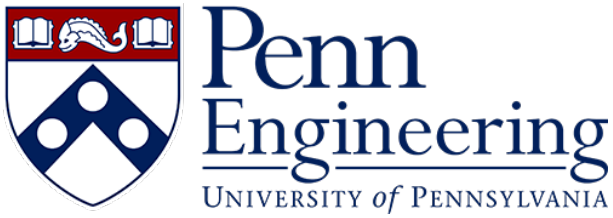

University of Pennsylvania

Department of Electrical and System Engineering

ESE5320: System-on-a-Chip Architecture

Deduplication and Compression Project Final Report

Team: Truong (Winston) Nguyen, Ke Liu, Willie Gu, and Kun Wang

Due: 9 Dec 2024

Contents

1 Single ARM Processor Mapped Design

1.1 Design Process

The ARM implementation is a streamline of functions as shown in Figure 1.

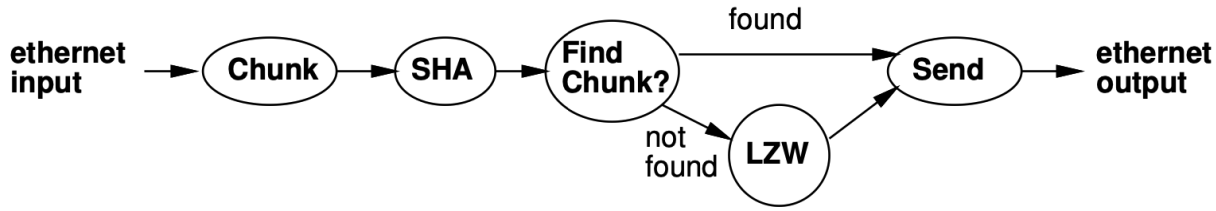

Figure 1: Project Functionality Streamline

As shown in Code 9, the project is designed step-wise as shown below:

1. **Input Data Buffering**:
    - Input data is received in real-time through Ethernet and stored in a vector of buffers using project-provided APIs. We have added another function to correct the endianess of the received data. Data packets with maximum of 70MB are accumulated to fill the `file_buffer` before processing.
2. **Compression Pipeline**:
    - Upon buffer completion, a sequential compression pipeline is executed:
        - **Content-Defined Chunking (CDC)**: The CDC function processes the input buffer using a rolling hash with a window size of 12 (`WIN_SIZE`). A chunk boundary is identified when the hash matches the target defined by the modulus (`MODULUS`) or exceeds `MAX_CHUNK_SIZE`. The end index is recorded, and all boundaries are stored in a vector for further processing.
        - **SHA-256 Hashing**: Computes SHA256 fingerprints for each chunk. This process is further accelerated by exploring SIMD using NEON Instrinsics.
        - **Deduplication**: The deduplication function checks each chunk's SHA256 fingerprint against the `chunk_hash_map`. If not found, the fingerprint is added with the current `lzw_chunk_index`, which is then incremented, and the function returns `-1` to indicate a new chunk. If found, the function returns the corresponding index, identifying a duplicate chunk.
        - **LZW Compression**: Compresses unique chunks when called based on up-

dated chunk boundaries.

- The final compressed data is bit-packed and written to an output file.

3. **Looping Process**:
   - The program continues iterating through all chunks received via Ethernet until all input is processed.
4. **Ethernet Output**:
   - The Ethernet Output after processing each packet stringently follows the design rules. The headers are identified and appended before each chunk output to adapt to the provided decoder.

1.2 Key Parameters in the Solution

These are parameters in the solution for each function.

1. **Content-defined Chunking (CDC)**:
   - `MAX_CHUNK_SIZE: 1024 * 5` (value: 5120 bytes)
     Maximum size of a chunk. A chunk cannot exceed this size.
   - `MIN_CHUNK_SIZE: 0` (value: 0 bytes)
     Minimum size of a chunk. A chunk must be at least this size.
   - `MODULUS: 64` (value: 64)
     Divisor used for computing hash values to check for chunk boundaries.
   - `TARGET: 3` (value: 3)
     Target hash value used for determining chunk boundaries.
   - `PRIME: 3` (value: 3)
     A small prime number used in the rolling hash calculation.
   - `WIN_SIZE: 12` (value: 12 bytes)
     Size of the sliding window used for hashing.
2. **Hashing function (SHA256)**:
   - **SHA256_BLOCK_SIZE:** `32` (value: 32 bytes)
     The size of the output digest produced by SHA-256.
3. **Deduplication function**:

- **sha_fingerprint:** `SHA256Hash` (32 bytes)
  The SHA-256 hash of the current chunk used as a unique identifier.
- **chunk_hash_map:** `std::unordered_map<SHA256Hash, int64_t, SHA256HashHash>`
  A hash map that stores mappings from SHA-256 hashes of chunks to their unique indices. Used for detecting duplicates.
- **lzw_chunk_index:** `int64_t` (integer)
  A reference to the current index for new chunks being processed. Incremented for each new chunk added.

