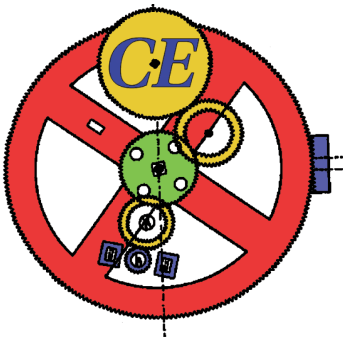

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2018

MSc THESIS

An FPGA-based Snappy Decompressor-Filter

Yang Qiao

CE-MS-2018-04

Abstract

New interfaces to interconnect CPUs and accelerators at memory-class bandwidth pose new opportunities and challenges for the design of accelerators. This thesis studies one such accelerator, a decompressor for Parquet files compressed with the Snappy library. Our design targets reconfigurable logic (FPGAs) attached via the open coherent accelerator processor interface(OpenCAPI) at 25.6GB/s. We give an overview of the previous research in hardware-based (de)compression engines and present and analyze our design. Much of the challenge of designing the decompression engine stems from the need to process more than one token per cycle. In our design, a single engine can process two tokens per cycle. A Xilinx KU15P FPGA is expected to support multiple such engines. The input throughput and the output throughput ranges of a single engine are 3.9∼6.3 bytes/cycle and 8.3∼15 bytes/cycle, respectively. Based on the implementation results, a single engine of the proposed design could work at 140MHz, meaning 0.51∼0.82 GB/s input throughput or 1.08∼1.96 GB/s output throughput. The Parquet format enables the parallel decompression of multiple blocks when multiple units are instantiated. With the latest generation of FPGAs, we estimate at most 28 units can be supported leading to a total input/output bandwidth of 14.28/30.24 to 22.96/54.88 GB/s. Because the output bandwidth can exceed the interface bandwidth if multiple engines are supported, the design is especially effective when combined with a filter engine that reduces the output size.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

An FPGA-based Snappy Decompressor-Filter

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

by

Yang Qiao
born in Jinan, China

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

An FPGA-based Snappy Decompressor-Filter

by Yang Qiao

Abstract

New interfaces to interconnect CPUs and accelerators at memory-class bandwidth pose new opportunities and challenges for the design of accelerators. This thesis studies one such accelerator, a decompressor for Parquet files compressed with the Snappy library. Our design targets reconfigurable logic (FPGAs) attached via the open coherent accelerator processor interface(OpenCAPI) at 25.6GB/s. We give an overview of the previous research in hardware-based (de)compression engines and present and analyze our design. Much of the challenge of designing the decompression engine stems from the need to process more than one token per cycle. In our design, a single engine can process two tokens per cycle. A Xilinx KU15P FPGA is expected to support multiple such engines. The input throughput and the output throughput ranges of a single engine are 3.9∼6.3 bytes/cycle and 8.3∼15 bytes/cycle, respectively. Based on the implementation results, a single engine of the proposed design could work at 140MHz, meaning 0.51∼0.82 GB/s input throughput or 1.08∼1.96 GB/s output throughput. The Parquet format enables the parallel decompression of multiple blocks when multiple units are instantiated. With the latest generation of FPGAs, we estimate at most 28 units can be supported leading to a total input/output bandwidth of 14.28/30.24 to 22.96/54.88 GB/s. Because the output bandwidth can exceed the interface bandwidth if multiple engines are supported, the design is especially effective when combined with a filter engine that reduces the output size.

Laboratory : Computer Engineering
Codenumber : CE-MS-2018-04

Committee Members :

Advisor:	Prof. Dr. H.P. Hofstee, CE, TU Delft
Chairperson:	Prof. Dr. H.P. Hofstee, CE, TU Delft
Member:	Dr. Ir. Z. Al-Ars, CE, TU Delft
Member:	Dr. Ir. A. Bossche, EI, TU Delft

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

BRAM Block random access memory

FPGA Field programmable gate array

AFU Accelerator functional unit

OpenCAPI Open coherent accelerator processor interface

SDP Simple dual port

TDP True dual port

LUT Look-up table

FIFO First in, first out

CRC Cyclic redundancy check

CLB Configurable logic block

FF Flip-flop

SIMD Single instruction multiple data

MRR Multi-round resolution

HWM High-water mark

Acknowledgements

Firstly, I want to express my sincere gratitude to my supervisor, Prof.Dr.H.Peter Hofstee, he had spent much time on my thesis project and gave me much constructive advice. He is very patient with me even if I do not have too much fundamental knowledge. Undoubtedly, I have learned a lot during this year. Thank you, Peter!

Secondly, I would thank my committee members for spending much time on my thesis, giving me feedback, and attending my thesis defense even if they are very busy.

Thirdly, I need to thank my daily supervisor, Jian Fang, who introduced me to this group and taught me much useful knowledge. He is a Ph.D. student now, so I hope he could gain the Ph.D. degree as soon as possible and make more contribution to academic field.

Many thanks to Jinho Lee and Yvo Mulder who spend time helping me the project.

I am also grateful to all my friends at TU Delft; I would not have such a happy life in the Netherlands without your company.

Last but most important, I sincerely thank my family, especially my parents, for giving me the chance to study at TU Delft and supporting me whenever I was in trouble. Thanks!

Yang Qiao
Delft, The Netherlands
January 23, 2018

1 Introduction

With the development of silicon technology, the integration level has improved dramatically. However, in recent years, the (power-limited) switching frequency of transistors has saturated, which indeed constrains the processing speed of some applications, especially, in Big Data.

1.1 Research background

With the development of Big Data, data volume is rapidly increasing, so data compression that can substantially reduce physical storage requirements is important due to the high cost of storage. However, improvements in CPU performance do not keep pace with the fast increases in data volumes. The compressed data has to be decompressed before doing operations on it. Thus, a fast decompression or a high decompression throughput is necessary. Therefore, much research focuses on hardware accelerators to improve processing rates.

Much research has focused on how to achieve efficient compression, a high compression ratio, and fast compression. Decompression speed is also important because data is compressed only once when being stored but is repeatedly read and decompressed. Especially in the era of Big Data, repeated decompression requires very high throughput [3]. With many data warehousing applications moving from spinning disk to in-memory, there is a challenge to operate at memory bandwidths rather than storage bandwidths for decompression.

1.2 Problems

There is some research concentrating on a hardware decompression accelerator, either FPGA or GPU. [3] proposed a GPU-based method called "Gompresso/Byte" that achieves a relatively high decompression speed. If not considering the bandwidth of PCIe, its decompression throughput was up to 16GB/s. However, if taking PCIe(both input and output) into account, it can only achieve the speed up to about 10GB/s. Therefore, decompression can be an I/O-bound operation.

1.3 Motivation and objective

From a traditional point of view, PCIe is one of the fastest interfaces. However, recently, a new class of interface has been proposed that significantly improves over PCIe on bandwidth and latency. One such interface is the open coherent accelerator processor

interface (OpenCAPI), which can achieve high bandwidth up to 25.6GB/s (8 lanes, bi-directional). Future versions of this interface will support 32 lanes and even higher frequencies per lane [8]. Therefore, these new interfaces provide a chance to improve the decompression speed further [9]. Most prior work focuses on hardware decompression based on Gzip [10], but there is little prior work focusing on hardware (de)compression on Snappy [11, 6].

The goal of this thesis is to maximize Snappy decompression rates on an FPGA attached to OpenCAPI. Optimizing the use of FPGA BRAM resources leads us to implement a hardware Snappy decompression accelerator that can process at most two tokens per cycle.

1.4 Contributions

This thesis gives an overview of prior work on (de)compression acceleration. We also propose an FPGA-based hardware architecture of a Snappy decompression engine. A single decompression engine is designed to process at most two tokens per cycle. The correct uncompressed files are obtained via behavior simulation. The expected input and output throughput of a single engine reach up to 0.82 GB/s and 1.96 GB/s. The working frequency (via implementation) of this architecture is 140MHz. A single engine consumes 91089, namely, 17.43% look-up table (LUT) resource and 128KB Block RAM resource of Xilinx KU15P FPGA.

1.5 Outline

Chapter 2 introduces the background knowledge and previous work on hardware acceleration of (de)compression. Chapter 3 presents an analysis of the Snappy compressed test files and motivates some design choices. Chapter 4 provides details of the implementation of our proposed hardware architecture and explains the function of each of the modules that make up the design. Chapter 5 gives the experimental results of this project and evaluates the performance.

2 Background and related work

This chapter motivates the use of accelerators and provides background on OpenCAPI, the Parquet file format, the Snappy algorithm and related work.

2.1 Accelerators

CPUs are optimized for the execution of general-purpose sequential programs. However, when a high performance or a high-speed computation is required, a CPU shows the drawback of being optimized for low degrees of parallelism.

Hardware acceleration is the use of computer hardware to perform some functions more efficiently than is possible in software running on a more general-purpose CPU [12]. Both FPGAs and GPUs can be used as accelerators of CPUs. However, FPGAs and GPUs are suitable for different applications due to different pros and cons.

2.1.1 FPGA

An FPGA is an application specific processor which supports higher degrees of parallelism than that of a CPU. An FPGA can be configured by a customer or a designer for a dedicated application.

2.1.2 GPU

A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. More recently the thread processing engines in GPUs have been generalized, allowing GPUs to be used for a variety of applications with high degrees of concurrency [13].

2.1.3 Comparison

Several aspects can be used to determine the choice. For example, power efficiency and cost efficiency which can be seen in Figure 2.1. Other qualitative comparisons among different characteristics are shown in Figure 2.2.

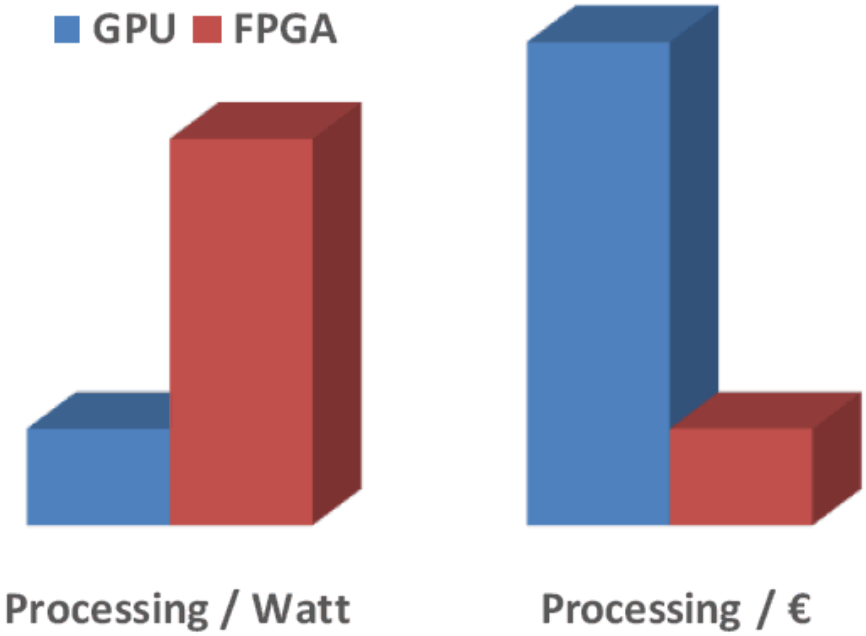

Figure 2.1: Processing efficiency [1]

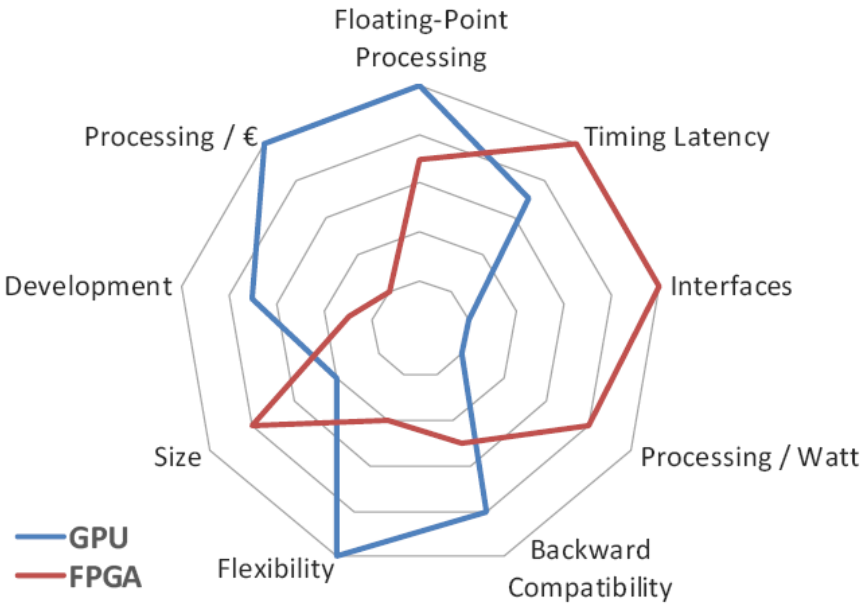

Figure 2.2: GPU vs FPGA qualitative comparison [1]

As can be seen from the Figure 2.1 and 2.2, FPGA is more power efficient while GPU is more cost efficient. Interfaces are another strong point for FPGAs. With GPUs typically limited to PCIe, interfacing with devices implementing any other standard or custom interfaces will require additional electronics. An FPGA has huge interface

flexibility, recently improved by the integration of programmable logic with CPUs and standard peripherals in SoC devices. Finally, RTL-based design enables an FPGA to be used as technology path to ASIC development. [1].

Also, in the era of Big Data, data center applications face a challenge of high power consumption of CPUs. FPGAs are power efficient and have huge interface flexibility. It would be a good choice to use FPGAs as accelerators to improve the performance and reduce the power consumption.

2.2 OpenCAPI

OpenCAPI is an open interface architecture that allows any microprocessor to attach to an accelerator or I/O device at high bandwidth and provide coherent access to shared memory [8]. The OpenCAPI Consortium is an open forum to manage the OpenCAPI specification and ecosystem. OpenCAPI was founded by Google, IBM, AMD and other companies. Some research works on the OpenCAPI protocol [14].

Traditional bus interfaces, such as PCIe, result in very high CPU overhead when applications communicate with I/O or accelerator devices at the necessary performance levels. Systems must be able to integrate multiple memory technologies with different access methods and performance attributes [8].

OpenCAPI can achieve a very high transfer rate of up to 25.6Gbps (single channel, bi-directional). Multi-channel is also supported by OpenCAPI. In conclusion, Open CAPI has four advantages compared to traditional I/O architecture, which is high performance, not occupying CPU resource, good compatibility and being a completely open consortium.

2.3 Parquet and columnar storage

Apache Parquet [15] is a columnar storage format; it is open-sourced and free to any project in the Hadoop ecosystem. With the development of Big Data, in many or even most cases, the number of fields (columns) is considerable, that is to say, a row tends to contain more data than before. However, most queries only focus on some of the records (rows). By storing data in columns rather than rows, the database can more precisely access the data it needs to answer a query rather than scanning and discarding unwanted data in rows. Query performance is often increased as a result, particularly in huge data sets [16]. Specifically, the columnar format has two advantages. The first one is that a subset of one column can be extracted when reading the data stored in the column, which significantly reduces the number of required I/O operations because for data stored in a row-based format the entire record must be read even if only a few of columns are needed. The second advantage is that it is easy to achieve a high compression ratio in a columnar storage file. Because all data within one column is of the same data type, and there are many methods of encoding, such as run length encoding (RLE), dictionary encoding, bit packing and plain encoding that effectively compress a column. Within the Parquet format definition, after encoding, the data can be further compressed, there are five options for Parquet which are Snappy, Gzip, LZO, Brotli and uncompressed.

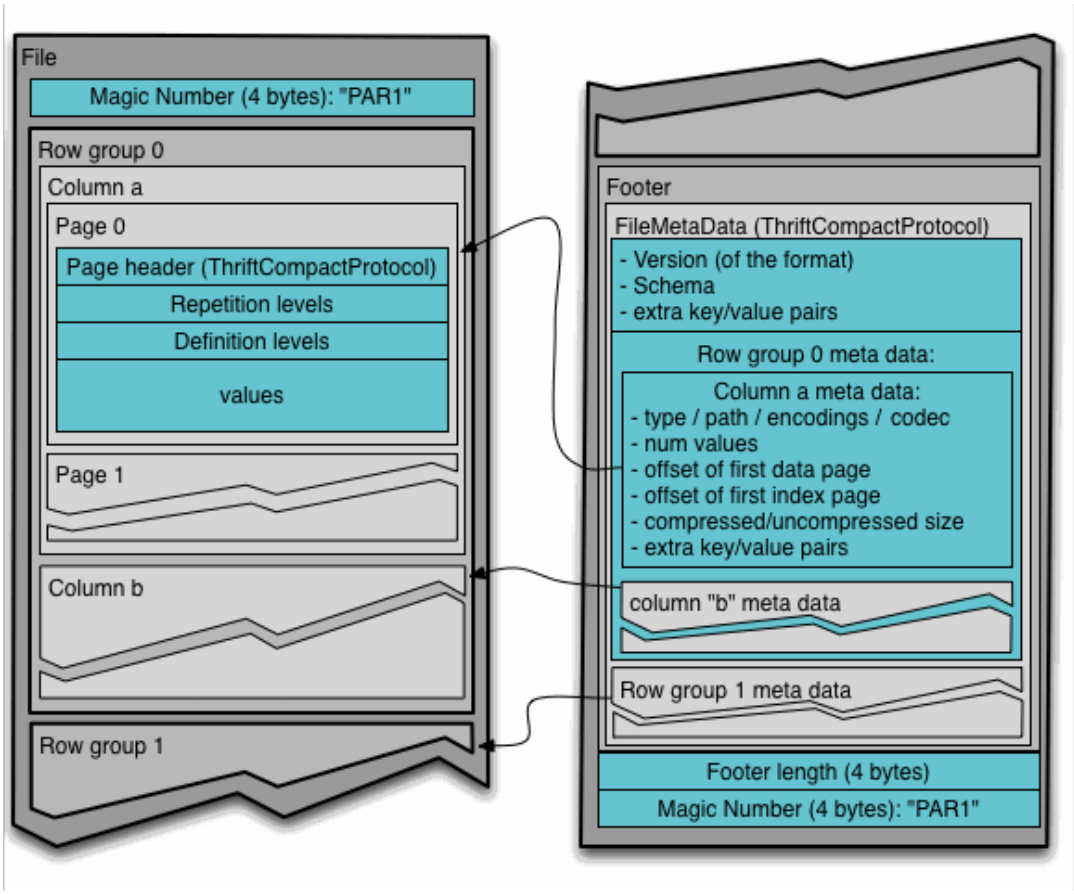

Figure 2.3: Parquet format [2]

A Parquet file is composed of a file header, one or more blocks (also called row groups), and a file footer. The header is a 4-byte magic number "PAR1" used to indicate Parquet file type. The footer contains the file metadata such as version information, schema, row group metadata and column metadata. At the end of the footer, there are 4 bytes of footer length and a 4-byte magic number "PAR1". Readers are expected first to read the file metadata to find all the column chunks they are interested. The columns chunks should then be read sequentially. The row group is several and fixed rows of data. One row group contains many column chunks. Each column in a row group is stored into one column chunk. One column chunk contains many pages. A page is the unit of encoding or compression. Parquet metadata is encoded according to Apache Thrift [17]. Figure 2.3 shows the Parquet format and detailed information about metadata can be seen in Figure 2.4.

