11	11|111|00
2	01	12	11|111|01
3	10	13	11|111|10
4	11|000	14	11|111|11
5	11|001		
6	11|010		
:	:		
10	11|110		

Performance Evaluation

- This specific instance of comparisons among LZRW1 and four implementations of LZP with different match length encoding schemes and different hash functions is made in condition of the same input stream to be read in containing randomly chosen and redundant contents with the size of 9189 Bytes.

	LZP (variable-size match length)		LZP (fixed-size match length)		LZRW1
	Inferior Hash Function	Improved Hash Function	Inferior Hash Function	Improved Hash Function	
Size of Uncompressed Input Stream	9189 Bytes				
Size of Compressed Stream	7333 Bytes	7311Bytes	7527 Bytes	7505 Bytes	5639 Bytes
Compression Ratio	79.80%	79.56%	81.91%	81.67%	61.37%
No. of Matches Found	1440	1499	1440	1499	1628
No. of Clock Cycles required by Execution (estimated by simulation)	18436		18428		18433
Minimum Time Period (estimated by synthesis)	15.748 ns		14.679 ns		13.022 ns
Maximum Frequency (estimated by synthesis)	63.5 MHz		68.1 MHz		76.8 MHz
FPGA Resource Utilization Ratio (estimated by synthesis)	16 %		7 %		6 %
Compression Speed	31.65 MB/s		33.97 MB/s		38.28 MB/s

- Compression Ratio = Size of Compressed Stream / Size of Input Stream
- Compression Speed = Size of Uncompressed Input Stream / (No. of Time Clocks required by Execution × Minimum Time Period)

Performance Evaluation

- Two kinds of corpus commonly used benchmarks:
- **Calgary Corpus** (a set of 18 files including text, image, and object files with totally more than 3.2 million bytes)
- **Canterbury Corpus** (another collection of files based on concerns about how representative the Calgary corpus is)

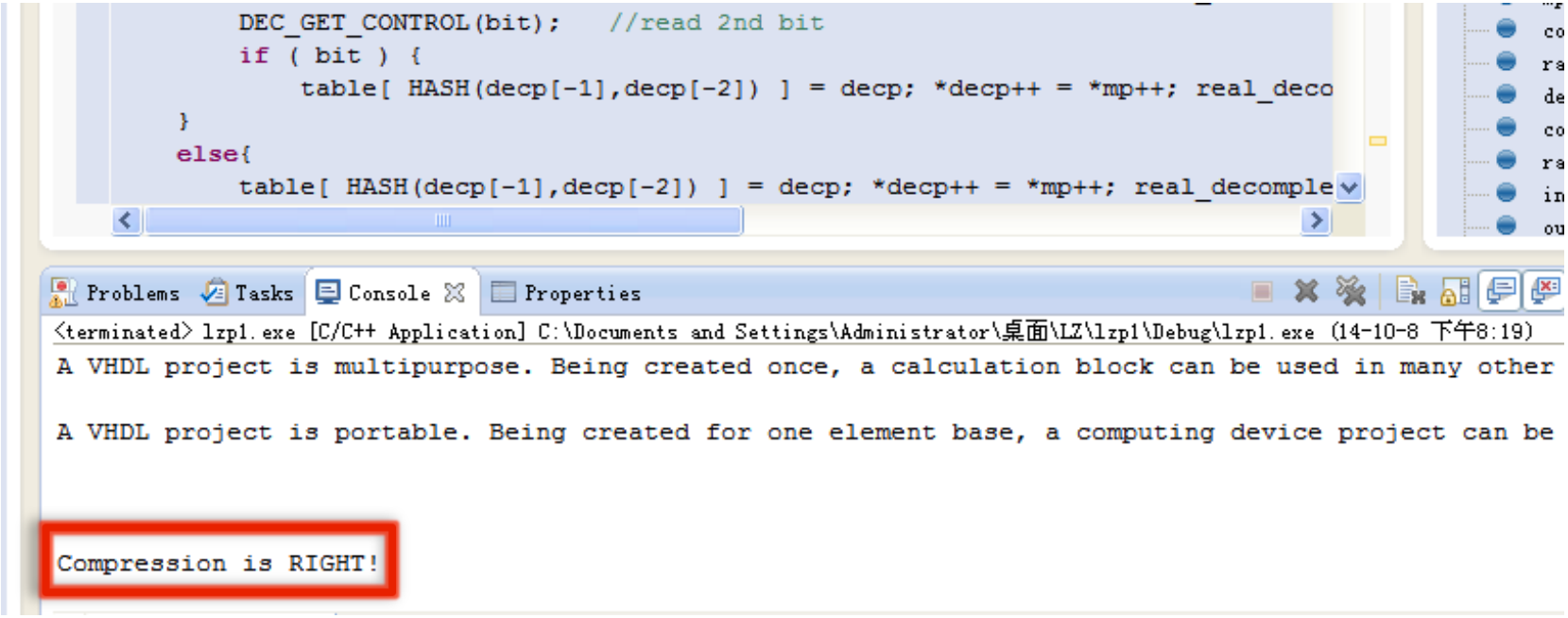
	LZP (variable-size match length, improved hash function)	LZRW1
alice29.txt	80.98%	61.20%
fields.c	59.14%	44.96%
lcet10.txt	79.30%	59.06%
plrabn12.txt	88.40%	67.76%
cp.html	67.07%	52.47%
grammar.lsp	59.71%	49.05%
xargs.1	72.96%	57.96%
asyoulik.txt	81.65%	63.07%