4. **LZW function**:
   - **CODE_LENGTH:** `13` (value: 13 bits)
     Length of each LZW code word in bits.
   - **CODE_MASK:** `((1 << CODE_LENGTH) - 1)` (value: 8191)
     Bitmask used to extract the least significant 13 bits of a code word.
   - **MAX_DICT_SIZE:** `(1 << CODE_LENGTH)` (value: 8192)
     Maximum size of the LZW dictionary, determined by the code length.
   - **INPUT_SIZE:** `8192` (value: 8192 bytes)
     Maximum size of the input data buffer for compression.
   - **OUTPUT_SIZE:** `40960` (value: 40960 bytes)
     Maximum size of the output data buffer for compressed data.

1.3 Performance Achieved

The following performance achieved by using default `-s` parameter at the value of 5 and and `-b` at the value of `BLOCKSIZE` = 8192.

- **Throughput**: The throughput of the single ARM implementation was calculated as:

$$\text{Throughput (Mb/s)} = \frac{\text{Bytes processed (in Mega bits)}}{\text{Total compression latency (in seconds)}}$$

  As shown in Figure 2, for the test case of **LittlePrice.txt**, the overall throughput achieved was **96 Mb/s**.

- **Compression Ratio**: Using the `LittlePrince.txt` test file:

$$\text{Compression Ratio} = \frac{\text{Original File Size}}{\text{Compressed File Size}} = \frac{14248B}{5571B} = 2.558$$

```
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./encoder_sw out.bin
Using out.bin as output file name.
setting up sever...
server setup complete!
Original Size: 14248 bytes
CDC function called: chunk_count = 278
Compressed Size: 5571 bytes
Compression Ratio (original size/ compressed size): 2.55753

Total chunk size as dedup input: 11022 bytes in 218 chunks.
Deduplication Contribution (dedup headers in Compressed size): 872 bytes

Total chunk size as LZW input: 3226 bytes in 60 chunks.
LZW Contribution (LZW header + LZ-compressed content in Compressed size): 4699 bytes
Compressed data size: 5571 bytes.

All data written successfully.
Written file with 5571 bytes
--------------- Key Throughputs ---------------
Input Throughput to Encoder: 2229.51 Mb/s. (Latency: 1.999e-05s).
-----------------------------------------------
Total Data Processed (input original size): 113984 bits.
Total Time Taken of Encoder: 0.00118589 seconds or 1.18589ms.
Overall Throughput: 96.1168 Mb/s.
-----------------------------------------------

CDC throughput 641.802 Mb/s.
SHA throughput 530.503 Mb/s.
Dedup throughput 748.859 Mb/s.
LZW throughput 40.2483 Mb/s.

Average time of CDC: 0.1776 (ms).
Average time of SHA: 0.000772878 (ms).
Average time of Dedup: 0.000547519 (ms).
Average time of LZW: 0.010687 (ms).

Total time of CDC: 0.1776 (ms).
Total time of SHA: 0.21486 (ms).
Total time of Dedup: 0.15221 (ms).
Total time of LZW: 0.64122 (ms).
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./decoder out.bin out.txt
Decoding complete.
Total chunks in header: 278
Successfully processed chunks: 278
Failed chunks: 0
All chunks were successfully processed!
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# diff out.txt littleprince.txt
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1#
```

Figure 2: Software throughput with LittlePrince

1.4 Characterization and Breakdown of Time

Task	Time Spent (days)
CDC Optimization	1
SHA/Deduplication Optimization	3
LZW Optimization	7
Application Integration	2
Debugging	7
Testing and Validation	7

Table 1: Time breakdown for the Single ARM Processor design.

1.5 Observations

- **CDC Performance**: Efficient gear hashing reduced chunk boundary determination latency. A pre-stored gear table optimized processing time further.
- **SHA Optimization**: NEON Intrinsics enables SIMD of SHA function, which significantly raised throughput from 90Mb/s to around 500Mb/s.
- **Deduplication**: In some applications with mostly unique chunks, Deduplication throughput can be extremely low since our calculation of throughput calculates the rate of passed chunk sizes accumulated.
- **LZW Bottleneck**: Despite optimization, LZW remained the slowest component in the pipeline.