Parquet adopts the Dremel encoding [18] with definition and repetition levels to encode nested data. According to Apache Thrift, there are three field repetition types, which are required(cannot be null), optional(can be null) and repeated. The field is required, and each record has exactly one value. The field is optional, and each record has 0 or 1 values. The field is repeated and can contain 0 or more values. Definition

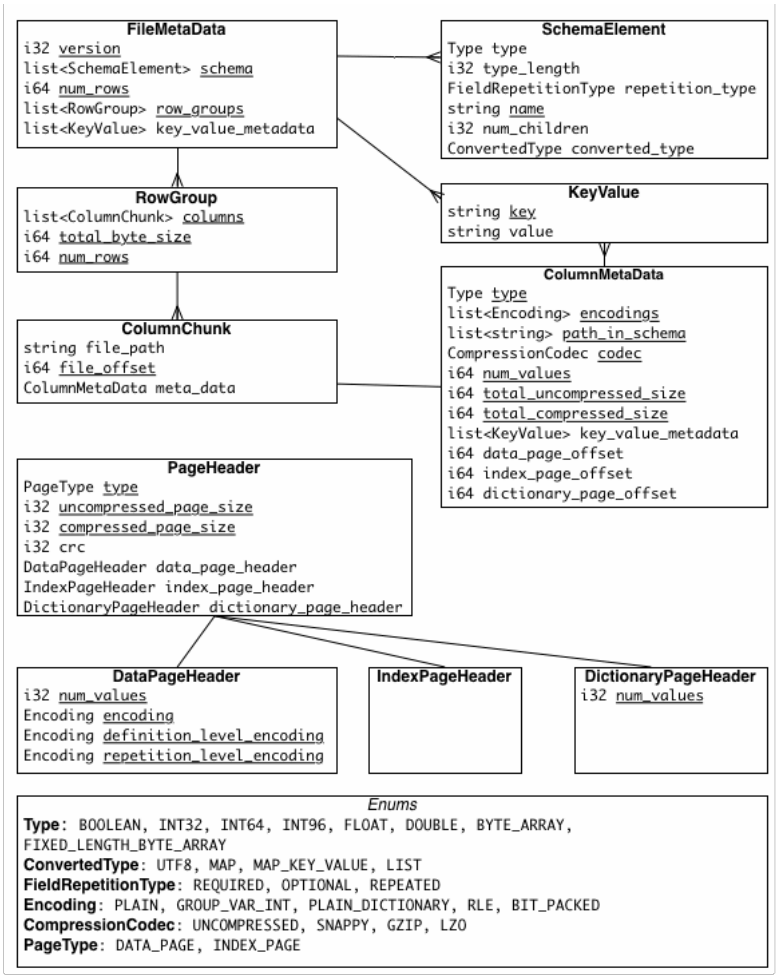

Figure 2.4: Parquet metadata [2]

levels specify how many optional fields in the path for the column are defined. Repetition levels specify at what repeated field in the path has the value repeated.

There are three types of pages, which are data page, index page(not supported by the current version) and dictionary page. A data page is composed of a repetition level, a definition level, and encoded values. The pages share a common header, and the pages can be skipped over if they are not interested in. The data for the page follows the header and can be compressed and/or encoded. The compression and encoding is specified in the page metadata. The compression dictionary is stored in dictionary page, and each column chunk could contain at most one dictionary page. The recommended row group size and page size are 512MB∼1GB and 8KB, respectively [15]. In addition, there is a "i32 crc", which is used to validate whether decompression is correct, in "PageHeader" in Figure 2.4.

2.4 Snappy algorithm

Snappy is a compression/decompression algorithm developed by Google based on LZ77 [19] and open-sourced in 2011. It is very fast but it compromises on the compression ratio. Snappy encoding is not bit-oriented with variable, data-dependent compressed symbols, but uses a fixed encoding and is byte-oriented, which means a byte is the unit of operation. On a single core of Core i7, its compression and decompression rate can reach up to 250MB/s and 500MB/s, respectively [20].

2.4.1 Snappy compression

During compression, the input file (uncompressed file) is no larger than 4GB and is cut into many equally-sized blocks, each of which is 64KB, except for the last block whose size is no larger than 64KB. Blocks are independent of each other, each block is compressed, and can, therefore, be decompressed, independently.

The size of the uncompressed file is stored as little-endian varint format into the first several bytes (no larger than 5 bytes) of the output file (compressed file), then these bytes are followed by compressed data blocks, the size of different compressed data blocks may not be the same. The simplified input and output file of Snappy compression is shown in Figure 2.5.

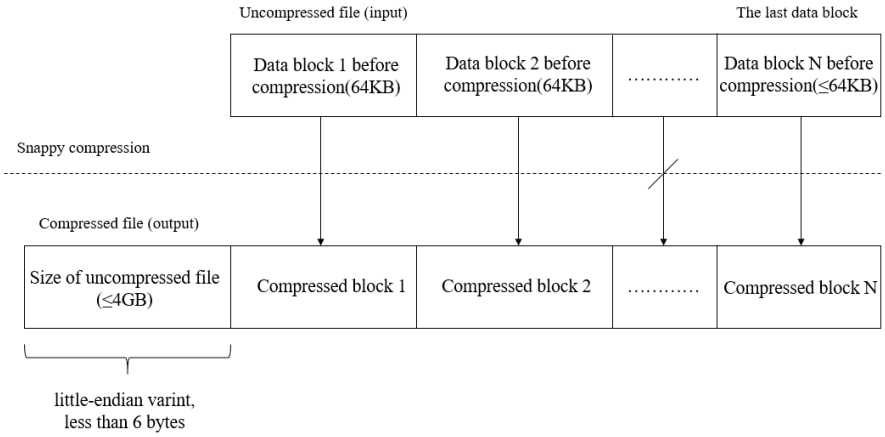

Figure 2.5: Input/output file of Snappy compression

Within each data block, the algorithm constantly finds matches. If it has found a 4-byte match, then it continues to find whether the match length can be longer than 4. Next, the compressor will output this match to the compressed file. Any match whose length is larger than 64 bytes will be cut into sub-matches and then output. Thus, the match length of each sub-match is no larger than 64 bytes. A match is also called a “copy”. If the compressor cannot find a match whose length is larger than 3 bytes within a specific range, then the content will be directly output as a literal.

Thus, there are two types of tokens; literal and copy. Literal means that the data is not compressed and the header (tag byte + extra bytes) is followed by several bytes of literal content. The copy means that the data is compressed as a pair of length and

offset. If "op" is the start output address of this copy, a copy operation needs to read "length" bytes from the output file starting from the address of "op - offset". Figure 2.6 and 2.7 indicate the structure of a token. The tag byte is always the start point of a token. For literals, the header is composed of a tag byte and extra bytes. While for copy, the header is the copy itself.

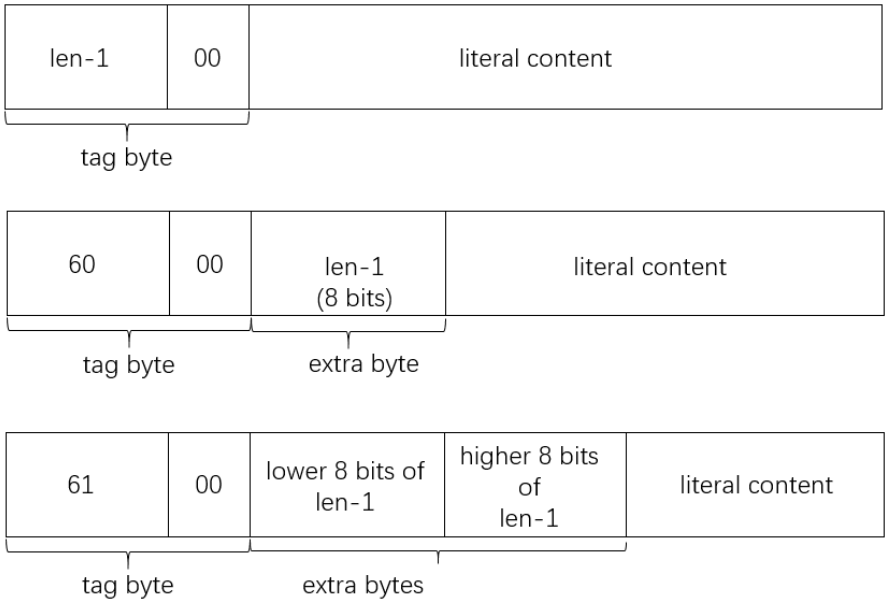

Figure 2.6: Structure of literal

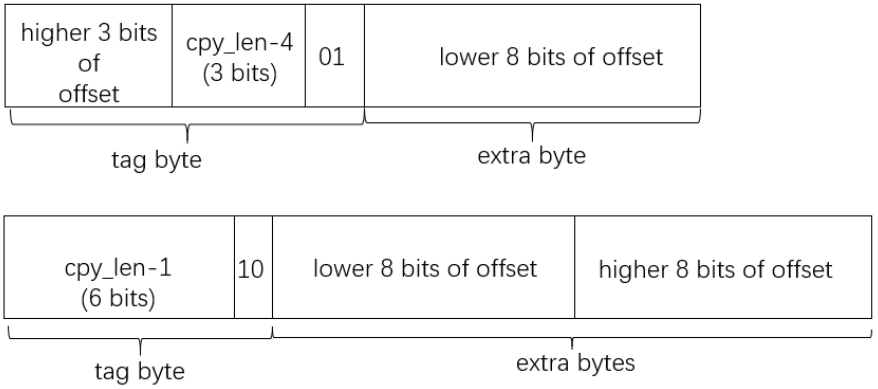

Figure 2.7: Structure of copy

As can be seen from Figure 2.6 and Figure 2.7, the lower two bits of tag byte indicate the token type, "00" represents literal and "01" or "10" represents copy. The extra bytes of literal represent the literal length, and those of copy indicate copy offset. Literal content means the uncompressed data itself.

For the literal type, as can be seen in Figure 2.6, when the higher six bits is less

than 60 (decimal system), the number of extra bytes is zero, and these six bits indicate "length-1". If the six bits have a value of 60 or 61, this indicates one extra byte or two extra bytes, respectively. The extra bytes are used to represent the value of "length-1".

Figure 2.7 shows the cases of the copy type. If the lower two bits are "01", then it means that the copy has one extra byte, the copy length is less than 12, and the offset is less than 2048. Otherwise, it indicates that there are two extra bytes, the copy length is represented by the higher 6 bits of the tag byte, and the extra bytes of copy are used to describe the offset.

2.4.2 Snappy decompression

During decompression, the input file is a compressed file and output file is the corresponding uncompressed file. When the decompression program runs on a single core of a CPU, at first, the size of the uncompressed file stored into the first several (less than six) bytes is extracted. If the most significant bit (MSB) of a byte is 1, it means the next byte is also used to describe the uncompressed file size. Otherwise, the current byte is the last byte indicating the uncompressed file size. After this procedure, the start and end address of the output file can be determined.

Second, the data is sequentially read and processed. The tag byte is parsed to determine the token type and the number of extra bytes, and then literal length (for literal type), or copy length and offset (for copy type) are determined.

The third step is the read and the write operation. For literals, the data is directly copied from the input file into the output file, and for copy, memory copy is performed, data is moved from one position to another position in the output file according to the copy length and offset.

After the third step, the decompressor will go to the second step again to find the tag byte of the next token. This procedure is repeatedly executed until reaching the end of input file. After that, the entire input file is decompressed completely, and an uncompressed output file can be obtained.

2.5 Related work

There is some research focusing on hardware decompression [3] [11] [6]. Although most decompression algorithms are related to DEFLATE [21] which is often used in Gzip and ZIP file formats, this could still provide some inspiration for this thesis project. DEFLATE uses the LZ77 [19] compression algorithm followed by Huffman encoding [22]. To decompress a Gzip file, firstly the decompressor should decode the Huffman coding and then decompress the LZ77 compressed data. LZ77 is a sliding dictionary-based compression algorithm. This thesis focuses on the Snappy algorithm based on LZ77. The Snappy algorithm also uses a dictionary or history, but the difference between Snappy and LZ77 is that the dictionary of Snappy is not a sliding window but a fixed 64KB history.

[3] presents a decompression technique called Gompresso/Bit which is used for compressed files like Gzip, and also presents Gompresso/Byte which is based on LZ77 with byte-level encodings such as Snappy. Gompresso/Byte can achieve decompression rates

up to about 10GB/s. There are two kinds of parallelism which are inter-block parallelism and intra-block parallelism. Different blocks do not depend on each other. This paper also gives relative changes between compression ratio and decompression speed to show the tradeoff between them. This paper enlightens the author on the idea of multi-engine architecture.

Gompresso/Byte mainly contains two steps, The first step is parallel compression; the input file is split into equally-sized data blocks which are then independent compressed. This step makes inter-block parallelism feasible. Figure 2.8 illustrates the scheme of parallel compression.

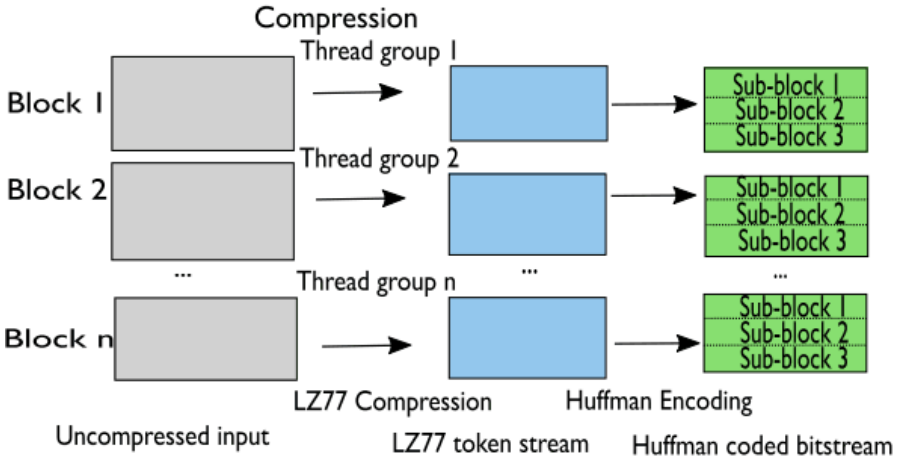

Figure 2.8: Parallel compression of Gompresso [3]

In this thesis, we also propose the similar architecture, because Snappy compression itself has already included splitting file into equal-sized data blocks which makes inter-block parallelism possible. For a Parquet file, there is still similar parallelism available. For example, data pages are compressed independently.

The second step of Gompresso/Byte is parallel decompression which is shown in Figure 2.9. One GPU warp (32 threads operating in lock-step) is responsible for the decompression of a data block, and because each thread decompresses one sub-block of one data block, the starting offset of sub-blocks can be computed from the file header.

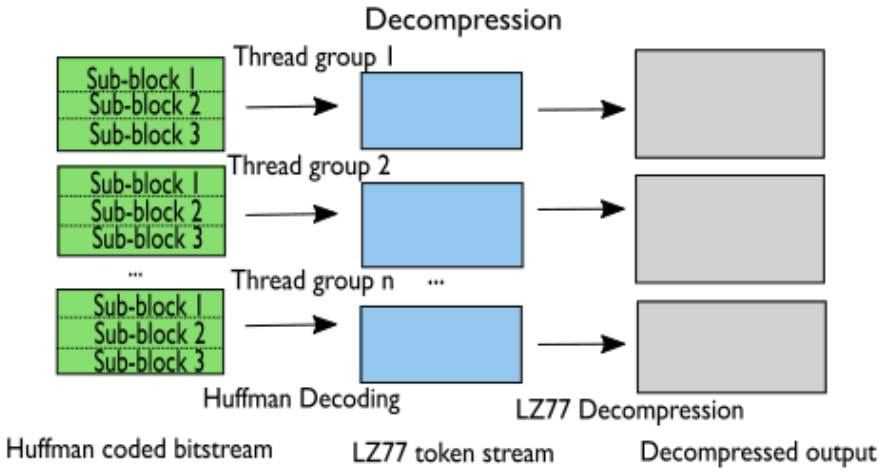

Figure 2.9: Parallel decompression of Gompresso [3]

However, within a compressed data block, there is too much data dependency. This paper proposed two methods to improve intra-block parallelism, the one is exploiting the SIMD-like architecture of a GPU, which is called multi-round resolution(MRR). Data dependencies occur when a copy in one thread needs to read data from another copy in a former thread. In this case, the later copy should wait for the former copy. Thus, MRR first emits all of the literals. Then MRR emits copies without data dependencies. Afterwards, the last step is repeated until all of the tokens are decompressed. There is a flag called the "high-water mark"(HWM) determining whether dependency is resolved that is updated at the end of each iteration. Figure 2.10 describes how MRR works. The other method is eliminating data dependency during compression.

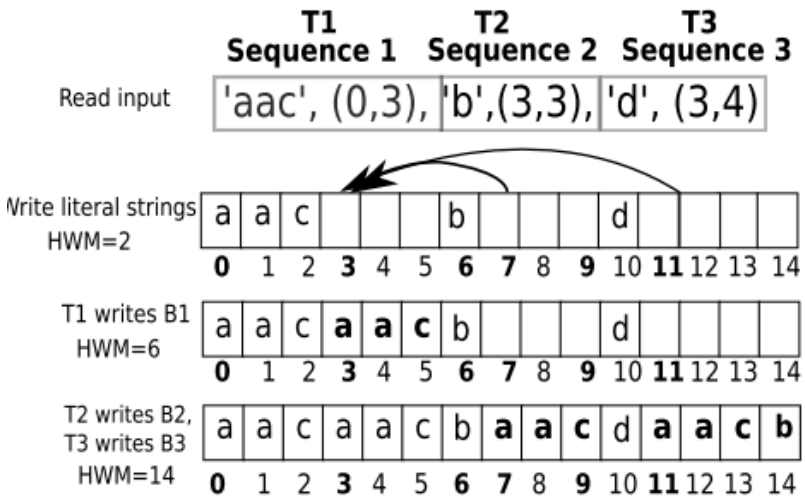

Figure 2.10: MRR execution [3]

[4] proposes a hardware decompression architecture based on DEFLATE. The input

line of data is a portion of the variable length encoded(Huffman encoded) data stream. Thus, the difficulty is how to identify the boundary between symbols. The block diagram of decoder/decompressor is shown in Figure 2.11. The input is M bytes per cycle and the output is N bytes per cycle.

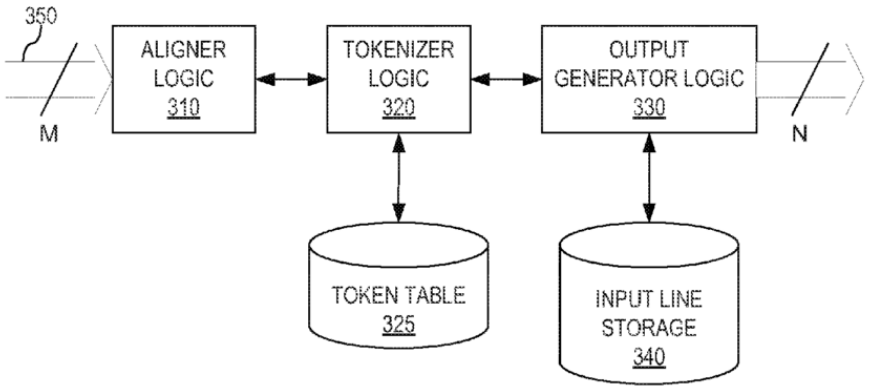

Figure 2.11: Block diagram of decoder/decompressor [4]

Because the input is fixed length and Huffman encoding is a variable length encoding, a symbol may be cut by the boundary of each input line, as can be seen in Figure 2.12.

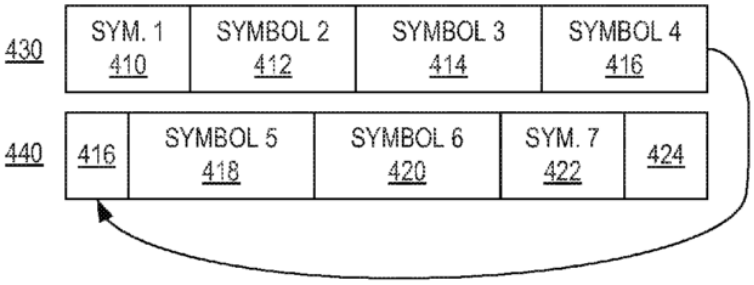

Figure 2.12: Spill-over of bits [4]

Hence, an aligner logic is used to detect and rearrange the symbols. Figure 2.13 shows how the aligner logic works. CL is computation logic used to decode one symbol and update an accumulative shift amount. Hmin and Hmax are the minimum and maximum length of a Huffman code, respectively. So the pipeline depth is 8*M/Hmin and there are Hmax ways of possible solutions. In this way, the first symbol will be found at the end of the first cycle. In the end, the "bit spill" will select only one of Hmax ways as actual output.