Compression ratios comparisons between LZRW1 and LZP tested by Canterbury Corpus

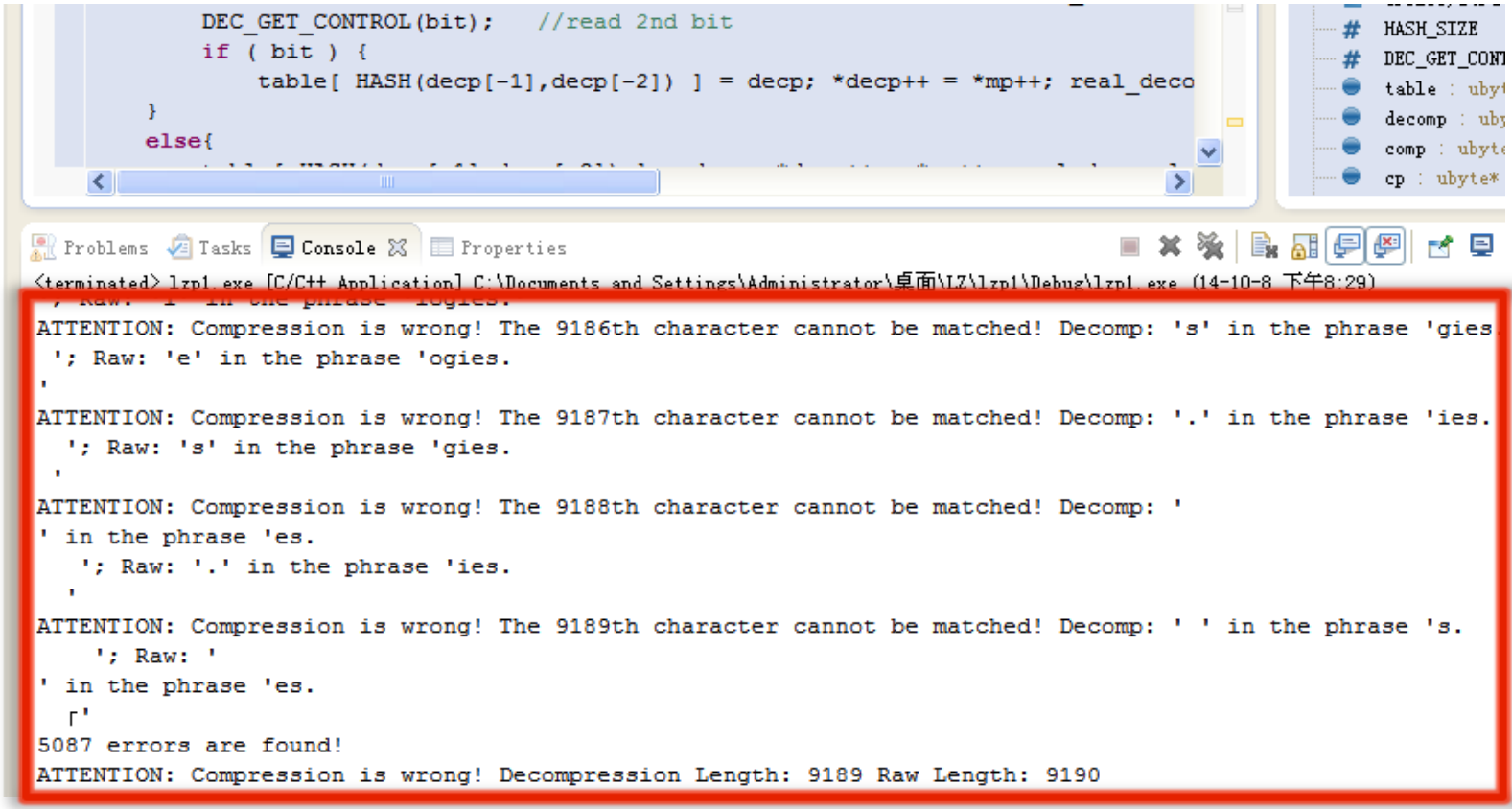
LZP Decompressor

- The LZP decompressor is implemented in C language. It can not only decompress the compressed stream, but also verify the correctness of compressed stream through comparing the decompressed stream with the original input stream. After many times and kinds of tests, the functional correctness of the LZP compressor can be guaranteed.

- Furthermore, compared to the LZRW1 decoder, actually LZP decoder is more complex because a hash table is a must to also know the position nformation by hashing contexts.



LZP Decompressor



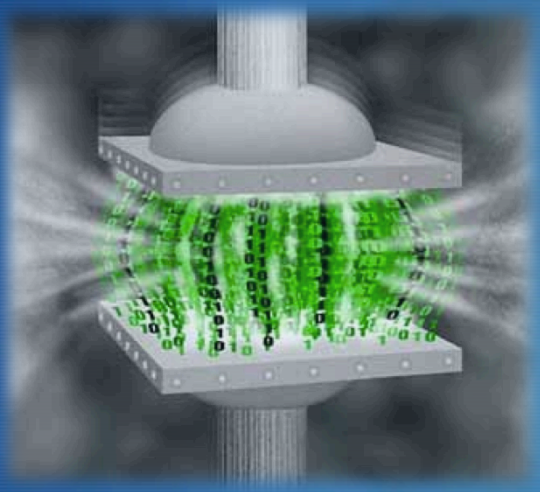
Conclusions and Future Work

- The LZP algorithm has good intention to improve the state-of-the-art compression algorithm. The good strategies of encoding control flags and match lengths should be affirmed. However, it pays more cost to replace the offset by the context, which results in worse compression performance in fact. So it is a regret for my work that the goal of optimizing compression has not been realized.

- In future work, it is very necessary to be more cautious to select or to optimize an algorithm for implementation. And some other optimization techniques should be possibly considered, for example, more than one compressor process different input blocks divided from a same input stream in parallel.

References

[1] http://en.wikipedia.org/wiki/Morse_code
[2] https://www.scribd.com/doc/190107945/20-Compression
[3] http://www.ieeeghn.org/wiki/index.php/History_of_Lossless_Data_Compression_Algorithms
[4] http://www.heliontech.com/downloads/lzrw3_xilinx_datasheet.pdf#view=Fit
[5] David Salomon, G. Motta, D. Bryant, "Data Compression: The Complete Reference", Springer Science & Business Media, Mar 20, 2007.



Thank you!