2 Final Ultra96 Mapped Design

We primarily mapped LZW to Ultra96 FPGA to explore loop unrolling and pipelining. Since we also defined a global dictionary to store the chunk entries and LZW compressed codes, array partitioning can explore potential parallel computations to further reduce computation latencies. Detailed design achievements and topologies are discussed in this chapter.

2.1 Design Performance

2.1.1 Performance achieved

The following performance achieved by using default `-s` parameter at the value of 5 and and `-b` at the value of `BLOCKSIZE` = 8192.

We have tested with 3 files with different file sizes.

Test Case 1: b_data.dat (Size: 409,600B)

Compression achieved: 3.49 (from original size of 409,600B to compressed size of 117,259B).

Decomposed throughput:

Function	**Throughput (Mb/s)**	**Total Latency (ms)**
CDC	733.526	4.46719
SHA	236.69	13.8443
Dedup	388.211	8.44076
LZW	3.90608	0.7496
Total	119.149	27.5018

Table 2: Function Throughput and Total Latency

Terminal Output:

```
setting up sever...
server setup complete!
CDC function called: chunk_count = 29240
[ 4307.845339] [drm] bitstream 4d3ce153-a64b-4c4d-b461-25440533f50c locked, ref=1
[ 4312.958466] [drm] User buffer is not physical contiguous
All data written successfully.
Written file with 117259 bytes
Original Size: 409600 bytes
Compressed Size: 117259 bytes
Compression Ratio (original siz[ 4313.019553] [drm] bitstream 4d3ce153-a64b-4c4d-b461-25440533f50c unlocked, ref=0
e / compressed size): 3.49312

Total chunk size as dedup input[ 4313.023779] [drm] Pid 1541 closed device
: 409234 bytes in 29231 chunks.
Deduplication Contribution (dedup headers in Compressed size): 116924 bytes

Total chunk size as LZW input: 366 bytes in 9 chunks.
LZW Contribution (LZW header + LZ-compressed content in Compressed size): 335 bytes
INFO: Program completed successfully.
--------------- Key Throughputs ---------------
Input Throughput to Encoder: 329.599 Mb/s. (Latency: 0.0028461s).
-----------------------------------------------
Total Data Processed (input original size): 3276800 bits.
Total Time Taken of Encoder: 0.0275018 seconds or 27.5018ms.
Overall Throughput: 119.149 Mb/s.
-----------------------------------------------

CDC throughput 733.526 Mb/s.
SHA throughput 236.69 Mb/s.
Dedup throughput 388.211 Mb/s.
LZW throughput 3.90608 Mb/s.

Average time of CDC: 4.46719 (ms).
Average time of SHA: 0.00047347 (ms).
Average time of Dedup: 0.000288672 (ms).
Average time of LZW: 0.0832889 (ms).

Total time of CDC: 4.46719 (ms).
Total time of SHA: 13.8443 (ms).
Total time of Dedup: 8.44076 (ms).
Total time of LZW: 0.7496 (ms).
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./decoder out.bin out.dat
Decoding complete.
Total chunks in header: 29240
Successfully processed chunks: 29240
Failed chunks: 0
All chunks were successfully processed!
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# diff out.dat b_data.dat
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1#
```

Figure 3: Hardware Compression for b_data.dat

Test Case 2: LittlePrince.txt (Size: 14,248B)

Compression achieved: 2.56 (from original size of 14,248B to compressed size of 5,571B).

Decomposed throughput:

Function	Throughput (Mb/s)	Total Latency (ms)
CDC	677.146	0.16833
SHA	455.044	0.25049
Dedup	514.054	0.171513
LZW	10.7955	2.39063
Total	38.24	2.98098