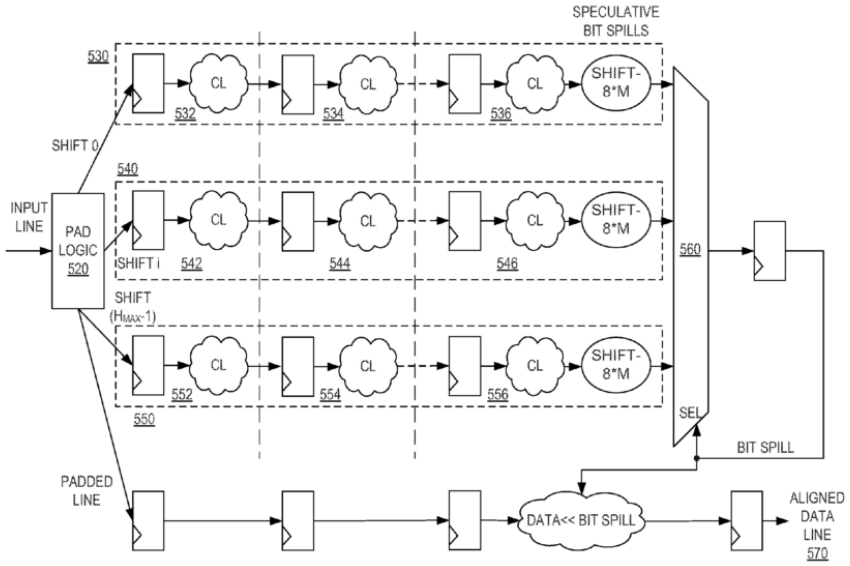

Figure 2.13: Aligner logic [4]

Afterwards, the correct aligned data line will be transferred to tokenizer logic. The tokenizer extracts the symbol data from the compressed symbols in the aligned input line. Finally, output generator logic takes the tokens from the tokenizer logic and generates decoded/decompressed data of N bytes per cycle.

Jinho Lee proposed a framework, named ExtraV, for near-storage graph processing [5]. In this paper, the expand-and-filter combines the decompressor with filter, which is shown in Figure 2.14. The expand phase is decompression, the bandwidth of the storage device is amplified by the compression ratio. The data will get larger due to decompression, which increases the bandwidth requirement of CPU-AFU interface. The filter here could normally reduce the bandwidth requirement of the AFU-CPU interface.

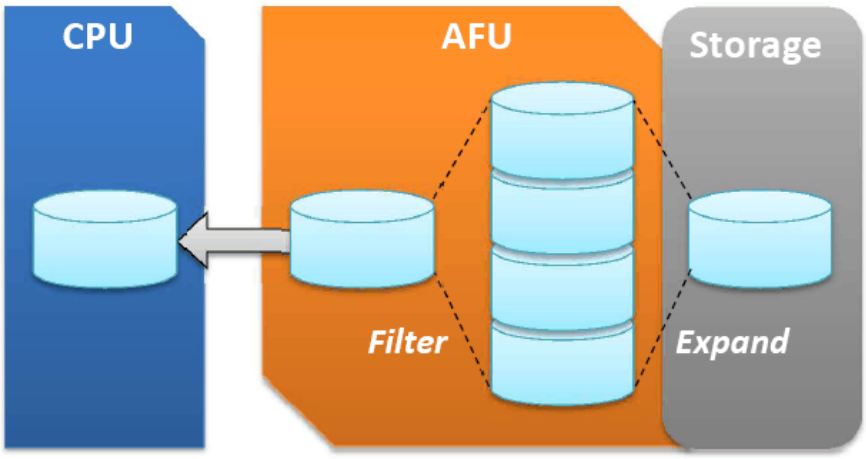

Figure 2.14: Scheme of expand-and-filter [5]

Snappy (de)compression is developed by Google, and Google has published two patents about hardware acceleration for LZ77-based algorithms including Snappy [11, 6]. [11] proposes a general idea for data decompression. The symbol can be decoded using either a fast-path routine or a slow-path routine. The slow-path routine includes a branch prediction hardware. Next, [6] is based on this idea and proposes a hardware and software architecture with interconnections of the whole system. The embodiment of a system for accelerated decompression is shown in Figure 2.15. Generally, the architecture includes memory, processor core, and accelerator. The compressed stream, uncompressed stream, and fast and slow processing code are stored in memory. The accelerator reads compressed data from compressed stream, gets decoded data and outputs intermediate records, formats an intermediate record into a token, and emits fixed-length tokens (e.g. 5 bytes). Next, the execution units of the processor core would do specific memory operations to generate uncompressed data. In this system, the accelerator and the execution units act as a front-end and a back-end, respectively.

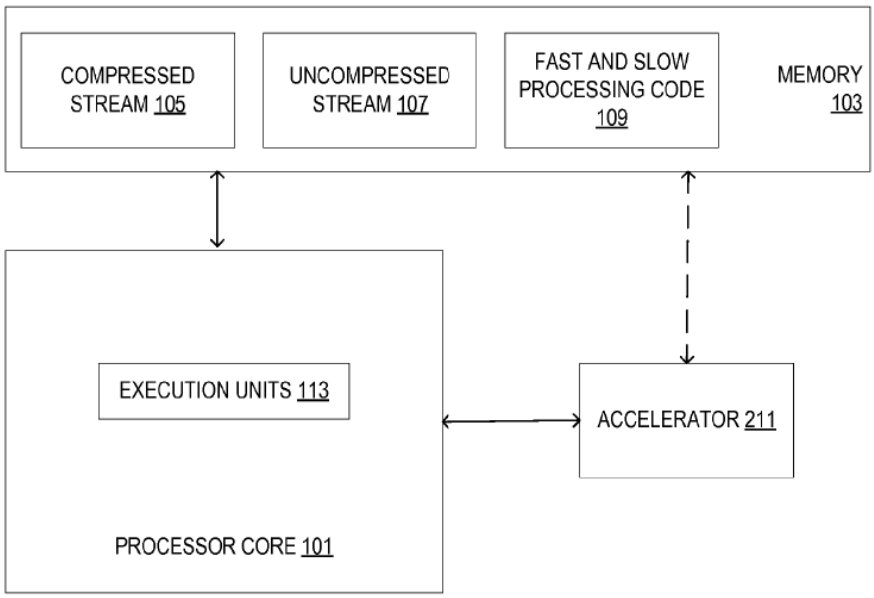

Figure 2.15: Embodiment of a system for accelerated decompression [6]

3 Method Analysis

3.1 Requirement

According to the background knowledge in section 2.2, an accelerator functional unit (AFU) could get 25.6 Gb/s per lane via OpenCAPI. So far, eight lanes can be provided; the bandwidth limit can be improved up to 25.6 GB/s. Assuming that the working frequency for our target Xilinx FPGA (KU15P) is 200MHz, this means an AFU could get 128 bytes per cycle via OpenCAPI. The Xilinx KU15P has 34.6Mbit block RAM (BRAM), 1045K CLB flip-flops, and 523K CLB LUTs [7]. To make the design reasonable, the first thing that should be determined is how many tokens are included in 128 bytes of data on average. Therefore, statistic data of test files need to be gathered. Table 3.1 illustrates basic statistics of test files, where "num_token" means the total number of tokens of the test file, which is also the sum of "num_literal" and "num_copy". Besides, "av_size_input" and "av_size_output" are average sizes per token of the compressed and the uncompressed file, respectively. TPCH is a text file generated by the author, while the other files are biological data from National Center for Biotechnology Information(NCBI) [23].

Table 3.1: Basic information of test files

Test File	compressed size (byte)	uncompressed size (byte)	compression ratio (x)	num_copy	num_literal	num_token	av_size_input	av_size_output
TPCH	4628431	11728193	2.534	1448929	255612	1704541	2.715	6.881
gbbct162.aso	114302958	269396757	2.357	10368744	5106536	15475280	7.386	17.41
gbbct168.aso	101890399	262320412	2.575	9848745	4638536	14487281	7.033	18.11
gbbct1.aso	79093036	262144601	3.314	9942758	3706722	13649480	5.795	19.21

From the Table 3.1, general conclusions can be drawn. One token in the compressed file occupies approximately 3∼7 bytes. The compression ratio of the Snappy algorithm is about 2x. Although an AFU gets 128 bytes of data per cycle via OpenCAPI, it is not realistic that the AFU could process all of the tokens included in the 128 bytes within one cycle. Therefore, a multi-engine architecture is reasonable, where each engine does not have to process too many tokens per cycle.

Several factors constrain the number of engines. The first factor is the BRAM resource needed per engine. For each compressed block, in principle, a 64KB BRAM block is the minimum requirement so that a Xilinx KU15P could provide at most 67 groups of 64KB BRAM blocks. The second factor is the average input throughput. Because input

files get larger after decompression and different files have different compression ratios, it is more reasonable to consider the input throughput. Assuming that each engine consumes N bytes input data per cycle on average, then our design needs 128/N cycles to consume 128 bytes data completely. At the (128/N+1)-th cycle, the engine could obtain new 128 bytes data via OpenCAPI. Thus, 128/N gives an estimated value of the number of engines. Another factor is the hardware resource utilization per engine.

As is mentioned in section 1.3, the engine is designed to process two tokens per cycle. A token occupies at least two bytes according to Figure 2.6 and 2.7. Normally, the size of data being processed is the M-th power of two, where M is a non-negative integer. Four bytes are usually less than the average length of one token. 8 bytes could contain 1∼4 tokens, and the average number of tokens is two. 16 bytes could contain 1∼8 tokens, and the average number of tokens is four, which is more than needed. Hence, initially, each engine is designed to consume 8 bytes, which include an average two tokens, per cycle. The number of engines is initially set to 16.

Because a Snappy compressed file does not have any dependencies between compressed blocks, different engines should be responsible for different compressed blocks, which is much simpler than having different engines process the same compressed block. The page is the unit of compression according to the Parquet format, and the page size is recommended to be set to a small number such as 8KB for the purpose of fine-grained reading [15]. According to the Snappy algorithm, each compressed block has a 64KB history window. Hence, the parallel unit could be a data page. Thus, the multi-engine architecture could also make inter-block parallelism [3] possible.

3.2 Analysis of Block RAM

Block RAM (BRAM) is a configurable memory module that attaches to a variety of BRAM interface controllers [24]. BRAM is a clock triggered device, after the rising edge of a cycle, a line of data can be written, and another line of data can be read in the same cycle. If the two lines are the same line, it is called address collision, which will be discussed in section 3.2.2.

3.2.1 SDP and TDP

Normally, a BRAM primitive can be configured as simple dual port ram (SDP, see Figure 3.1) or true dual port ram (TDP, see Figure 3.2). There are two independent ports (A and B) in TDP RAM mode and each of port A or port B has a read port and a write port. In an SDP RAM mode, there are also two ports. However, port A only works as a read port and port B only works as a write port. Hence, for a given size of the primitive, the primitive configured as SDP RAM is able to reach a larger port width than that configured as TDP RAM.

Additionally, there are two kinds of primitives, RAMB18E2 and RAMB36E2 (see Table 3.2), which means the port width of port A or B can be up to 18 bits and 36 bits, respectively. In order to get a larger read and write port width, the RAMB36E2 configured as SDP RAM is preferred. In this way, the width of the read and write port can reach up to 64 bits per BRAM primitive.

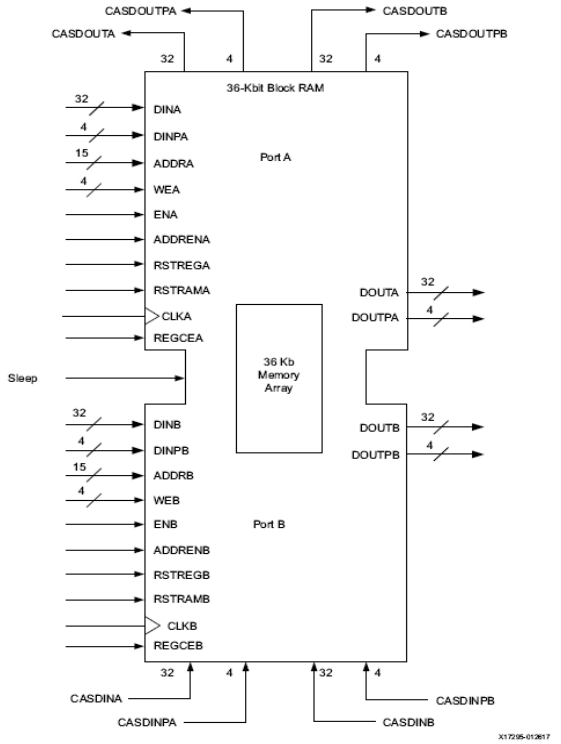

Figure 3.1: RAMB36 usage in a TDP data flow [7]

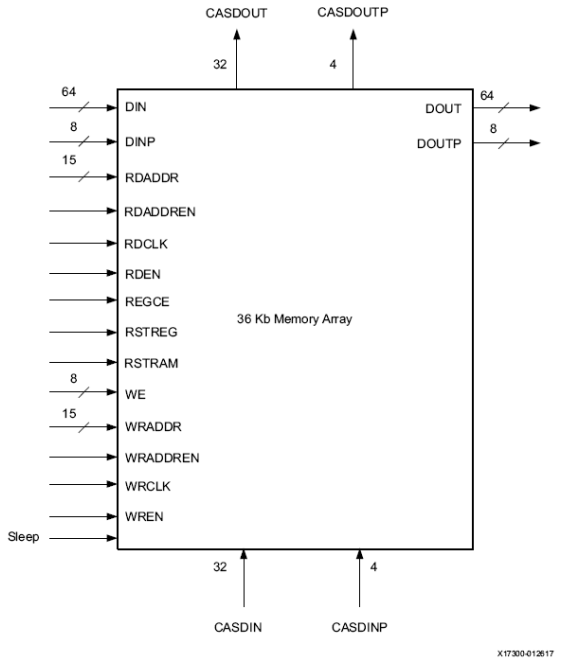

Figure 3.2: RAMB36 usage in an SDP data flow [7]

Table 3.2: Parity use scenarios [7]

Primitive	Settings		Effective Read Width	Effective Write Width
	Read Width	Write Width		
RAMB18E2	1, 2, or 4	9 or 18	Same as setting	8 or 16
RAMB18E2	9 or 18	1, 2, or 4	8 or 16	Same as setting
RAMB18E2	1, 2, or 4	1, 2, or 4	Same as setting	Same as setting
RAMB18E2	9 or 18	9 or 18	Same as setting	Same as setting
RAMB36E2	1, 2, or 4	9, 18, or 36	Same as setting	8, 16, or 32
RAMB36E2	9, 18, or 36	1, 2, or 4	8, 16, or 32	Same as setting
RAMB36E2	1, 2, or 4	1, 2, or 4	Same as setting	Same as setting
RAMB36E2	9, 18, or 36	9, 18, or 36	Same as setting	Same as setting

3.2.2 Write mode

If the read and write address of a BRAM primitive are not the same, the BRAM can be independently written and read a line of data within one cycle. Otherwise, there are three write modes showing different behaviors when faced to address collision, which are WRITE_FIRST mode, READ_FIRST mode, and NO_CHANGE mode. When an address collision happens, BRAM working in WRITE_FIRST mode outputs the newly written data, BRAM working in READ_FIRST mode outputs previously stored data while new data is being written, BRAM working in NO_CHANGE mode maintains the output previously generated by a read operation [7]. Figure 3.3, 3.4 and 3.5 illustrate the waveforms of the three write modes.

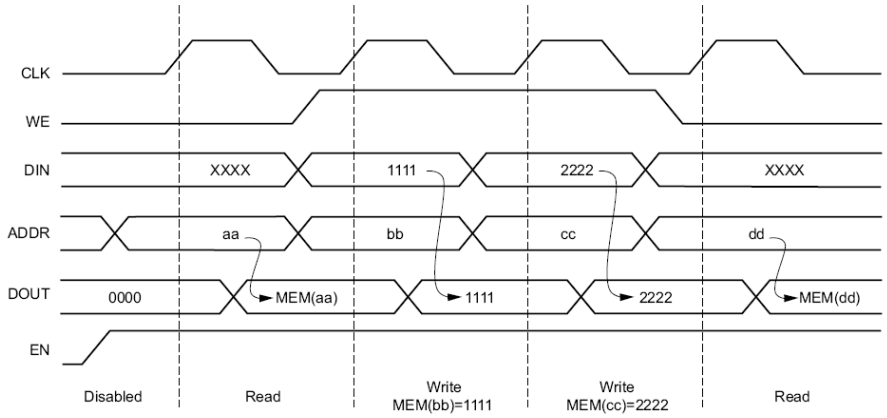

Figure 3.3: WRITE_FIRST Mode waveforms [7]

For a simple dual port ram, the write port always successfully commits the data into memory. However, the read port data is deterministic only for common clock designs (both read and write clocks are driven by the same clock buffer) and the write port is in READ_FIRST mode [7]. Although the WRITE_FIRST mode looks the best, in this thesis, the clock is a common clock. Thus, the write mode of a primitive cannot be configured in WRITE_FIRST mode but in READ_FIRST mode. An improvement

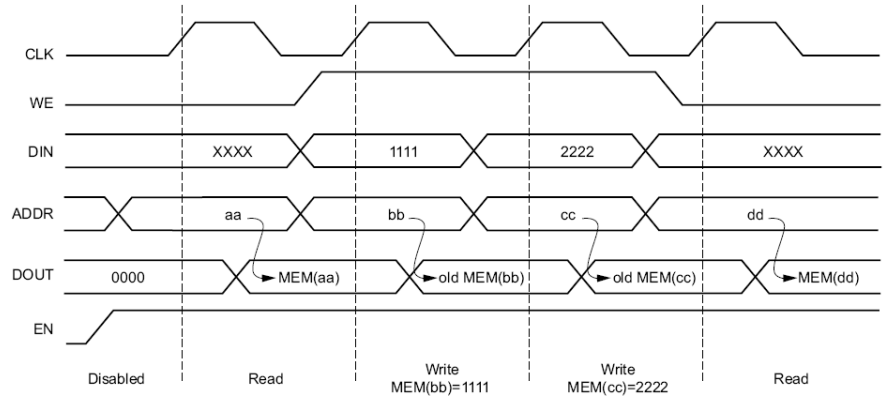

Figure 3.4: READ_FIRST Mode waveforms [7]

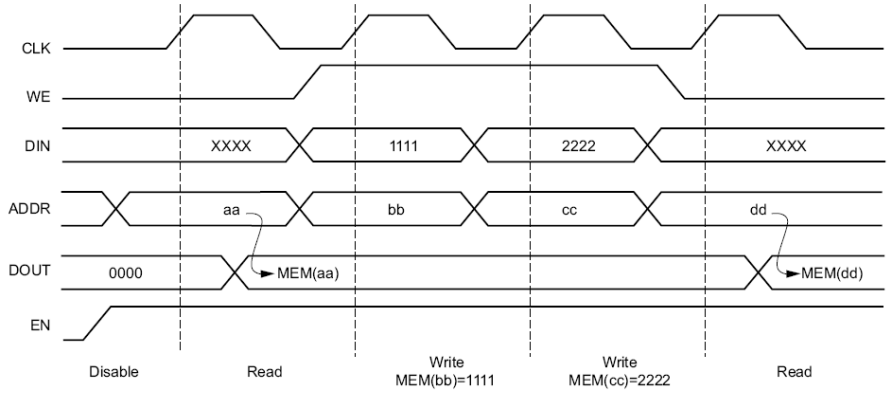

Figure 3.5: NO_CHANGE Mode waveforms [7]

method, through which the BRAM works just like in WRITE_FIRST mode, is adopted. More detailed discussion about this is shown in section 3.4.2 and 4.2.5.