Table 3: Function Throughput and Total Latency

Terminal Output:

```
setting up sever...
server setup complete!
CDC function called: chunk_count = 278
All data written successfully.
Written file with 5571 bytes
Original Size: 14248 bytes
Compressed Size: 5571 bytes
Compression Ratio (original size / c[13454.661191] [drm] bitstream 4d3ce153-a64b-4c4d-b461-25440533f50c locked, ref=1
ompressed size): 2.55753

Total chunk size as dedup input: 110[13471.462755] [drm] bitstream 4d3ce153-a64b-4c4d-b461-25440533f50c unlocked, ref=0
22 bytes in 218 chunks.
Deduplication Contribution (dedup heade[13471.479287] [drm] Pid 2014 closed device
rs in Compressed size): 872 bytes

Total chunk size as LZW input: 3226 bytes in 60 chunks.
LZW Contribution (LZW header + LZ-compressed content in Compressed size): 4699 bytes
INFO: Program completed successfully.
--------------- Key Throughputs ---------------
Input Throughput to Encoder: 1815.4 Mb/s. (Latency: 2.455e-05s).
-----------------------------------------------
Total Data Processed (input original size): 113984 bits.
Total Time Taken of Encoder: 0.00298098 seconds or 2.98098ms.
Overall Throughput: 38.2371 Mb/s.
-----------------------------------------------

CDC throughput 677.146 Mb/s.
SHA throughput 455.044 Mb/s.
Dedup throughput 514.054 Mb/s.
LZW throughput 10.7955 Mb/s.

Average time of CDC: 0.16833 (ms).
Average time of SHA: 0.000901043 (ms).
Average time of Dedup: 0.000617016 (ms).
Average time of LZW: 0.0398438 (ms).

Total time of CDC: 0.16833 (ms).
Total time of SHA: 0.25049 (ms).
Total time of Dedup: 0.171531 (ms).
Total time of LZW: 2.39063 (ms).
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./decoder out.bin out.txt
Decoding complete.
Total chunks in header: 278
Successfully processed chunks: 278
Failed chunks: 0
All chunks were successfully processed!
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# diff out.txt littleprince.txt
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1#
```

Figure 4: Hardware Compression for LittlePrince.txt

Test Case 3: Franklin.txt (Size: 399,406B)

Compression achieved: 0.71 (from original size of 563,009B to compressed size of 563,009B). This text file mostly includes unique chunks, that is why the compressor can not perform data compression as expected but add overheads due to header size.

Decomposed throughput:

Function	**Throughput (Mb/s)**	**Total Latency (ms)**
CDC	939.936	3.39943
SHA	415.335	7.69311
Dedup	1.47806	7.50666
LZW	14.74	216.016
Total	13.62	234.675

Table 4: Function Throughput and Total Latency - Franklin Test Case

Terminal Output:

```
setting up sever...
server setup complete!
CDC function called: chunk_count = 6453
[13565.993593] [drm] bitstream 4d3ce153-a64b-4c4d-b461-25440533f50c locked, ref=1
[13574.504403] [drm] User buffer is not physical contiguous
All data written successfully.
Written file with 563009 bytes
Original Size: 399406 bytes
Compressed Size: 563009 bytes
Compression Ratio (original size / compressed size): 0.709413

Total chunk size as dedup input: 1[13576.756426] [drm] bitstream 4d3ce153-a64b-4c4d-b461-25440533f50c unlocked, ref=0
398 bytes in 538 chunks.
Deduplication Contribution (dedup head[13576.761059] [drm] Pid 2027 closed device
ers in Compressed size): 2152 bytes

Total chunk size as LZW input: 398008 bytes in 5915 chunks.
LZW Contribution (LZW header + LZ-compressed content in Compressed size): 560857 bytes
INFO: Program completed successfully.
--------------- Key Throughputs ---------------
Input Throughput to Encoder: 1536.38 Mb/s. (Latency: 0.00293162s).
-----------------------------------------------
Total Data Processed (input original size): 3195248 bits.
Total Time Taken of Encoder: 0.234675 seconds or 234.675ms.
Overall Throughput: 13.6156 Mb/s.
-----------------------------------------------

CDC throughput 939.936 Mb/s.
SHA throughput 415.335 Mb/s.
Dedup throughput 1.47806 Mb/s.
LZW throughput 14.74 Mb/s.

Average time of CDC: 3.39943 (ms).
Average time of SHA: 0.00119219 (ms).
Average time of Dedup: 0.00117258 (ms).
Average time of LZW: 0.03652 (ms).

Total time of CDC: 3.39943 (ms).
Total time of SHA: 7.69317 (ms).
Total time of Dedup: 7.56666 (ms).
Total time of LZW: 216.016 (ms).
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./decoder out.bin out.txt
Decoding complete.
Total chunks in header: 6453
Successfully processed chunks: 6453
Failed chunks: 0
All chunks were successfully processed!
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# diff out.txt franklin.txt
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1#
```

Figure 5: Hardware Compression for Franklin.txt

2.1.2 Vivado Analysis

The power consumption and resource utilization are analyzed using Vivado.

Power Consumption: The estimated total on-chip power is **2.389W**.