3.2.3 BRAM latency

In Figure 3.3, 3.4 and 3.5, read enable and read address signals arrive at the input port of BRAM before the first rising edge, and then the data, MEM(aa), appears at DOUT after the first rising edge. If the previous module of BRAM is also clock triggered, then it can only read the MEM(aa) after the second rising edge. Hence, it takes one extra cycle to read MEM(aa). Write operation has the same story. In this way, the scheme describing a single engine is drawn in Figure 3.6, and the corresponding block diagram is drawn in Figure 3.7, respectively. "X" means not doing any operation because literal does not need to read data from BRAM, it only needs to write the input data into BRAM. Unknown module refers to the module before BRAM. Figure 3.6 describes a four-stage pipeline structure. The second stage named "Read Out/X" is simple and will not consume too much resource. However, this four-stage pipeline structure would introduce inter-stage data dependencies and intra-stage data dependencies. The intra-stage data dependency

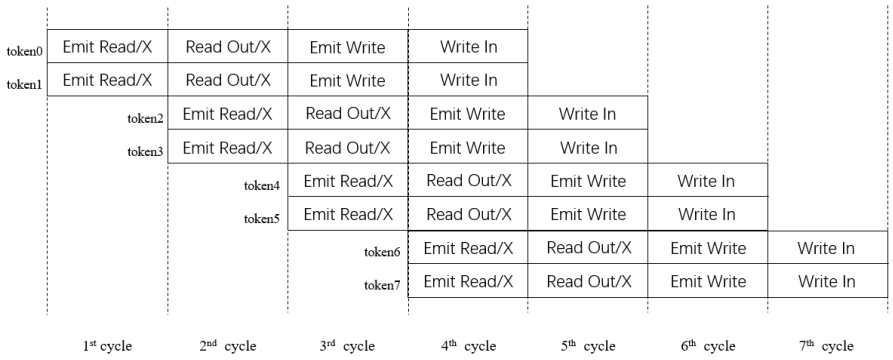

Figure 3.6: Scheme of four-stage structure

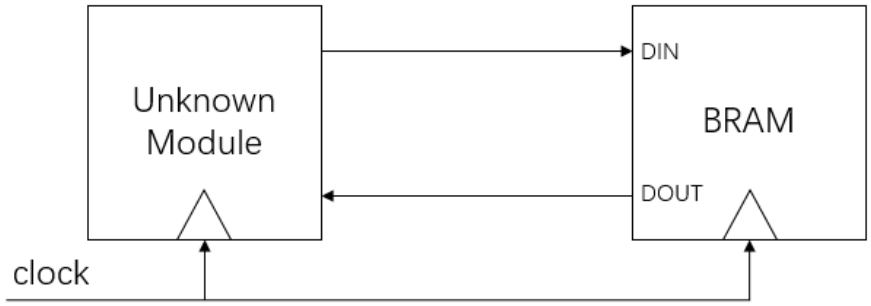

Figure 3.7: Block diagram of four-stage structure

is the dependency between two tokens from the same stage. Inter-stage data dependency means a dependency among tokens from different pipeline stages.

Things are different for WRITE_FIRST mode and READ_FIRST mode. For WRITE_FIRST mode, for example, there is no dependency between token6 and token3 even if token6 has to read data from token3. Because the cycle when token6 emits the read signals and the cycle when token3 emits the write signals is the same cycle, 4th cycle. According to the mechanism of the WRITE_FIRST mode, the read operation always reads the newly written data from BRAM. However, for READ_FIRST mode the same scenario has a different result. For READ_FIRST mode, token6 cannot read the newly written data of token3 so that a data dependency can exist. However, the inter-stage dependency between adjacent two stages may still happen in the case of both WRITE_FIRST mode and READ_FIRST mode. Therefore, this four-stage structure is not a good solution.

Another good and feasible solution is combining the second stage with the third stage into one stage. If the module before BRAM is not sequential logic but combinational logic, the output data of BRAM could be sensed and processed before the next rising edge. That is to say, the MEM(aa) can be read and processed before the second rising edge in Figure 3.3, 3.4 and 3.5. The resulting scheme and block diagram are illustrated in Figure 3.8 and Figure 3.9. Figure 3.8 shows a three-stage pipeline structure which reduces the logic complexity. The unknown module is the alignment module described

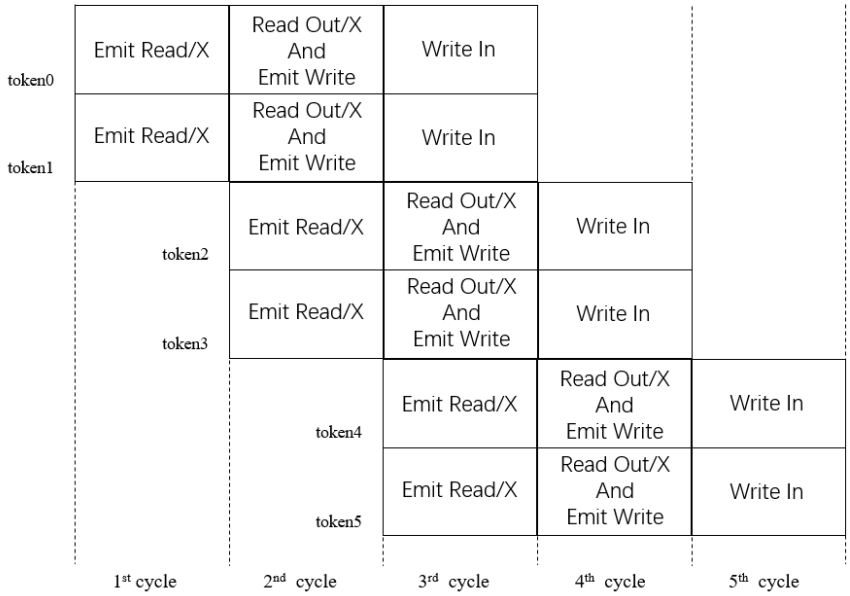

Figure 3.8: Scheme of three-stage structure

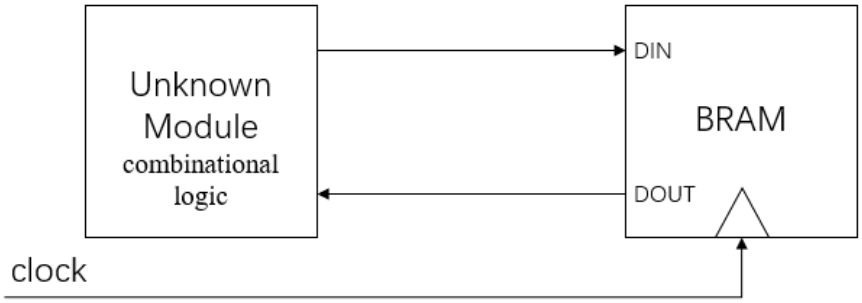

Figure 3.9: Block diagram of three-stage structure

in section 4.2.4.

The three-stage structure could eliminate inter-stage data dependency for WRITE_FIRST mode and reduce one source of inter-stage data dependencies for READ_FIRST mode. The intra-stage data dependency cannot be removed.

3.2.4 Byte-wide write enable

There is a byte-wide write enable input port named “WE” for each BRAM primitive, it is an 8-bit wide port if the write port of a BRAM primitive is 64 bits, where each bit can be set or reset independently. The value of one bit indicates whether the corresponding byte of input data can be written into BRAM.

3.3 Data dependency and address conflict

The definition of data dependency has already been introduced. Address conflicts can be further divided into two subcategories, the first one is the read address conflict, which means two or more tokens need to simultaneously read different lines of the same BRAM primitive. The second one is the write address conflict, which means two or more tokens have to write data into different lines of the same BRAM primitive.

3.3.1 Data dependency

Data dependency can only happen between a copy and a token(copy or literal). The root of data dependency is a copy needs to read data that has not been written into BRAM. There are two types of data dependency, which are inter-stage dependency and intra-stage dependency.

However, the BRAMs working in different write modes have different numbers of potential data dependencies. The BRAM working in NO_CHANGE mode would have more data dependencies than that working in READ_FIRST mode. Because once address collision of one BRAM happens, the BRAM working in READ_FIRST mode could still read the data stored in that line even if it is the old data, and in many cases, there is still a portion of so-called old data being useful.

A BRAM working in READ_FIRST mode usually has more data dependencies than one working in WRITE_FIRST mode, which has been discussed in section 3.2.3. The data dependency that happens on the BRAM working in READ_FIRST mode could be solved by forwarding. Forwarding means that if we look at the example in Figure 3.8 and token4 is assumed to be a copy that has to read data from token2 and token3, the data of token2 and token3 is prepared well before the rising edge of the 4th cycle, so it can be transferred to the second stage of token4 and then used by token4.

In conclusion, the BRAM working in READ_FIRST mode has an extra source of data dependency which could be solved by forwarding. However, the forwarding technique would consume much more resource which will be mentioned in section 3.4.2.

3.3.2 Address conflict

As mentioned above, address conflicts consist of read address conflicts and write address conflicts. No matter which kind of address conflict it is, the reason it happens is that for a BRAM primitive configured as SDP RAM, it only has one read port and one write port, so it cannot accept two different address signals at the same time.

Both read address conflicts and write address conflicts only happen when both tokens that are going to be emitted are copies. For read address conflicts, the reason is that the literal does not need a read operation. For write address conflicts, if one of the tokens is literal, then the sum of the length of two tokens is no larger than 8+64=72 bytes. Because each literal that to be emitted is no larger than 8 bytes, an engine is designed to read 8 bytes per cycle. The long literal will be split into many short literals. In our design, the write address conflict may only happen when the sum of length of two tokens is larger than 121 bytes.

For each compressed block, one 64KB BRAM block is necessary according to the Snappy algorithm in section 2.4. Note that a BRAM block is composed of many BRAM primitives. As is shown in Table 3.2, if the choice of primitive is not just limited to RAMB36, then a primitive can be configured as an SDP RAM with the port width of one byte, two bytes, four bytes and eight bytes. The port width of a primitive is called granularity. The capacity of a primitive is fixed, and the size of RAMB36 and that of RAMB18 are 36Kbit and 18Kbit, respectively. Thus, a larger port width also means a smaller depth.

The statistics about the read and write address conflicts is given in section 3.3.2.1 and 3.3.2.2. It is noted that the statistic data of address conflicts in Table 3.3 and Table 3.4 does not represent the actual situation because it is hard to determine which two successive tokens would be processed within one cycle during the actual hardware decompression. The statistics shown in Table 3.3 and Table 3.4 reveal a general distribution or frequency of conflicts.

3.3.2.1 Read address conflict

The read ranges of two copies are random, so read address conflicts will always have the possibility of occurrence as long as there is only one BRAM block whose size is 64KB being used for one compressed block. As is known above, the sum of the length of two copies is no larger than 128 bytes. Hence, the width of one BRAM block is initially set as 128 bytes, namely, 1024 bits. Table 3.3 gives statistics about the frequency of the read address conflict of some test files, which is based on the change of granularity.

"num primitive per BRAM block" is the number of primitives standing side by side in one BRAM block. Because the port width of a BRAM block is fixed to 128 bytes, the product of granularity and "num primitive per BRAM block" is a constant. "num_copy" is the total number of copies in the sample file. "num consecutive copy" is the number of consecutive copies, consecutive copy means any two successive copies. The statistics about the same lines of one or more primitives. "num_read_conflict" is the number of the read address conflict which means two successive copies will read different lines of at least one primitive. Finally, "num_no_conflict" is the number for which there is no overlap or read address conflict between two copies. Each of "num_overlap", "num_read_conflict", and "num_no_conflict" will be divided by "num consecutive copy" to get the corresponding percentage and the sum of three percentages should add up to one. The case of overlap can always be solved by some alignment operations.

A conclusion can be drawn from the Table 3.3 that the percentage of the read address conflicts decreases with the decreasing granularity, which is reasonable because one read address points to a line of data in one primitive, and if the width of one primitive gets shorter then the read address conflict is less likely to happen. However, read conflicts cannot be eliminated even if the granularity is one byte. Figure 3.10 shows this trend.

Unfortunately, if the granularity gets smaller, the BRAM resource usage will become larger. For example, if the granularity is one byte, then a BRAM block needs 128 primitives. If the primitive is RAMB18 whose effective size is 2KB, then 256KB of BRAM resource is necessary in order to obtain one BRAM block. However, only 64KB of which can be effectively used as history or dictionary. This method would waste lots of BRAM

Table 3.3: Statistics about the frequency of the read address conflict

Test File	granularity (byte)	num primitive per BRAM block	num_copy	num consecutive copy	num_overlap	%_overlap	num_read_conflict	%_read_conflict	num_no_conflict	%_no_conflict
TPCH	8	16	1448929	1193496	2947	0.2469216	191749	16.06616193	998800	83.68691642
	4	32	1448929	1193496	2066	0.1731049	155857	13.05886237	1035573	86.76803274
	2	64	1448929	1193496	1518	0.1271894	138111	11.57197008	1053867	88.30084056
	1	128	1448929	1193496	918	0.0769169	129132	10.81964246	1063446	89.10344065
gbbct162.aso	8	16	10368744	5264676	479603	9.1098294	1675033	31.81644986	3110040	59.07372078
	4	32	10368744	5264676	441218	8.3807247	1463162	27.79206166	3360296	63.82721368
	2	64	10368744	5264676	311122	5.9096134	1282409	24.35874496	3671145	69.73164161
	1	128	10368744	5264676	5352	0.1016587	1068120	20.28842801	4191204	79.60991332
gbbct168.aso	8	16	9848745	5212445	511804	9.8188854	1767606	33.91126429	2933035	56.26985033
	4	32	9848745	5212445	462117	8.8656475	1552619	29.78676993	3197709	61.34758256
	2	64	9848745	5212445	321227	6.1626933	1354932	25.99417356	3536286	67.84313312
	1	128	9848745	5212445	5713	0.1096031	1083247	20.78193631	4123485	79.10846062
gbbct1.aso	8	16	9942758	6237715	406697	6.5199676	2296628	36.81841828	3534390	56.66161407
	4	32	9942758	6237715	353140	5.661368	1976992	31.69417006	3907583	62.64446195
	2	64	9942758	6237715	238071	3.816638	1546814	24.79776649	4452830	71.38559553
	1	128	9942758	6237715	17647	0.2829081	776172	12.44321037	5443896	87.27388154

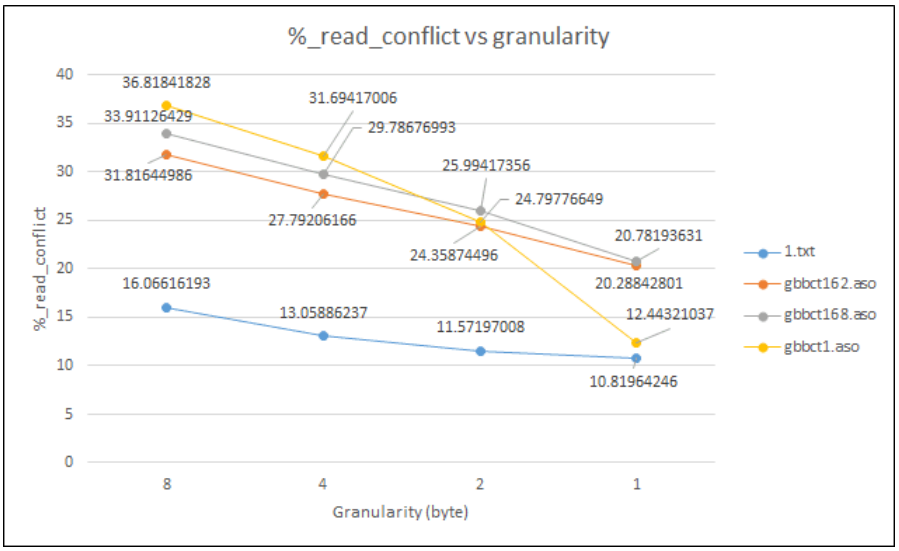

Figure 3.10: Read conflicts as a function of granularity

resource. Therefore, we propose a solution that gets a balance between the frequency of the read address conflicts and required BRAM resources in the section 3.3.2.2.

3.3.2.2 Write address conflict

The write operation is not random but continuous. Write address conflicts can also only happen when both consecutive tokens are copies. Table 3.4 gives statistics about the frequency of write address conflicts of some test files as a function of granularity. Also, Figure 3.11 shows the trend of how write address conflicts change when minimizing the granularity.

Table 3.4: Statistics about the frequency of write address conflict

Test File	granularity (byte)	num primitive per BRAM block	num_copy	num consecutive copy	num_overlap	%_overlap	num_write_conflict	%_write_conflict	num_no_conflict	%_no_conflict
TPCH	8	16	1448929	1193496	1044625	87.526477	0	0	148871	12.47352316
	4	32	1448929	1193496	895461	75.028404	0	0	298035	24.97159605
	2	64	1448929	1193496	597730	50.082279	0	0	595766	49.91772071
	1	128	1448929	1193496	0	0	0	0	1193496	100
gbbct162.aso	8	16	10368744	5264676	4217839	80.115832	390771	7.422508052	656066	12.46165956
	4	32	10368744	5264676	3646379	69.261223	303389	5.762728799	1314908	24.97604791
	2	64	10368744	5264676	2446054	46.461625	185500	3.523483686	2633122	50.0148917
	1	128	10368744	5264676	0	0	0	0	5264676	100
gbbct168.aso	8	16	9848745	5212445	4085311	78.376098	476292	9.137592819	650842	12.48630921
	4	32	9848745	5212445	3526422	67.653894	382261	7.333621746	1303762	25.01248454
	2	64	9848745	5212445	2362382	45.321955	241490	4.632950564	2608573	50.045094
	1	128	9848745	5212445	0	0	0	0	5212445	100
gbbct1.aso	8	16	9942758	6237715	4150553	66.539638	1309013	20.98545701	778149	12.47490467
	4	32	9942758	6237715	3567521	57.192754	1112470	17.83457564	1557724	24.97267028
	2	64	9942758	6237715	2381668	38.181738	742764	11.90762964	3113283	49.91063234
	1	128	9942758	6237715	0	0	0	0	6237715	100

The labels in Table 3.4 have similar definitions to those in Table 3.3. A conclusion can be obtained that the percentage of write address conflicts also decreases with decreasing granularity. Furthermore, the possibility of write address conflicts can be eliminated only if the granularity is one byte. That is because each byte of data has its own address. So granularity is indeed the minimum size of data that can be addressed. However, just as is mentioned in the previous section, the smaller the granularity is, the more BRAM resource is used and wasted. Assumed that granularity is one byte, so all of write address conflicts and many read address conflicts can be eliminated. But each engine requires 256KB of BRAM as computed in section 3.3.2.1, and the system is designed to have 16

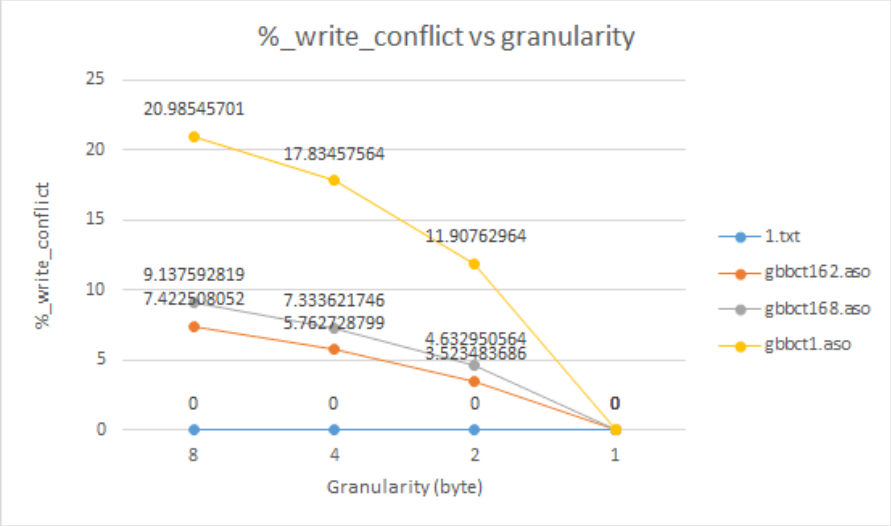

Figure 3.11: Write conflicts as a function of granularity

engines, yields 4MB of BRAM, which is a huge consumption and waste of resource. For the Xilinx KU15P, there is only about 4MB of BRAM resource being supported.