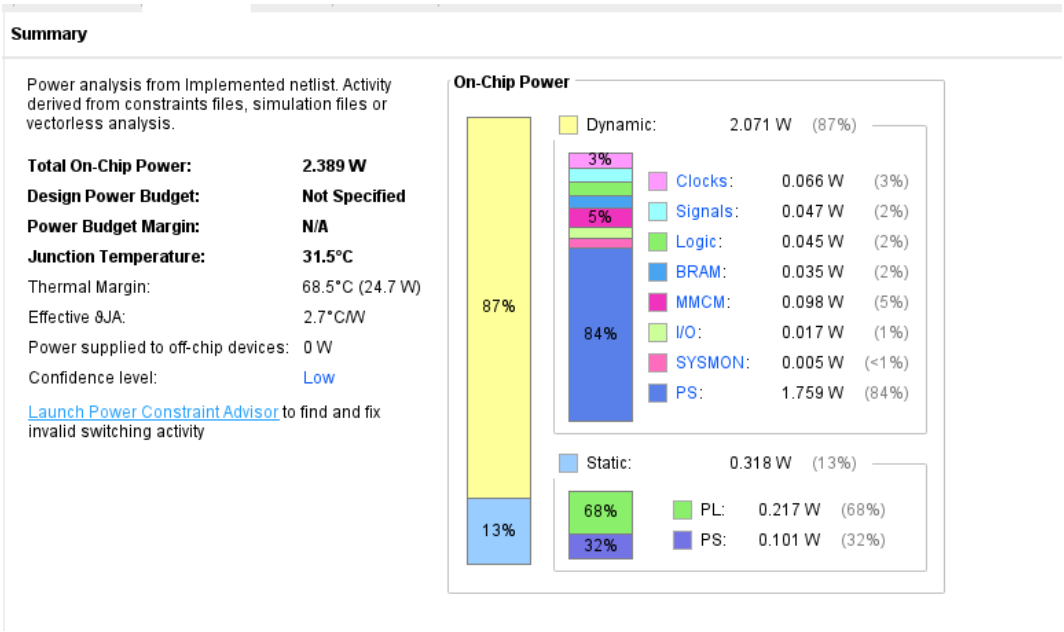

Figure 6: Power Consumption

Resource Utilization:

Resource	Utilization	Available	Utilization %
LUT	19351	70560	27.42
LUTRAM	1938	28800	6.73
FF	23459	141120	16.62
BRAM	24.50	216	11.34
IO	45	82	54.88
BUFG	4	196	2.04
MMCM	1	3	33.33

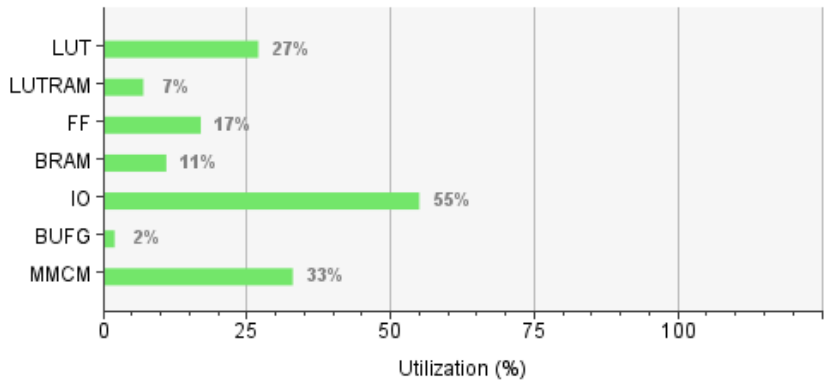

Figure 7: Resource Utilization

Hardware mapping in control interface of the accelerator: From the Address Mapper, we can discover that `lzw_encode_hw/s_axi_control` is mapped to addresses 0x00_B000_0000 to 0x00_B000_FFFF.