If we examine Table 3.3 and Table 3.4 more deeply, the statistics show that there are much more read address conflicts than write address conflicts. Also, the decrease of granularity has little effect on write address conflicts. Thus, we propose a structure that could eliminate all of the read address conflicts but does not affect write address conflicts. In this structure, two identical BRAM blocks of 64KB each will be used, and the granularity is 8 bytes, so 16 primitives of RAMB36 form a BRAM block. Two BRAM blocks share their write ports but have independent read ports, which can guarantee that at any time the data written into two BRAM blocks is the same. Besides, independent read ports make independent read operations of two copies possible.

3.4 FPGA logic resource

3.4.1 Special copy

Normally, assuming that for the output file, the start output address of a copy is denoted as “op”. A copy contains information about offset and length, denoted as (offset, length), then the copy needs to read data from output file itself. It reads “length” bytes of data from the start address of “op-offset” in the output file.

However, the possibility that the offset is less than the length exists, which means that copy can only read “offset” bytes of data because the following data is the copy itself and has not been written into the output file. In this case, the number of bytes being read varies from 1 to 63, and the range of copy length is from 4 to 64. Hence, there are lots of combinations of offset and length which will consume a large number of hardware resources. To solve this issue appropriately, at first the frequency that this problem occurs should be gathered as statistics. Table 3.5 gives the frequency of special

copy in several sample files.

As can be seen from the Table 3.5, the frequency where the offset is less than the length is extremely low. It is not reasonable or realistic to consume too much hardware resource for this problem. Thus, an extra software module that can preprocess the compressed file could be added before hardware decompression. This module will be implemented via software on the CPU. Its function is splitting a special copy into many sub-copies, for example, a special copy with the offset of 3 and the length of 7; then it will be cut into three copies, which are (3,3), (3,3) and (3,1). However, a copy whose length is less than 4 bytes is not supported by the Snappy algorithm, which is mentioned in section 2.4. Some small changes can be made on compression program for solving this problem. For example, for type "01" copy, the 2∼4 bits may not represent "length-4" but represent "length-1". Alternatively, we could even add a new type of copy to indicate this special case.

Both methods would lead to a throughput penalty, extra cycles are needed to process each particular copy, but this penalty is small and acceptable due to the low frequency of occurrence. Also, in a future implementation, the preprocessing done by the CPU could be transferred to the FPGA.

Table 3.5: Special copy statistics

Test File	num_copy	num_literal	num_token	num_offset<len	%_offset<len
TPCH	1448929	255612	1704541	425	0.02933201
gbbct162.aso	10368744	5106536	15475280	382786	3.691729683
gbbct168.aso	9848745	4638536	14487281	335142	3.402890419
gbbct1.aso	9942758	3706722	13649480	505973	5.08885965

In this thesis, we do not implement this software module due to time reason. This part of work is not difficult or complex. During behavior simulation, to get a correct uncompressed file, we unblock the VHDL code that would consume too much resource and is used to solve the special copy. However, during synthesis and implementation, we assume that the special copy issue has been solved and we block the related VHDL code.

3.4.2 The choice of write mode of BRAM

As is discussed in section 3.3.1, the architecture whose BRAMs working in READ_FIRST mode has more potential data dependencies than that whose BRAMs working in WRITE_FIRST mode. Therefore, the modules related to data dependency and shown in section 4.2.3 and 4.2.4 would get much more complex. The section 3.3.1 also mentions that a primitive cannot be configured in WRITE_FIRST mode for a common clock design. Thus, a wrapper is added outside of each primitive to make BRAM behave just like in WRITE_FIRST mode. The BRAM primitive itself still works in READ_FIRST mode.

In the initial design of this thesis, the wrapper is not used, and each primitive is configured in READ_FIRST mode, which leads to the whole hardware architecture ex-

cept for BRAM blocks consuming 81% of look-up table(LUT) resource of Xilinx KU15P. However, after changing write mode with so-called WRITE_FIRST mode, it only consumes 17.43% of LUT resource. It is reasonable to save such a huge logic consumption at the cost of a little logic resource of BRAM. Hence, in this thesis, write mode is configured in pseudo WRITE_FIRST mode. The detailed configuration method is shown in section 4.2.5.

4 Implementation

This chapter introduces how the architecture is implemented and the function of each module. The implementation mainly focuses on the design of a single engine which can process at most two tokens per cycle. The hardware module is implemented by VHDL.

4.1 System level design

There are 16 decompression engines, in total, in the system. The AFU can get 128 bytes per cycle via OpenCAPI from main memory and then the data needs to be transferred to one of the engines. Each decompression engine can read 8 bytes per cycle from its 128 bytes of the input buffer. As discussed in chapter 3, different engines are responsible for different compressed blocks, so in principle, there is no dependency between any two engines. An engine emits a read request to show that it has already consumed the 128 bytes of data and is ready to read new data. An engine also emits a write request only after the corresponding compressed block is completely decompressed. If there is only one engine emitting a request at one cycle, then the arbiter will respond to its request. However, the possibility that more than one engine emit request exists. Then, an arbiter that decides which engine could get data is necessary. Figure 4.1 illustrates the system level diagram.

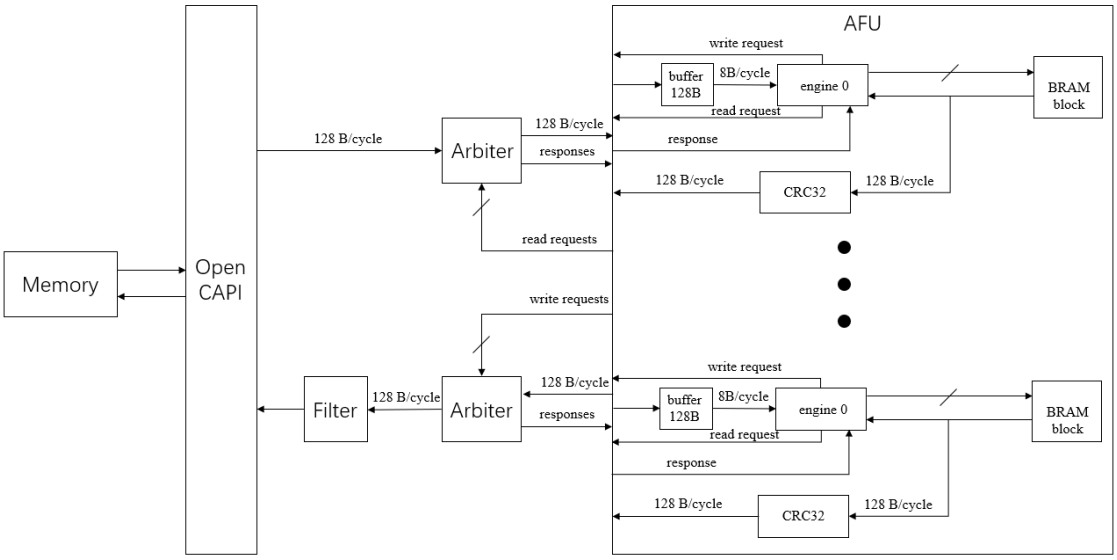

Figure 4.1: System level diagram

The upper arbiter in Figure 4.1 is responsible for read requests, each time, it selects one of the engines emitting read requests and transfers the input data to the input buffer of that engine. Once a read request from an engine is responded to, the AFU needs to fetch the 128 bytes data from the compressed block corresponding to that engine.

The other arbiter in Figure 4.1 processes write requests. It selects one of the engines emitting write requests as an output. Once a write request from an engine is responded, the corresponding engine would read a line of data, starting from the first line of BRAM block, per cycle. Each line of a BRAM block occupies 128 bytes, and each BRAM block contains 512 lines. The data that is read will first pass through a CRC32 module. Finally, the data passes a filter, and the filtered output reaches the OpenCAPI. After the whole uncompressed data is emitted, the result of CRC32 module will be compared with the CRC code of the original file.

4.1.1 Arbiter

Two types of arbiters are common: the Round-Robin arbiter and Fixed-Priority arbiter. If the priorities of engines are given an initial condition, for example, the engine whose subscript is smaller has a higher priority. When multiple engines emit a request simultaneously, a Round-Robin Arbiter will first respond to the engine whose priority is the highest. Next, the arbiter will update its priorities; all of the engines except for the one that has been responded to increase their priorities by one. The engine that has been responded to would be assigned the lowest priority. However, a Fixed-Priority arbiter will never update the priorities of engines.

In this thesis project, each kind of arbiters is feasible, but Round-Robin arbiter is more suitable because, generally, the engine that was just responded to still needs multiple cycles to process the data.

4.1.2 Filter module

The filter can select a part of the data according to a specific filter algorithm. In this thesis, output data rate of 16 decompression engines tends to exceed the bandwidth requirement of OpenCAPI. The filter is used to reduce the output throughput to make the design meet the bandwidth requirement.

The filter is not a main or important module in this thesis. Thus, the function of filter does not have to be too complex. To filter the correct uncompressed data, a filter can be enabled after an entire compressed block is completely decompressed. At that time, the filter needs to read data from BRAM, filter the data, and output filtered data. For simplicity, the function of the filter is selecting characters from “0” to “9”, which corresponds to the binary codes from 30H to 39H.

4.1.3 CRC32 module

A cyclic redundancy check (CRC) is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. CRCs are popular because they are simple to implement in binary hardware, easy to analyze mathemati-

cally, and particularly good at detecting common errors caused by noise in transmission channels [25].

There is a constant called generator polynomial working as the divisor. The most commonly used polynomial lengths are 9 bits (CRC-8), 17 bits (CRC-16), 33 bits (CRC-32), and 65 bits (CRC-64). In this thesis, our final target file is Parquet file, and each page header includes an "i32 crc", which is shown in Figure 2.4. Hence, CRC-32 is preferred, and the corresponding generator polynomial is 0x04C11DB7 (a public value). Because the MSB of a generator polynomials is one, the MSB is omitted. That is why CRC-r could represent r+1 bits.

The raw data (before compression) in a column chunk of Parquet file is divided by the generator polynomial, the resulted remainder is so-called CRC code or check value, which is stored in "i32 crc". The decompressed data also goes through the process that the raw data goes through, after that, a remainder is also obtained. Finally, "i32 crc" is compared with that remainder, if they are equal, then decompression is considered to be correct. Otherwise, the decompression is considered to be wrong. One thing to note is that CRC cannot 100% detect whether the decompression is correct, but its accuracy is extremely high. Although the accuracy depends on the selection of generator polynomial, the well-known value is feasible without doubt.

4.1.4 Architecture of a single decompression engine

Figure 4.2 shows the block diagram of a single engine that can be divided into five parts, which are the parser, FIFO, conflict detector, alignment module and BRAM module.

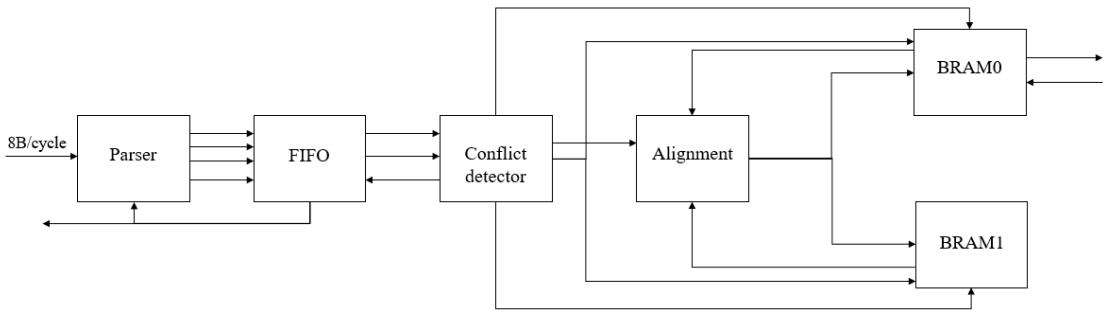

Figure 4.2: Single engine

The parser reads 8 bytes per cycle from the input buffer whose size is 128 bytes if the parser receives the enable signal from the FIFO. Then the parser can output four groups of signals which are used to describe the information of at most four tokens.

The FIFO has 4 data input ports and 2 data output ports. At each cycle, at most four tokens can be written into the FIFO and at most two tokens can be read from the FIFO. The FIFO will also give an almost full signal that can be used as the enable signal to the previous module.

The conflict detector reads at most two tokens from the FIFO module, and detects

the data dependency and write address conflicts between two tokens. It emits the same write signals to both of BRAM blocks but emits independent read signals to the BRAM blocks. Other signals used for describing the token or conflict will be transferred to the alignment module.

The alignment module is a combinational logic module. It receives signals from both BRAM blocks and the conflict detector. It has only one output which is connected to the data input ports of BRAM blocks.

There are 2 BRAM blocks within one engine, with each containing a copy of the decompressed output block up to 64KB. The BRAMs have common write ports but independent read ports to ensure that at any time the data stored in the two BRAMs is same. After a whole compressed block is decompressed completely, the uncompressed data in one of BRAM blocks is read out to the filter module. Because the data can be read out to the filter module at 256 bytes per cycle if both BRAMs are used and data is produced, on average, at 8.3∼15 bytes per cycle, the engine is available for decompression 90% percent of the time.

4.2 Module design

In this section, I/O ports and the detailed function of each module is introduced. All of modules in this section are implemented by VHDL.

4.2.1 Parser module

The parser is a two-stage pipeline structure. To successfully decompress a token according to the data format of the compressed file, first, the copy length and offset have to be obtained for a copy, and the literal length and literal content itself have to be obtained for a literal. The parser is just used to parse the compressed data, extract this information from the input, and emit the decoded tokens. Figure 4.3 shows the I/O ports of the parser module.

As can be seen in Figure 4.3, “en” comes from FIFO module. The parser is enabled when the number of data stored in FIFO module does not exceed the threshold value. “ip_start” is 16-bit wide, it means the start input address of a compressed block, and its initial value is zero. “ip_limit” means the size of compressed data in a compressed block. In a compressed block, the first several bytes indicate the uncompressed size of the compressed block, which will not be considered into “ip_limit”. These two input ports would be given by the CPU. “din” is the 8 bytes of data being read from 128 bytes of input buffer.

A token occupies at least two bytes, and the parser reads 8 bytes per cycle. Hence, there are at most four tokens within one cycle, that is to say, the effective number of parsed tokens varies from 0 to 4. Thus, the outputs of the parser contain information for at most four tokens. Each output port is an array that has four elements, one per token. “writer_en” is 4-bit wide and each bit of it indicates whether the corresponding token is valid. Each bit of “writer_token_type” shows the token type of the corresponding token, where 0 and 1 represent literal and copy, respectively. “writer_len” indicates how many bytes of literal content or copy can be written into BRAMs. “writer_offset” indicates

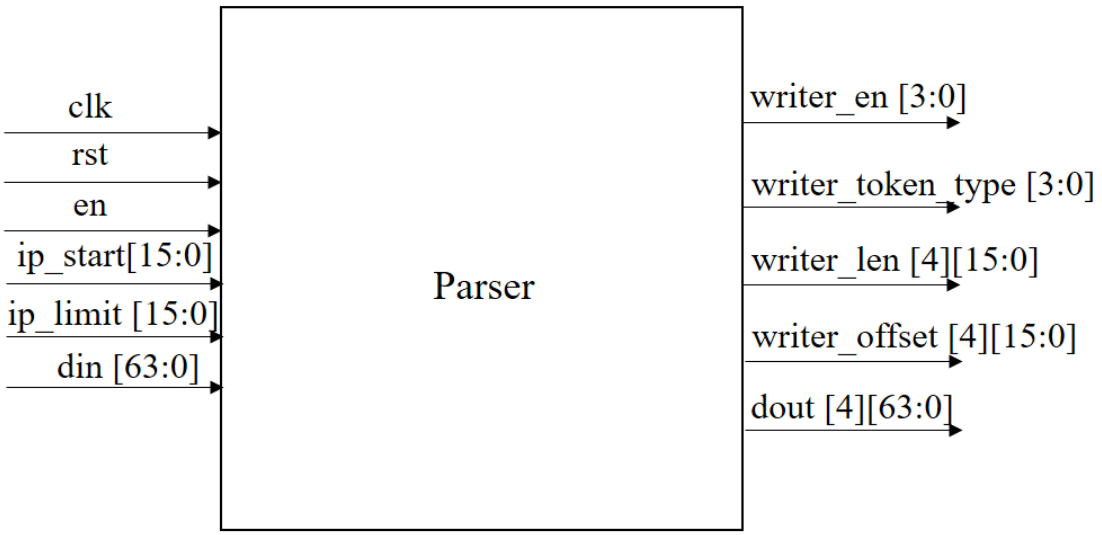

Figure 4.3: I/O ports of parser module

the offset of a copy, it equals zero for a literal. At last, "dout" only represents the literal content itself that has to be written into BRAMs, for a copy, it is equal to zero. Hence, to completely describe a token, a 98-bit wide data is necessary, where 98=1+1+16+16+64. That is why the width of the FIFO module is 98 bits. The internal function and structure of parser is shown in Figure 4.4.

As can be seen from the Figure 4.4, the parser module contains two stages.

- Stage 1

During the first stage, each byte of the input bytes is considered as a possible tag byte. The input 8 bytes are transferred to each of 8 decoders and then decoded independently. Assuming that n is an integer from 0 to 7. The n-th decoder receives the input 8 bytes and considers the n-th byte of input 8 bytes as a tag byte. Next, it extracts the information according to the input data from the n-th byte to the 7-th byte. That is to say, the data before the n-th byte will not be considered. After the decoding, 8 groups of possible decoded data can be obtained, one per token, where each group contains 7 signals, which are "maybe_token_type", "maybe_len", "maybe_offset", "maybe_extra", "maybe_header_finish", "maybe_data" and "maybe_next_tag". The detailed definition is as follows.