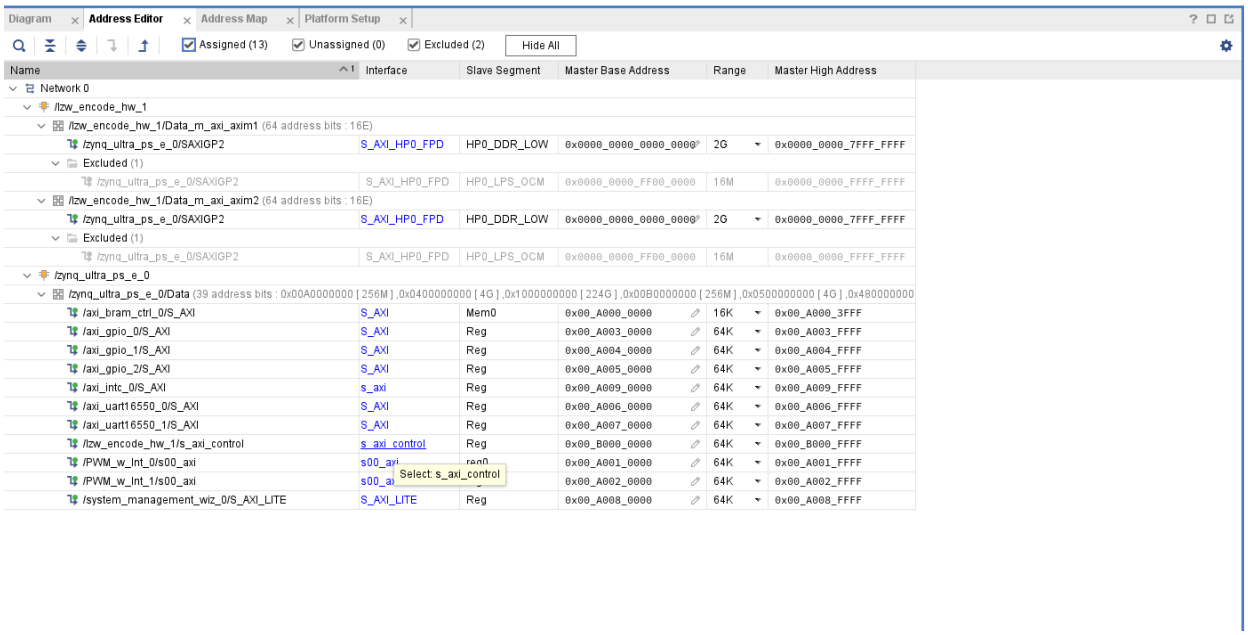

Figure 8: Address Editor Output

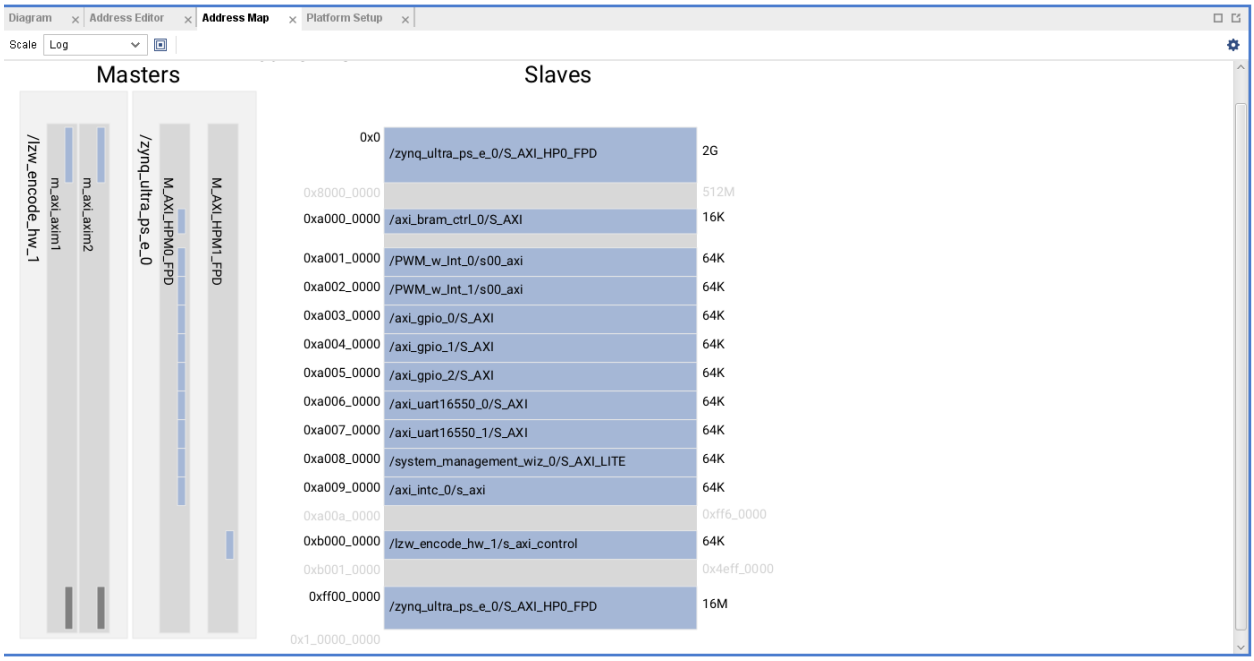

Figure 9: Address Map

2.2 Key Design Aspects

2.2.1 Design Flow

The final design efficiently distributes tasks across the ARM processor, NEON vector instructions, and FPGA logic.