The first five signals are used to describe a possible token. "maybe_token_type" is a 1-bit signal, where 0 represents literal and 1 means copy. "maybe_len" indicates literal length for the literal or copy length for the copy. "maybe_offset" represents the offset of the copy, if the token is a literal, then it equals zero. "maybe_extra" equals the number of extra bytes. The number of extra bytes of a literal ranges from 0 to 2, while that of a copy can only be 1 or 2. "maybe_header_finish" is also a 1-bit signal that indicates whether the header of this token is cut by the boundary of 8 bytes of input data, 1 means that the header of the token is not cut by boundary, while 0 means that the header of the

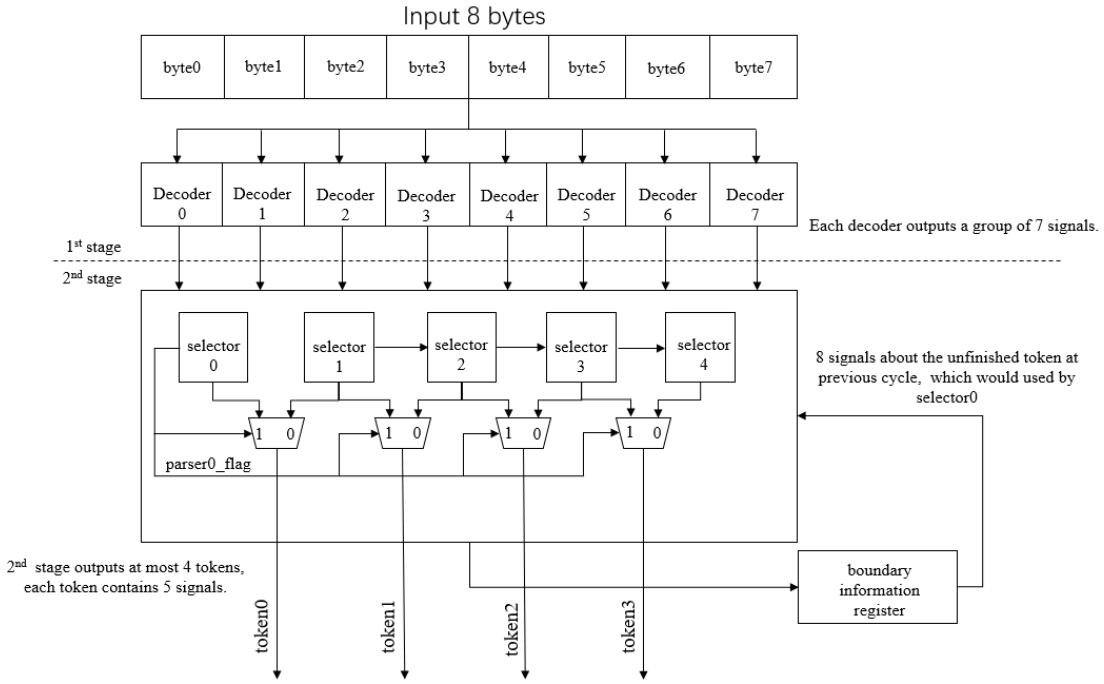

Figure 4.4: Internal structure of parser module

token is cut off, and there will be remaining parts of header in the input of next cycle. "maybe_data" is the input data after being right shifted; because there are 8 groups of data, the input data should be logically shifted rightwards from 0 to 7 bytes. This signal is only used to transfer the literal content if the content of one or more literals is included in the input 8 bytes, then the "maybe_data" can contain all of the possible situations. At last, "maybe_next_tag" indicates the address of next tag byte in the compressed block.

The working principle of a decoder is based on the structure of a compressed token, which is shown in Figure 2.6 and 2.7. The decoder can directly obtain the information about the token type, the number of extra bytes, the token length and the copy offset. The "maybe_header_finish" can be determined by both the number of extra bytes and the position of the byte considered as a tag byte. The "maybe_next_tag" can be computed from the position of the tag byte, the number of extra bytes, and literal length (only if the token is a literal).

However, two things should be kept in mind. The first thing is that the possible tag byte may not be an actual tag byte, because the tag byte of literal or copy has its range of values, for example, if the lower 2 bits of the possible tag byte is "11" then this byte cannot be an actual tag byte anymore. The second thing is that the header of a possible token may be cut off by the boundary of the 8 bytes. The header (tag byte + extra bytes) of a token occupies at most 3 bytes. Hence, the decoder6 and the decoder7 may not give the completely decoded data, which does not matter. It is feasible for these two decoders to give decoded information as much as possible. The incompletely decoded data will be recorded and then recomputed again in the second stage.

- Stage 2

The second stage is related to 5 selectors, from selector0 to selector 4. Although we assume that each byte is a tag byte of a token in the 1st stage, it is impossible for 8 bytes of data to contain 8 tokens. Namely, it is impossible that all the decoders give a correct and real group of data. As is discussed before, 8 bytes of data can contain at most 4 tokens. That is why only 4 of 5 selectors can output their results, which is reasonable and sufficient. Each selector has 5 outputs, "writer_en", "writer_len", "writer_offset", "writer_token_type" and "dout", but only 4 of 5 selectors are able to transfer these signals to next module. All the selectors are able to have access to the outputs of the 1st stage.

During the second stage, there is a strict order among 5 selectors. The selector with a lower index would execute its function first. The selector0 can output its results only if the final token at previous cycle is cut by the boundary. Otherwise, if the first byte at this cycle is not a hypothetical but an actual tag byte, selector0 would not output its results. There is a 1-bit signal named "selector0_flag" indicating whether selector0 could output its results. If "selector0_flag" is 0, then selector1∼selector4 could output their results, otherwise, selector0∼selector3 could output their results.

A selector being able to emit its results does not mean the results are valid. Specifically, from selector1 to selector4, the selector whose index is smaller has a higher priority, which means, for example, if the selector1 cannot output a valid token, then the selectors after that cannot, neither. Things are different between selector0 and selector1, if selector0 can output its valid results, then the address of next tag byte computed from selector0 would decide whether selector1 could output a valid token. Otherwise, as discussed above, selector1 can output a valid token, because the first byte in the 8 bytes of data at this cycle is a tag byte. A valid literal means that the literal being emitted should includes at least one byte of literal content. A valid copy means the length and offset are complete and correct.

If a token is cut by the boundary, there are 2 types of cutting, the one is that the header of a token is cut, the other one is the literal content of a literal is cut. Concrete examples can be seen in Figure 4.5.

As is known that a token may be cut by the boundary of 8 bytes of data. Thus, storing the information of boundary and how the unfinished token is cut by boundary is very important for correct selection in the next cycle. 8 signals describing boundary information are stored in the boundary information register, and then transferred to the parser module itself at next cycle, so that the selectors could adopt corresponding operations according to boundary information at next cycle.

The 8 signals that will be transferred to parser module itself are "unfinish_extra_byte", "unfinish_offset", "unfinish_token_type", "unfinish_len", "unfinish_next_tag", "unfinish_tag_offset", "literal_finish" and "header_finish". "unfinish_extra_byte", "unfinish_offset", and "unfinish_token_type" describe the number of extra bytes, copy offset, and token type of the token that is cut by the boundary. If the header of a token is cut, "unfinish_len" represents the length of that token, if the literal content is cut, "unfinish_len" represents how many bytes left that have to be written. "unfinish_next_tag" indicates the address in the compressed block of next tag byte. "un-

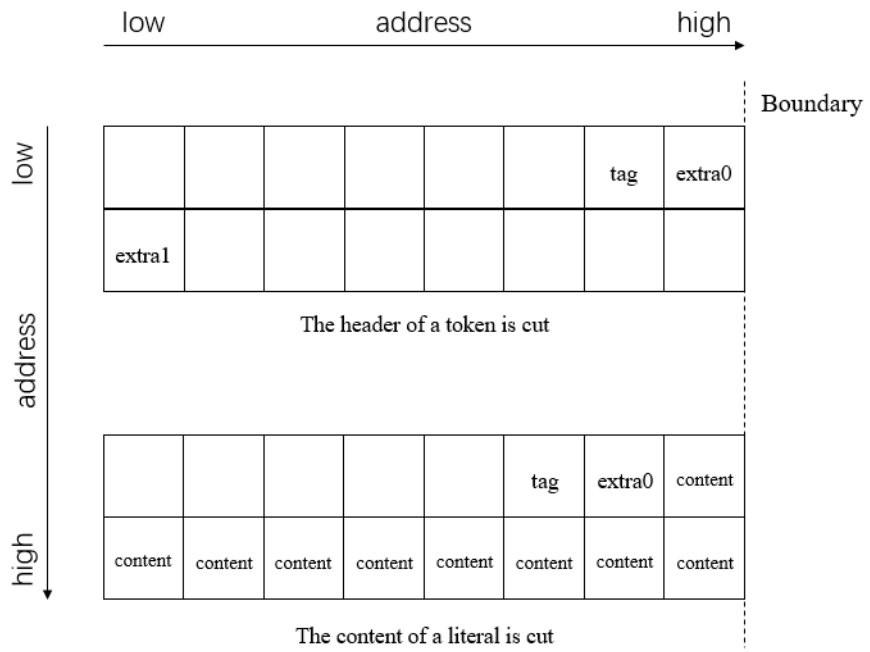

Figure 4.5: Two types of the token being cut

finish_tag_offset" is used to record the position, in the 8 bytes of input data, of the tag byte of unfinished token, which ranges from 0 to 7. "literal_finish" and "header_finish" describe whether the literal content and the header of a token are cut by boundary, respectively. It should be noted that in some cases the value of "unfinish_next_tag", "unfinish_offset" and "unfinish_len" may not be correctly computed. There are only 2 cases that would lead to the wrong computation. The one is that the header of a copy is cut, which leads to wrong computation of "unfinish_offset". The other one is that the header of a literal is cut, which leads to wrong computations of "unfinish_next_tag" and "unfinish_offset". Even so, there is no need to worry about that, all useful information is temporally stored in corresponding signals, and selector0 will continue to compute the correct value of them at next cycle.

Because the minimum length of a token is 2 bytes, general conclusions can be obtained as follows. The second token can only start from one of the bytes after byte0. The third token can only start from one of the bytes after byte2. The fourth token can only start from one of the bytes after byte4. Although byte7 may be the tag byte of the fifth token, the token cannot be emitted to next module. Because the complete information cannot be obtained only from the tag byte. Even if the token is a literal without extra bytes, the literal content itself is still unknown so that the following modules will not know what to write.

Finally, the detailed function of selectors is introduced. When starting to parse a compressed block, the boundary information is initially set to values, which guarantees that the selector0 is not used. The boundary information will be updated before the

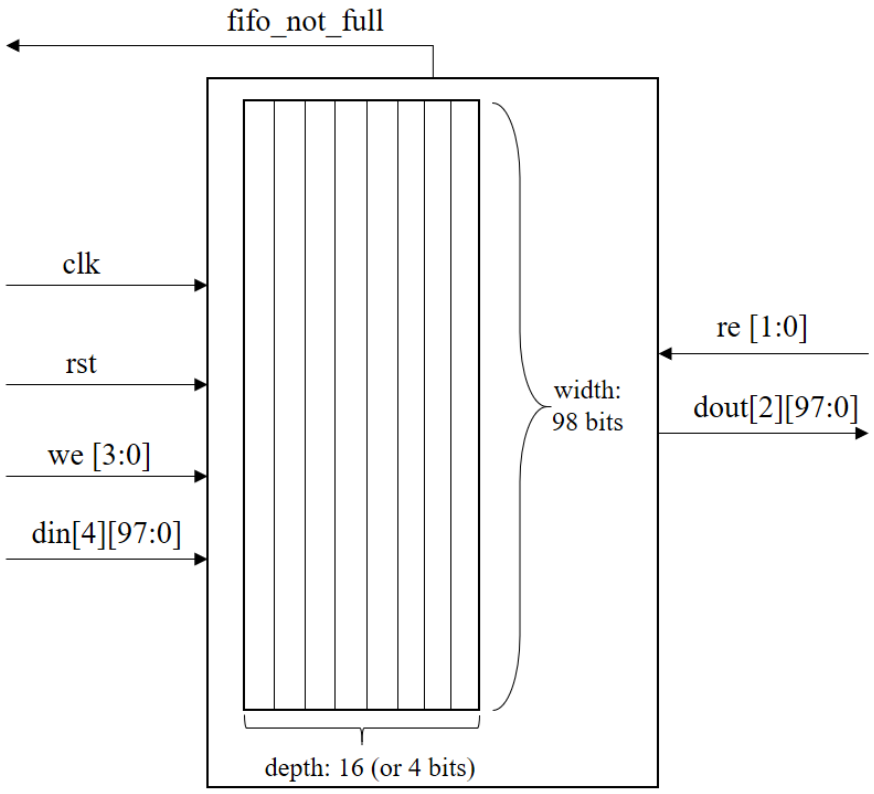

Figure 4.6: I/O ports of FIFO

end of each cycle, thus, if the very first cycle of 8 bytes data can be correctly parsed, then, the following data will also be correctly parsed. selector0 is only used to process the unfinished token, it can recompute the incomplete signals from the boundary information register to obtain a correct value. Each of selectors from selector1 to selector4 will compute and transfer a signal, which indicates whether the next tag byte is still in the current 8 bytes of input data, to the next selector.

4.2.2 FIFO module

The FIFO module is an asymmetrically circular FIFO which has four write ports and two read ports. Because the parser module outputs at most four effectively parsed tokens per cycle, and the following module, the conflict detector module, reads at most two tokens per cycle. Hence, this FIFO has four write ports and two read ports. The FIFO width is 98 bits containing 64-bit wide "din", 16-bit wide "writer_len", 16-bit wide "writer_offset", 1-bit wide "writer_token_type" and 1-bit wide "writer_en". FIFO depth is 16 (4 bits). Figure 4.6 illustrates the I/O ports of the FIFO. The "we" port is connected to the "writer_en" port of the parser module. "din" is an array containing 4 elements and is connected to parser module, besides, each element is 98-bit wide. "re" and "dout" are connected to the conflict detector. "fifo_not_full" equals 1 when internal count of tokens is less than 8.

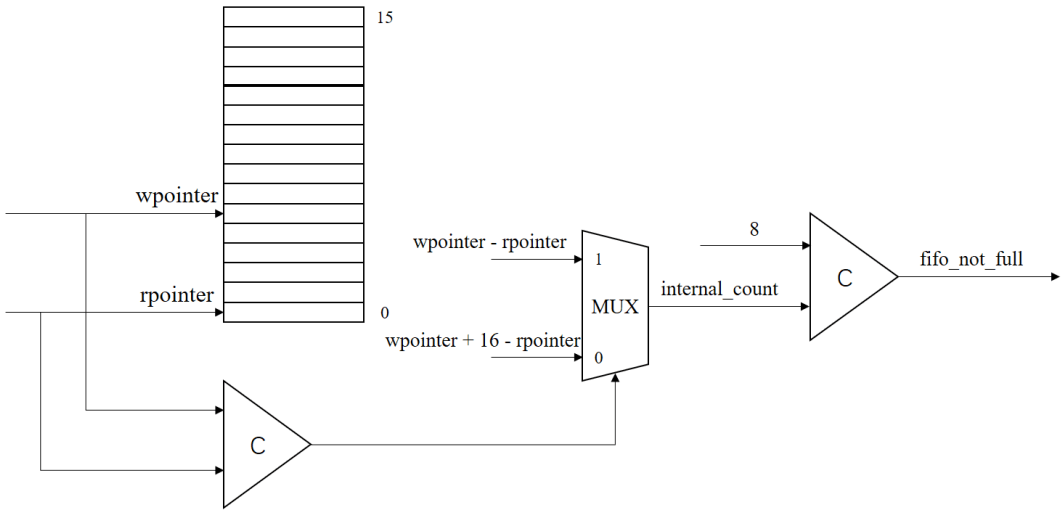

Figure 4.7: Internal structure of FIFO

Although there is more than one read or write port, the read ports or the write ports do have different priorities. That is because, as is mentioned in section 4.2.1, the five selectors also have different priorities, which also affects their outputs.

The FIFO module is a circular FIFO, which means the read pointer, denoted as "rpointer", does not have to be always smaller than the write pointer, denoted as "wpointer". Figure 4.7 shows how the internal count is computed and how the almost full signal is obtained.

"rpointer", "wpointer" and "internal_count" are 4-bit wide. Before the end of each cycle, "wpointer" and "rpointer" may increase by 0∼4 and 0∼2, respectively. If the value of "rpointer" or "wpointer" is "1111B", then it will be 0 after increasing by 1. There is an internal signal named "internal_count" counting the effective number of tokens stored in FIFO. The comparator outputs one if "wpointer" is larger than "rpointer", otherwise, it outputs 0. Next, the MUX will select one of two inputs as output according to the relationship between "wpointer" and "rpointer". The "internal_count" is further compared with 8, if it is less than 8, the comparator outputs 1, otherwise, it outputs 0. To be more precise, it is an almost full signal with a threshold of 8. The threshold value sufficiently considers the latency between FIFO and the modules before FIFO. The FIFO should have sufficient space to store tokens of one extra cycle after "fifo_not_full" becoming 0. In the worst case, assuming that the "internal_count" becomes seven after one cycle, then the FIFO could still be written into by at most eight tokens. If the width of address is n bits, the threshold value can always be set as 2^n-8. Hence, 4 is the minimum value of n, which could also save many resources. "fifo_not_full" is used as the enable signal of the parser module.

For write ports, four groups of input data are always written into the FIFO. However, the value of "wpointer" may increase by 0∼4 only according to "we". "we" is a 4-bit port connected to "writer_en" port of the parser module. From the lowest significant bit (LSB) to the most significant bit (MSB) of "we", only continuous "1" can be computed

into the increment of “wpointer”. For example, if “we” equals “1101B”, then only the first token is valid and “wpointer” is increased by one.

Things are different for read ports, the increment of “rpointer” has to be determined by both “re” and “internal_count”. Even if “re” equals “11”, it does not mean that two valid outputs can be read, because “internal_count” may be less than 2. Therefore, read ports cannot always output two valid tokens stored in the FIFO. If valid tokens cannot be read, the FIFO would output an all-zero signal for that read port to indicate that the output is invalid.

4.2.3 Conflict detector module

The conflict detector detects the type of data dependency, determines whether a write address conflict exists, and emits read and write signals to the BRAM blocks. Figure 4.8 illustrates the I/O ports of the conflict detector.

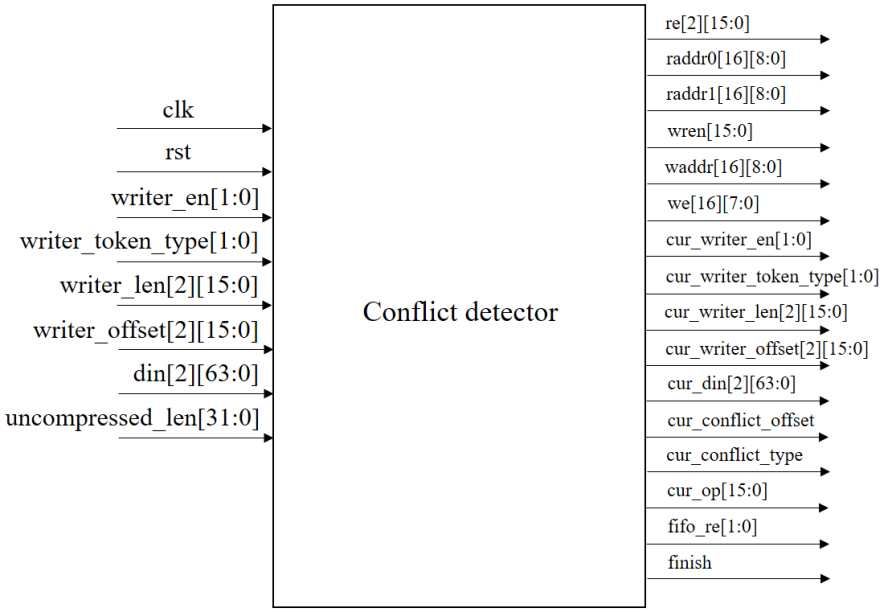

Figure 4.8: I/O ports of conflict detector

As can be seen in Figure 4.8, the ports from “writer_en” to “din” are connected to the FIFO module. These ports contain the information of two tokens. “uncompressed_len” is given by CPU and it means the size of the uncompressed file which can be obtained from the first several bytes of each compressed block. As for output ports, the ports from “re” to “we” are connected to corresponding ports of two BRAM blocks. The ports from “cur_writer_en” to “cur_op” are connected to alignment module. “fifo_re” is connected to the “re” port of FIFO module. “finish” indicates whether the compressed block is completely decompressed. “cur_op” means the start byte address of the first token of two tokens which are going to be emitted to BRAM.