The ARM processor is responsible for data reception, buffer management, and content-defined chunking (CDC). Data packets are received via Ethernet, decoded, and stored in a large file buffer.

CDC function running on ARM: The ARM processor then performs CDC using a rolling hash function to identify chunk boundaries, ensuring that data is divided dynamically based on its content.

SHA256 and Deduplication function: SHA-256 hash computation for each chunk is accelerated using NEON vector instructions on the ARM processor. NEON's parallelism enables efficient hashing, which is critical for deduplication. The deduplication logic itself is also executed on the ARM processor. It employs a hash table to identify duplicate chunks by comparing their SHA-256 fingerprints. Duplicate chunks are referenced by their index in the hash table, while new chunks are indexed and sent for further processing.

LZW function running on FPGA: Non-duplicate chunks are compressed using the FPGA, which accelerates the LZW compression process with dedicated hardware kernels. The ARM processor facilitates this by managing data transfers between the host and the FPGA using OpenCL buffers. This offloading of compression to the FPGA significantly reduces the computational burden on the ARM processor.

The kernel diagram is shown here:

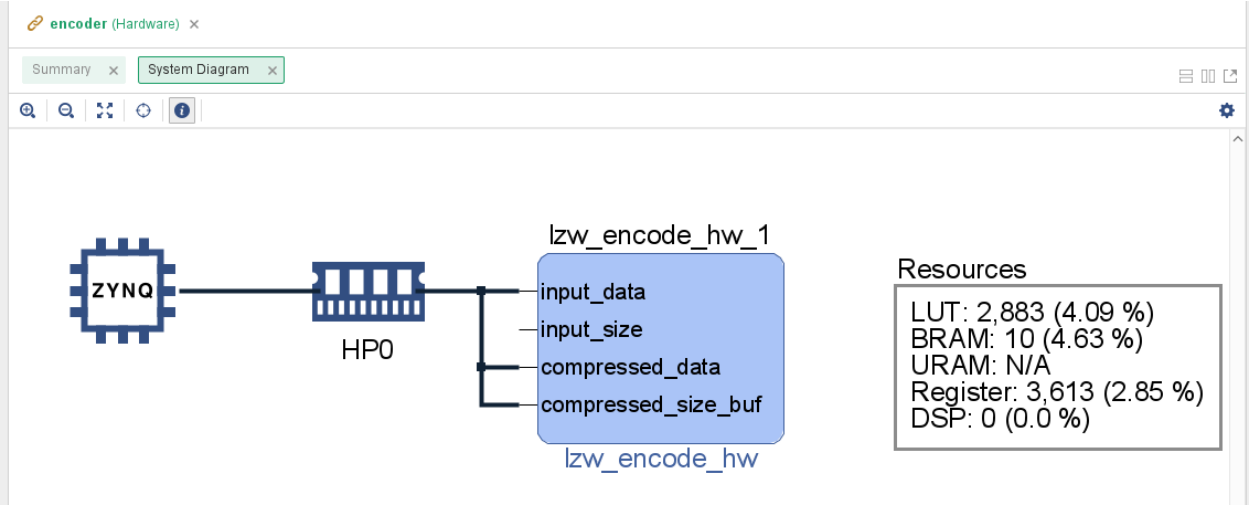

Figure 10: Kernel Design

Finally, the ARM processor packages the deduplication and compression results, adding appropriate headers to each chunk, and writes the final compressed output to a file.

2.2.2 ZYNQ resource mapping

The following diagrams show the block diagram of hardware and how our LZW kernels are mapped on FPGA.