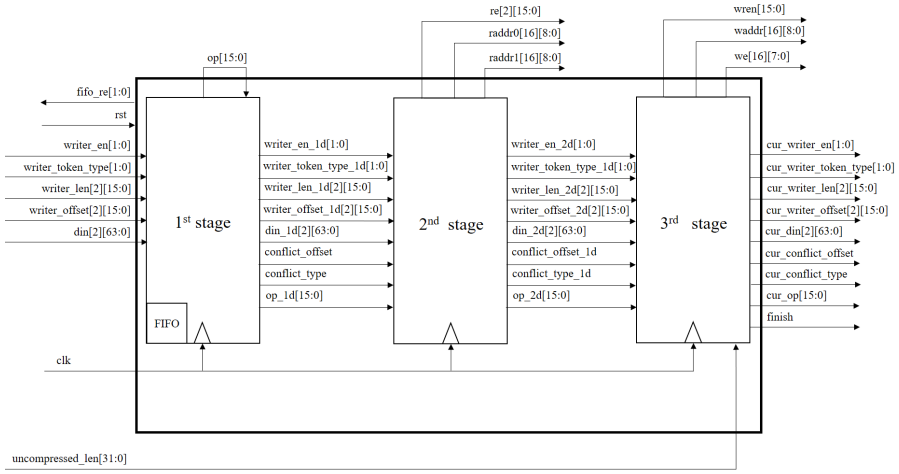

Figure 4.9: Three-stage structure of conflict detector

Normally, a single engine needs to process two tokens per cycle, that is to say, the conflict detector reads two tokens per cycle from the FIFO module. The two tokens have 4 possible combinations, literal/copy, literal/literal, copy/literal and copy/copy. Each literal can emit at most 8 bytes per cycle because the input data of the parser is 8 bytes per cycle, long literals will be cut into many sub-literals. Therefore each copy can write at most 64 bytes. The conflict detector has a three-stage pipeline architecture, which is illustrated in Figure 4.9. The first stage detects data dependencies and write address conflicts, the second stage emits read signals, and the third stage emits write signals. The signals include “1d” and “2d” meaning one cycle delay and two cycles delay, respectively.

The function of the first stage is detecting data dependencies and write address conflicts. There is a special internal FIFO inside this stage. The depth of internal FIFO is 8 (3 bits) and the width of FIFO is 98 bits. The internal FIFO is also a circular FIFO just like the FIFO module. The FIFO is used to store tokens that are stalled due to write address conflicts. Therefore, at the same cycle, the tokens stored in the internal FIFO have a higher priority than those being read from FIFO module. In the internal FIFO, the token that is stored earlier has a higher priority. “internal_count” is the number of effective tokens stored in the internal FIFO.

There are two sources of tokens that can be emitted. The first source is the input tokens of the conflict detector. The second source is the tokens stored in the internal FIFO. Not all the tokens from the two sources are valid. The first stage of the conflict detector selects two tokens from the two sources. If the “internal_count” is larger than one, both tokens are from the internal FIFO, if the “internal_count” is one, the first token comes from the internal FIFO and the second one comes from the input tokens, if the “internal_count” is zero, both tokens are from the input tokens. Next, the exact number of tokens that can be emitted has to be determined, possible values are 0, 1, and 2. 0 means both tokens are invalid. There are two cases for the value of 1. One case is that both tokens are valid. However, the write address conflict exists, then the conflict detector only emits the first one of two tokens and stores the other one into the internal

FIFO. The other case is that only the first token is valid. 2 means that both tokens are valid and there is no write address conflict between them. Only for the value of 2, data dependencies have to be considered.

There are two signals about data dependency, which are "conflict_type" and "conflict_offset". "conflict_type" has 3 possible values which are 0, 1, and 2. The range of "conflict_offset" is from 0 to 63. Figure 4.10 gives examples of three types of data dependency, where "token0" and "token1" represent the first and the second token at the current cycle. The token1 in this Figure is assumed to be a copy, if it is a literal, then the conflict type is 0.

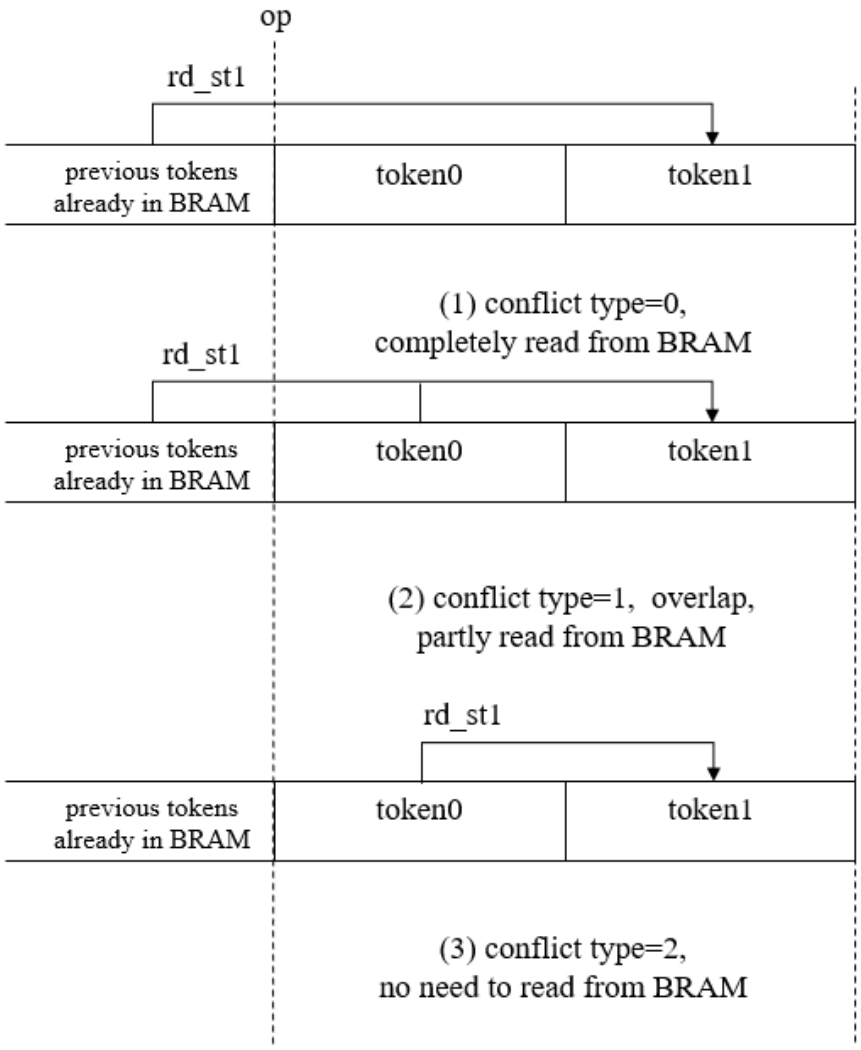

Figure 4.10: Three possible values of conflict type

The arrows in Figure 4.10 indicate the direction of data flow of a copy. The data of previous tokens has already be written into BRAM. When the conflict type is 0, the data dependency does not exist between token0 and token1. If conflict type is 1, then it

means that the token1 has to read data from both previous tokens and token0. If conflict type is 2, namely, the token1 only reads data from token0, thus, a read operation for BRAM is not needed.

If the BRAM start read address of token1 is denoted as "rd_st1", then the "conflict_offset" is equal to the absolute value of the difference between "op" and "rd_st1", where "op" is the start write address of the token0, and "op" is updated before the end of each cycle. If the token1 is a literal, the "conflict_offset" is also 0. Hence, if both "conflict_type" and "conflict_offset" are known, then everything about data dependency can be determined so that related operations can be performed.

The second stage transfers intermediate signals to the third stage and emits three read signals, which are "raddr0", "raddr1" and "re", to the two BRAM blocks.

The third stage transfers intermediate signals to outputs that are connected to the next module (alignment module), in addition, it also emit three write signals, which are "waddr", "wren" and "we", to the two BRAM blocks. Finally, "finish" shows whether the compressed block is completely decompressed according to the comparison result of "op_2d" and "uncompressed_len".

4.2.4 Alignment module

The alignment module is a combinational module. It receives signals from both the conflict detector and BRAMs, and it only has one output which is connected to the data input port of BRAMs. The I/O ports of the alignment module can be seen in Figure 4.11.

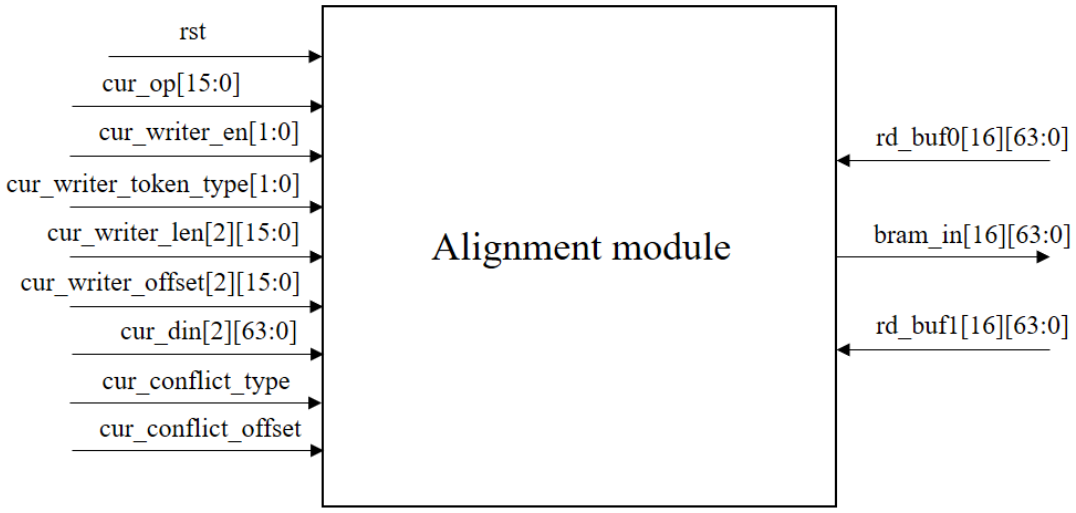

Figure 4.11: I/O ports of the alignment module

As is shown in Figure 4.11, the ports excluding "rst" at the left side are connected to the conflict detector, the "rd_buf0" and "rd_buf1" are connected to the data output port of two BRAM blocks, respectively. The "bram_in" is connected to the data input port of both BRAM blocks.

Specifically, because the BRAM works in WRITE_FIRST mode, only the intra-stage data dependency exists. The design of the alignment module is illustrated in Figure 4.12, where "CL" means computational logic.

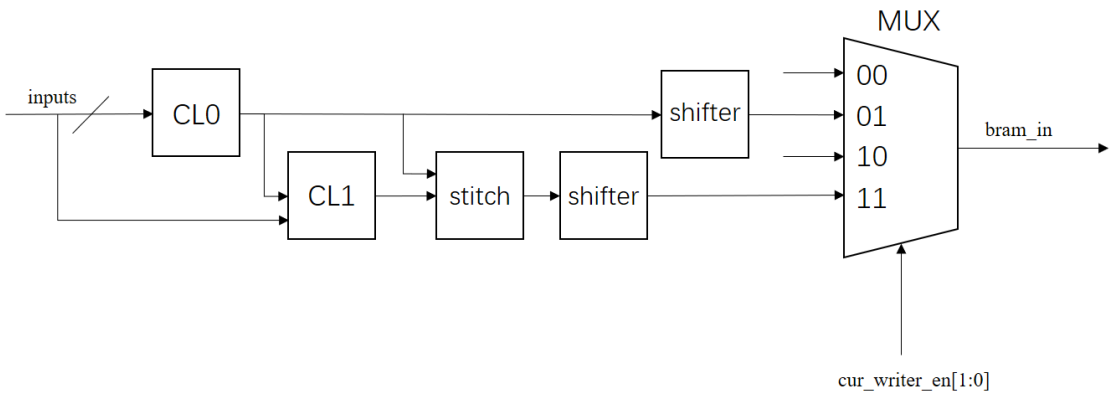

Figure 4.12: Internal structure of the alignment module

The "inputs" contain all the information of two tokens. "cur_writer_en" indicates whether two tokens are valid or not. "CL0" and "CL1" are computation units to obtain the content of token0 and token1, respectively. The function of "stitch" is integrating the content of token0 and token1 into one buffer. The "shifter" shifts the data in the write buffer to fit the data input port of BRAM blocks. Finally, the MUX will select one of inputs as the output according to "cur_writer_en". For the cases that "cur_writer_en" equals "10" or "00", "bram_in" remains unchanged. The order of "CL0" and "CL1" shows the data dependency between token0 and token1.

4.2.5 BRAM module

In this thesis project, 16 RAMB36 configured as SDP RAMs form a BRAM block whose data port width is 128 bytes, and each SDP RAM supports a 64-bit port width. Although each RAM occupies 36Kbit, the actual space that can be used to store data is 32Kbit, yielding 512 (depth) x 64 (width).

As is discussed in section 3.4.2, a Xilinx KU15P does not support WRITE_FIRST mode for a common clock. Thus, an improvement method that we add a wrapper for each SDP RAM to make it work just like in WRITE_FIRST mode in the case of common clock is proposed. The BRAM primitive itself still works in the READ_FIRST mode. The modified SDP RAM is illustrated in Figure 4.13.

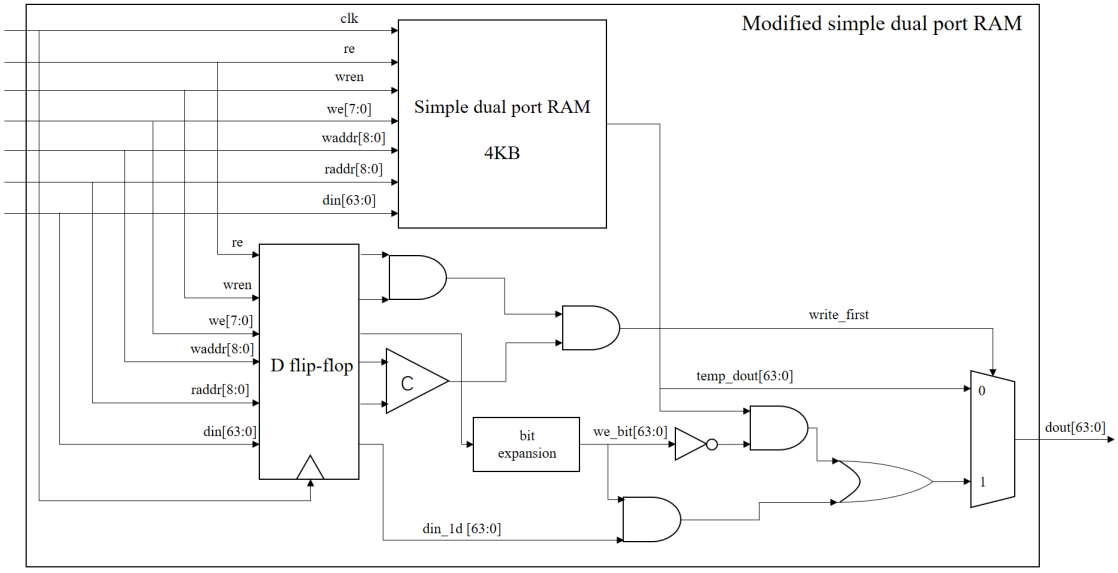

Figure 4.13: Modified simple dual port RAM

The D flip-flop maintains the input signals unchanged after the rising edge of clock. The comparator denoted by “C” outputs “1” only when its inputs are equal. Hence, the “write_first” indicates whether the address collision exists. Bit expansion expands “we” up to 64 bits. Each bit of “we” shows whether the corresponding byte of “din” is written into BRAM. Therefore, each bit becomes 8 bits after bit expansion, the value of the 8 bits can only be 00H or FFH according to the value of the bit before bit expansion. The “temp_dout” always gives the old data because the SDP RAM works in READ_FIRST mode. The output of OR gate gives the newly written data, and then the MUX selects one of inputs according to “write_first”. In conclusion, the modified SDP RAM works just like in WRITE_FIRST mode. Figure 4.14 shows the structure of one BRAM block. This Figure shows that one BRAM block contains 16 modified SDP RAM.

There are two identical BRAM blocks which share their data input, write enable, byte-wide write enable and write address. However, they have independent data output, read enable and read address. The interconnection of two BRAM blocks can be seen in Figure 4.15. Some ports contain two brackets with each having a number in it. The first number indicates the number of elements of the array, while the second one is the size of an element.

As can be seen in Figure 4.11 and Figure 4.2, “wren”, “we” and “waddr” of both BRAM blocks are connected to the third stage of the conflict detector module. The “re” and “raddr” ports of two BRAM blocks are independent and are connected to the second stage of the conflict detector. The “din” is connected to the “bram_in” port of alignment module. The “dout” port of BRAM block0 and BRAM block1 are connected to the “rd_buf0” and “rd_buf1” of alignment module, respectively.

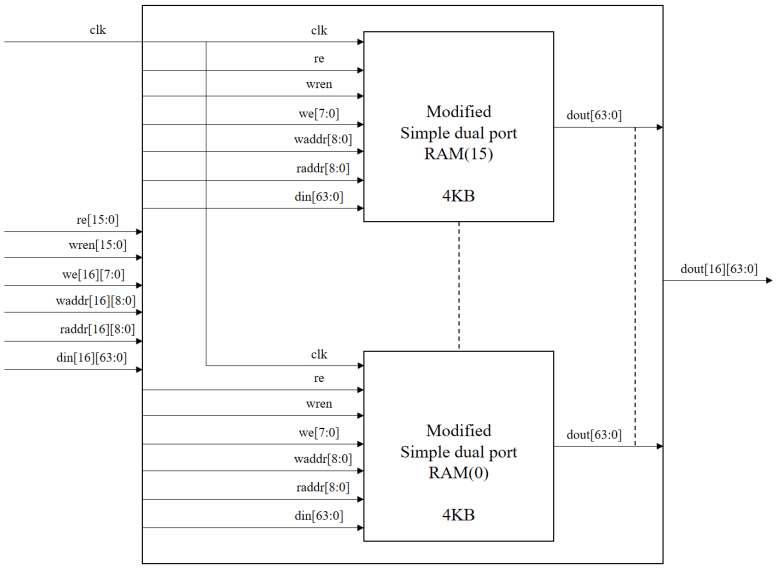

Figure 4.14: BRAM block

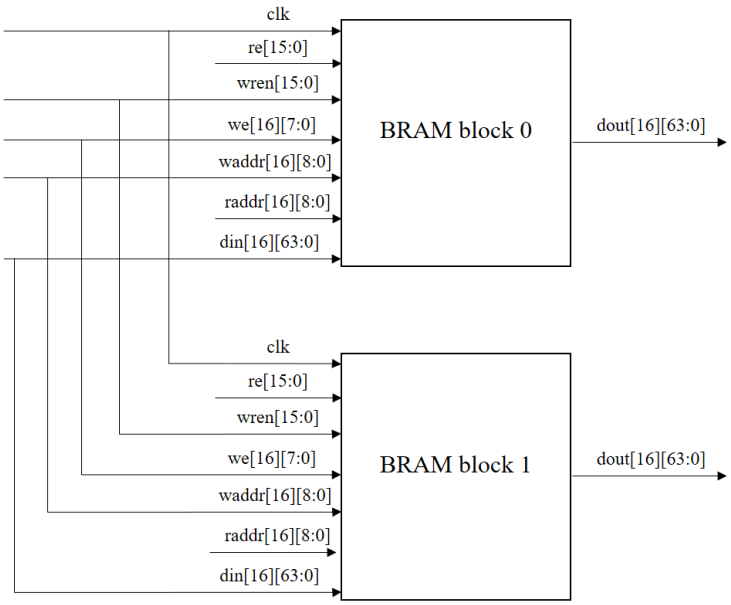

Figure 4.15: Interconnection between BRAM blocks

5 Experimental Results

In this chapter, some extra modules that are only used for simulation are introduced. Simulation and synthesis results are also presented. The statistics and experimental results of several benchmarks are shown and discussed. The type of FPGA is the Xilinx KU15P.

5.1 Experiment setup

5.1.1 Experiment platform

Xilinx KU15P FPGA supports 34.5Mbit Block RAM resource, 522720 CLB LUTs, and 1045440 CLB registers. Modelsim SE-64 10.4 is used for behavior simulation. The clock period of behavior simulation is 10ns. Vivado 2017.1 is used for synthesis and implementation. The initial clock frequency is constrained to 200MHz. The flatten hierarchy is configured as none, and other settings of synthesis are default values. For implementation, only the opt_design is enabled.

5.1.2 Benchmarks

The behavior simulation uses 8 benchmarks. "SampleTextFile_50KB.txt", "Sample-SQL-File-500-Rows.sql", "SampleCSVFile_11kb.csv" and "SampleXLSFile_38kb.xls" are from [26]. "embl.txt", "HIV_Dataset.txt", "FastaSeqCL.txt", and "Beckman_Sample.txt" are from [27]. "embl.txt" is a bionumerics sample text file for import. "Beckman_Sample.txt" is a VNTR sample peak table. Table 5.1 shows the statistics of these benchmarks.

Table 5.1: Basic information of benchmarks

Benchmark	compressed size (byte)	uncompressed size (byte)	compression ratio (x)	num_copy	num_literal	num_token	av_size_input	av_size_output
SampleXLSFile_38kb.xls	18618	38912	2.090	3804	2060	5864	3.175	6.636
SampleTextFile_50KB.txt	19335	50537	2.613	6931	658	7589	2.548	6.659
Sample-SQL-File-500-Rows.sql	25775	43511	1.688	4621	1584	6205	4.154	7.012
SampleCSVFile_11kb.csv	6535	10998	1.683	833	524	1357	4.816	8.105
Beckman_Sample.txt	12514	26770	2.139	2940	1173	4113	3.042	6.509
embl.txt	19647	63003	3.207	6291	729	7020	2.799	8.975
HIV_Dataset.txt	23343	60475	2.591	10493	387	10880	2.145	5.558
FastaSeqCL.txt	10202	21263	2.084	4514	198	4712	2.165	4.513

5.2 Modules used for behavior simulation

In this section, the modules that are used for simulation are introduced. The interconnection among these modules and hardware modules is shown in Figure 5.1.

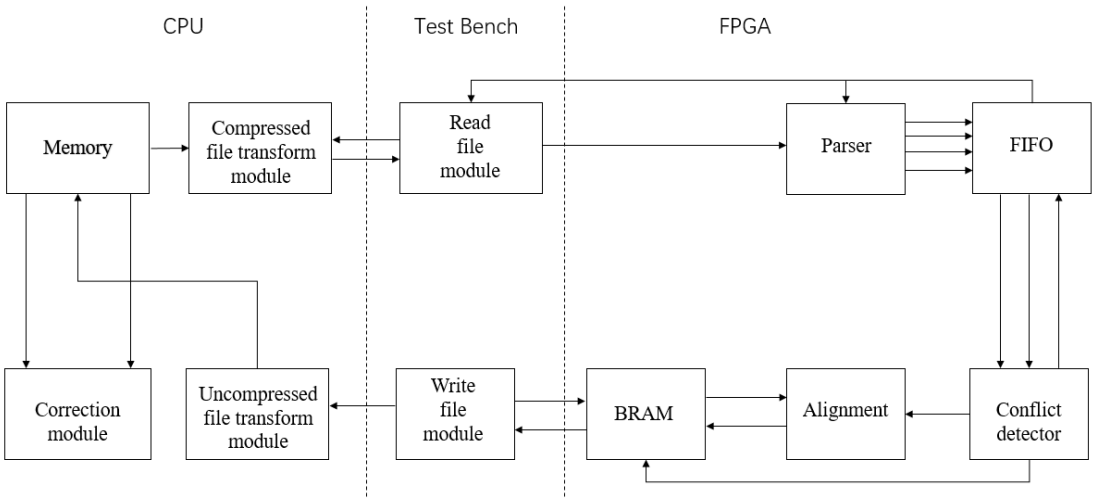

Figure 5.1: Interconnection between hardware and simulation modules

Specifically, first, raw data (uncompressed or original file) is compressed by the Snappy algorithm on the CPU. Then, the compressed file is transformed to a text file, where each character represents a bit in the compressed file. The reason why this module exists is that VHDL can read text files through the TEXTIO package. Next, the read file module reads in transformed data from the output file of the previous module line by line. Each line represents 8 bytes of compressed data. After the whole compressed block is decompressed, the write file module reads data from BRAM module line by line and writes it into the output file. Each line occupies 128 bytes. The next module after write file module transforms each character into a bit. Finally, a decompressed file in binary code is obtained. The correction module is used to verify the correctness of decompression.

5.2.1 Compressed file transform module

This module is implemented by software on the CPU. First, it reads the compressed file and parses the first several bytes of the input file, by which the uncompressed file size and input file size (excluding first several bytes that represents uncompressed file size) can be obtained, and these two variables are noted as uncompressed_len and ip_limit, respectively. Both of them are written to the first line of the output file (text format). From the very first byte that is not related to the uncompressed file size until the end of the input file, every 8 bytes of input data is written as a line into the output file. There is a transform inside this module; each binary bit is transformed into an ASCII code, “0” or “1”, and then stored in output text file. After every 8 bytes are transformed, an extra line feed character is written, because the following module can only read data in

units of a line. In this way, each line of text file contains 64 characters. An example of output format of this module is illustrated in Figure 5.2.

Figure 5.2: File format of the output of compressed file transform module

5.2.2 Read file module

This module is designed via VHDL using TEXTIO package, after the rising edge of the clock, the module will read the input file (text format) line by line and transfer the data to next module once the enable signal is set. At the first cycle, the first line indicating uncompressed_len and ip_limit is transferred to two different ports of the following module. After the first cycle, the output is the compressed data itself. Figure 5.3 shows the I/O ports of this module.

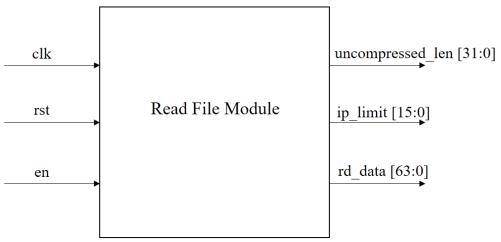

Figure 5.3: Read file module

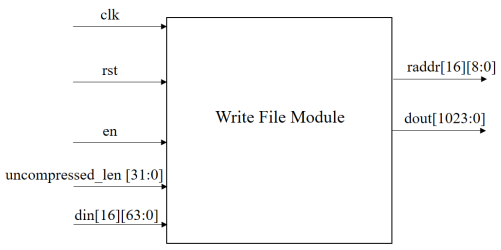

Figure 5.4: Write file module

5.2.3 Write file module

The write file module is also designed via VHDL using TEXTIO package. This module is used to read uncompressed data from the BRAM block and write into a text file. Once all the input data is decompressed, the conflict detector produces a flag named “finish” which is used as the enable signal of the write file module. The write file module reads 128 bytes from the BRAM block per cycle and writes them into a line of the text file.

"raddr" increases by one before the end of each cycle in order to get the next line of uncompressed data from the BRAM block. There is a transform in this module that transforms each bit of 128 bytes data into an ASCII code of a character, "0" or "1", and then stores this in the text file, which has the similar format shown in Figure 5.2. The difference is that each line of the output file of this module contains 1024 characters. Figure 5.4 illustrates the I/O ports of this module.

5.2.4 Uncompressed file transform module

Each line of the text file contains 128 bytes, but they are displayed and stored as ASCII codes. Therefore, this module transforms ASCII code into binary code, and every 8 characters are transformed into a byte. This procedure is an inverse process of the compressed file transform module. The output file of this module should be the uncompressed file itself. This module is implemented via C language on the CPU.

5.2.5 Correction module

This module is also implemented on the CPU. It can read two binary files. The one is the original file before compression. The other one is output file of the uncompressed file transform module. It then compares each corresponding byte of the two files until the end of the original file. Once a mismatch is found, then a conclusion that decompression is wrong can be drawn. All of the files listed in the Table 5.1 are successfully decompressed by our design.

5.3 Behavior simulation results

Each benchmark is compressed, transformed, parsed, decompressed, then read out. The simulation results in this section shows how many cycles the decompression procedure takes and how many cycles the reading out procedure takes. In principle, in our design, each BRAM block contains 512 lines with each line having the port width of 128 bytes. Hence, the reading out procedure would take at most 512 cycles. The simulation period in this section is 10ns.

The clock period of the behavior simulation is 10ns, and the system in behavior simulation starts to work at 25ns. Hence, the input and output throughput and be computed, and is shown in Table 5.2.

"num cycle decompression" is the number of cycles that the decompression procedure spends, which is denoted as T1. However, the uncompressed data has to be read out from BRAM blocks to memory. Thus, "num cycle readout" is the number of cycles that the decompression procedure and reading out procedure consume, which is denoted as T2. The input and output throughput are computed according to T2, because T2 represents the whole procedure.

As can be seen from the Table 5.2, the input throughput of a single engine varies from 3.9 bytes/cycle to 6.3 bytes/cycle, and the output throughput of a single engine varies from 8.3 bytes/cycle to 15.0 bytes/cycle. If the working frequency of FPGA could reach up to 200MHz, then, the corresponding input throughput and output throughput

Table 5.2: Behavior simulation results of benchmarks

Benchmark	compressed size (byte)	uncompressed size (byte)	num cycle decompression	num cycle readout	input throughput (B/cycle)	output throughput (B/cycle)
SampleXLSFile_38kb.xls	18618	38912	3340	3647	5.1	10.7
SampleTextFile_50KB.txt	19335	50537	3891	4288	4.5	11.8
Sample-SQL-File-500-Rows.sql	25775	43511	3857	4199	6.1	10.4
SampleCSVFile_11kb.csv	6535	10998	945	1033	6.3	10.6
Beckman_Sample.txt	12514	26770	2231	2443	5.1	11.0
embl.txt	19647	63003	3710	4205	4.7	15.0
HIV_Dataset.txt	23343	60475	5473	5948	3.9	10.2
FastaSeqCL.txt	10202	21263	2394	2563	4.0	8.3

are 0.73∼1.17 GB/s and 1.55∼2.80 GB/s, respectively. If an FPGA could support 16 such engines, then, the input throughput and output throughput of the whole system are 11.68∼18.72 GB/s and 24.8∼44.8 GB/s, respectively. Hence, a filter used to reduce the output bandwidth required is desirable.

5.4 Synthesis results

Synthesis results are obtained via Vivado, the target FPGA is Xilinx KU15P. The top module contains the parser, FIFO module, conflict detector, alignment module, and two BRAM blocks. These modules consume 89729 CLB LUTs and 8901 CLB registers, which is shown in Table 5.3.

Table 5.3: Post-synthesis utilization

Site Type	Used	Available	Utilization (%)
CLB LUTs	89729	522720	17.166
CLB Registers	8901	1045440	0.851
BlockRAM	32	984	3.252

The synthesis results also give a timing summary. The architecture can work at 145MHz. According to the resource utilization, Xilinx KU15P FPGA could support about six decompression engines. The input and output throughput of a single engine based on synthesis results are 0.53∼0.85 GB/s and 1.12∼2.03 GB/s, respectively.

5.5 Place and Route (Implementation)

Without a wrapper around our module the I/O utilization exceeds the upper limit of a KU15P. To obtain a result for Place and Route (P&R), we add another wrapper to reduce the I/O ports, which introduces some extra resource consumption. The corresponding resource utilization is shown in Table 5.4.

Table 5.4: Post-implementation utilization

Site Type	Used	Available	Utilization (%)
CLB LUTs	91089	522720	17.42596419
CLB Registers	8901	1045440	0.851411846
BlockRAM	32	984	3.25203252

The implementation results also give a timing summary. The architecture can work at 140MHz. Xilinx KU15P FPGA could support about six decompression engines. The input and output throughput of a single engine based on implementation results are 0.51∼0.82 GB/s and 1.08∼1.96 GB/s, respectively. After Place and Route, the resource utilization remains unchanged, while the working frequency gets a little bit worse than synthesis results.

In conclusion, Table 5.5 indicates timing summary of behavior simulation, synthesis, and implementation results.

Table 5.5: Timing summary

Type	Frequency (MHz)	input throughput (B/cycle)	output throughput (B/cycle)	input throughput (GB/s)	output throughput (GB/s)
Simulation	200	3.9 ∼ 6.3	8.3 ∼ 15	0.73 ∼ 1.17	1.55 ∼ 2.80
Synthesis	145	3.9 ∼ 6.3	8.3 ∼ 15	0.53 ∼ 0.85	1.12 ∼ 2.03
Implementation	140	3.9 ∼ 6.3	8.3 ∼ 15	0.51 ∼ 0.82	1.08 ∼ 1.96

“input throughput (B/cycle)” indicates the input throughput in the unit of bytes/cycle, which comes from the behavior simulation. Other tabs have the similar definitions. The left and right values are the lowest and the highest values, respectively, of corresponding tabs among the eight benchmarks.

6 Conclusion and Future Work

6.1 Conclusion

In this thesis, our goal is to build a Snappy decompressor trying to make use of the input bandwidth of a new class of interfaces like OpenCAPI. We implement a single Snappy decompression engine which can process at most two tokens per cycle. The design uses multiple decoders to handle an uncertain token boundary, and is implemented with a deep pipeline to obtain a high degree of parallelism. Correct uncompressed test files are obtained through behavior simulation. The input throughput in for one engine reaches 3.9∼6.3 bytes per cycle. The output throughput reaches up to 8.3∼15 bytes per cycle. According to synthesis results, a single engine architecture consumes 89729 LUT and 128KB BRAM resource and works at 145MHz. According to implementation results, a single engine architecture consumes 91089 LUT and 128KB BRAM resource and works at 140MHz. According to either the post-synthesis utilization or the post-implementation utilization, Xilinx KU15P FPGA can support approximately six such decompression engines. We draw the following conclusions.

1. Based on the experimental results, our design is suitable to implement on an FPGA. On a single core of Core i7, the decompression rate can reach up to 500 MB/s. The input throughput of our single-engine Snappy decompressor can reach up to 0.82 GB/s at the frequency of 140MHz. The input throughput can be further improved after frequency optimization. Furthermore, because we can place more engines in an FPGA, a multi-engine architecture in an FPGA is more competitive when compared with a similar size multi-core CPU. The FPGA is also more power efficient than the CPU.

2. The decompression performance is affected by the data dependency between tokens. A conflict detector is used to detect the data dependency between two tokens within the same cycle, and forwarding the data of the first token to the next token if necessary.

3. Address collisions frequently occur due to the characteristics of the Block RAM. We gather the statistics and find that the frequency of occurrence of address collisions is dependent on the port width (granularity) of each Block RAM primitive. The number of address collisions gets smaller with the decreasing granularity. Hence, we adopt two 64KB Block RAMs to totally eliminate read address collisions. We temporarily store or stall the token having the write address collision, because the frequency of occurrence of write address collisions is far lower than that of read address collisions.

4. The critical paths occur in the cascading combinational logic, which limits the working frequency, one of which is the multiple cascading selectors in the parser. The other one is the long combinational logic connected to Block RAMs to solve the data dependency. The combinational logic also consumes the most part of LUT utilization.

6.2 Future work

This section introduces several possible improvements for our design.

The first one is that the input of a single decompression engine is 8 bytes per cycle which contain at most four tokens. Thus, as a next step, the design could be enhanced to process at most four tokens per cycle.

The second possible improvement is increasing the working frequency from 140MHz to the expected frequency at 200MHz. During place and route, the working frequency of the second stage of the parser module itself is 140MHz, which means the critical path of the parser is too long. Hence, the next step is to improve the frequency of the parser.

The architecture consumes 91089 (17.43%) LUT resource of a Xilinx KU15P FPGA, which means KU15P could only support at most six engines. To increase the number of engines, we could either choose another type of FPGA with larger LUT resource or optimize our design to reduce the resource usage.

In our design, the benchmarks that we use are not Parquet files, because a Parquet file cannot be directly compressed or decompressed via the standard Snappy algorithm. Each page of Parquet file should be independently encoded and/or compressed. This kind of operations is based on the database platform. Therefore, we could make Parquet file as input file in the next step.

Finally, the whole system that we design is not just a single decompression engine. Thus, we can put everything together and implement an architecture including multiple decompression engines, filter, CRC32 modules, and arbiters. Moreover, the hardware architecture would also be combined with OpenCAPI.

Bibliography

[1] BERTEN, "Gpu vs fpga performance comparison," http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf, accessed Dec 17, 2017.

[2] Apache, "Apache parquet format," https://github.com/apache/parquet-format, apache License, Version 2.0.

[3] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, "Massively-parallel lossless data decompression," in *Parallel Processing (ICPP), 2016 45th International Conference on*. IEEE, 2016, pp. 242–247.

[4] K. B. Agarwal, H. P. Hofstee, D. A. Jamsek, and A. K. Martin, "High bandwidth decompression of variable length encoded data streams," Aug. 12 2014, uS Patent 8,804,852.

[5] J. Lee, H. Kim, S. Yoo, K. Choi, H. P. Hofstee, G.-J. Nam, M. R. Nutter, and D. Jamsek, "Extrav: boosting graph processing near storage with a coherent accelerator," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1706–1717, 2017.

[6] V. Gopal, J. D. Guilford, K. S. Yap, S. M. Gulley, and G. M. Wolrich, "Systems, methods, and apparatuses for decompression using hardware and software," Apr. 4 2017, uS Patent 9,614,544.

[7] Xilinx, "ug573," https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf, accessed Dec 17, 2017.

[8] opencapi consortium, "Opencapi overview," http://opencapi.org/wp-content/uploads/2016/09/OpenCAPI-Overview.10.14.16.pdf, accessed Dec 17, 2017.

[9] J. Fang, Y. Mulder, K. Huang, Y. Qiao, X. Zeng, P. Hofstee, J. Lee, and J. Hidders, "Adopting opencapi for high bandwidth database accelerators," in *Proc. 3rd International Workshop on Heterogeneous High-performance Reconfigurable Computing*, Denver, USA, November 2017.

[10] L. P. Deutsch, "Gzip file format specification version 4.3," 1996.

[11] V. Gopal, S. M. Gulley, and J. D. Guilford, "Technologies for efficient lz77-based data decompression," Sep. 24 2014, uS Patent 20160085555A1.

[12] Wikipedia, "Hardware acceleration," https://en.wikipedia.org/wiki/Hardware_acceleration, accessed Dec 17, 2017.

[13] Revolvy, "Gpu," https://www.revolvy.com/main/index.php?s=Graphics%20processing%20unit, accessed Dec 17, 2017.

[14] Y. Mulder, "Feeding high-bandwidth streaming-based fpga accelerators," Master's thesis, Delft University of Technology, Delft, 2018.

[15] Apache, "Apache parquet configurations," http://parquet.apache.org/documentation/latest/, copyright 2014 Apache Software Foundation.

[16] Wikipedia, "Column oriented dbms," https://en.wikipedia.org/wiki/Column-oriented_DBMS, accessed Dec 17, 2017.

[17] Apache, "Apache thrift," https://thrift.apache.org/, apache License, Version 2.0.

[18] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010.

[19] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[20] Google, "Snappy," https://github.com/google/snappy, accessed Dec 17, 2017.

[21] L. P. Deutsch, "Deflate compressed data format specification version 1.3," 1996.

[22] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[23] NCBI, "test file," ftp://ftp.ncbi.nlm.nih.gov/ncbi-asn1, accessed Dec 17, 2017.

[24] Xilinx, "Blockram," https://www.xilinx.com/products/intellectual-property/block_ram.html, accessed Dec 17, 2017.

[25] Wikipedia, "Cyclic redundancy check," https://en.wikipedia.org/wiki/Cyclic_redundancy_check#Standards_and_common_use.

[26] sample videos, "benchmark2," http://www.sample-videos.com/download-sample-text-file.php, accessed Dec 17, 2017.

[27] applied maths, "benchmark1," http://www.applied-maths.com/download/sample-data, accessed Dec 17, 2017.