System Block Diagram

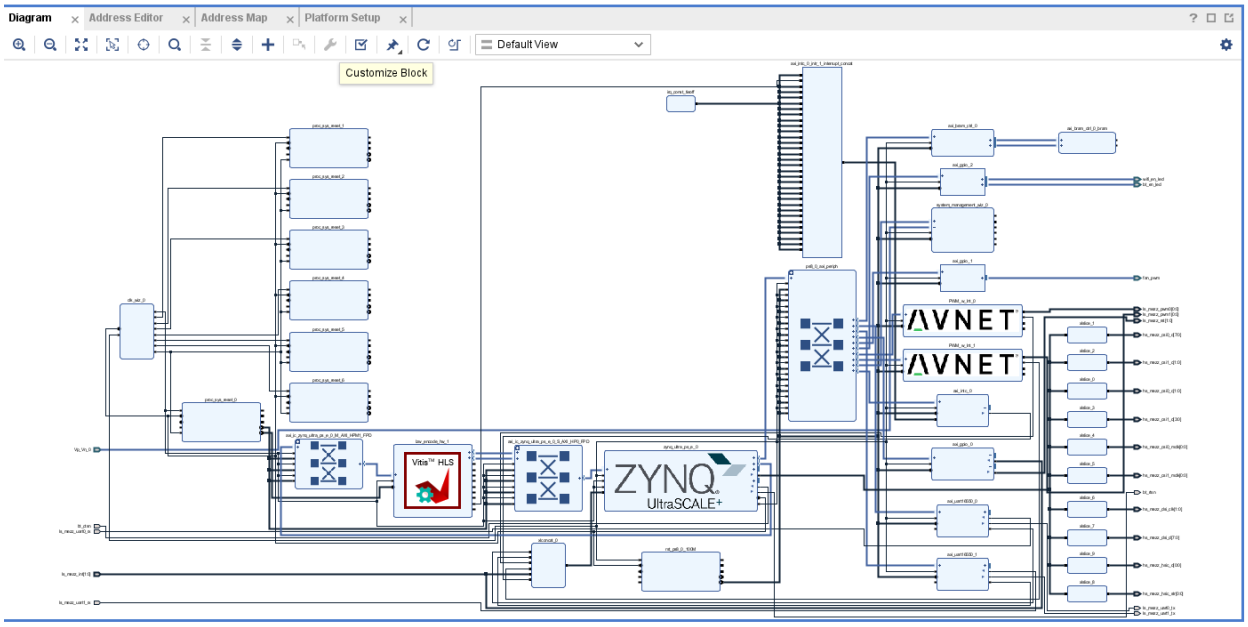

Figure 11: System Block Diagram

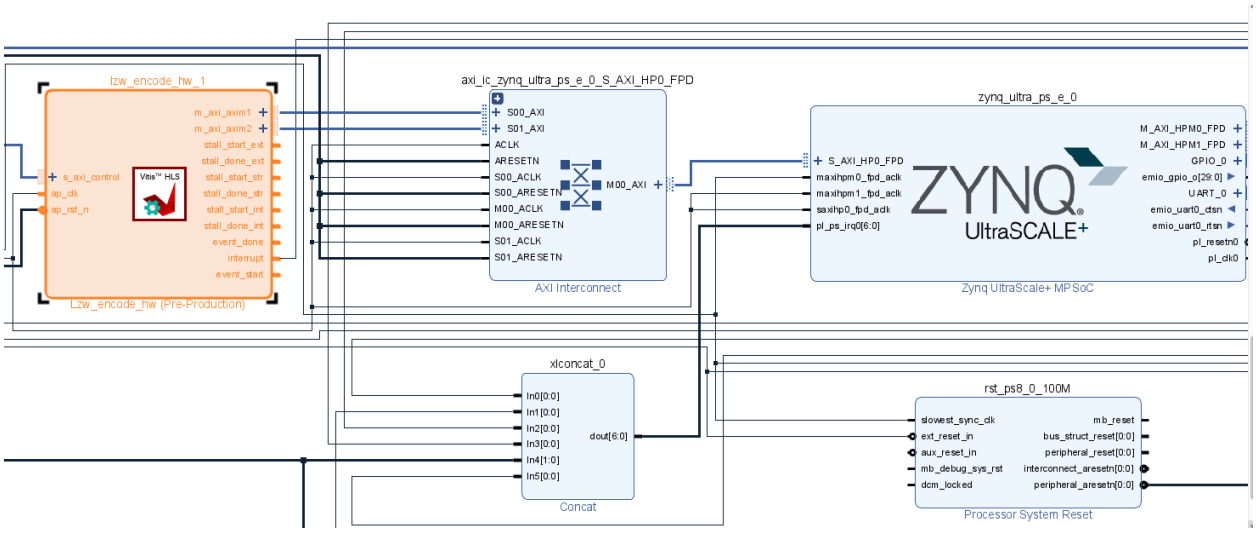

Figure 12: A Close Look at the System Block Diagram

On-Device Kernel Mapping

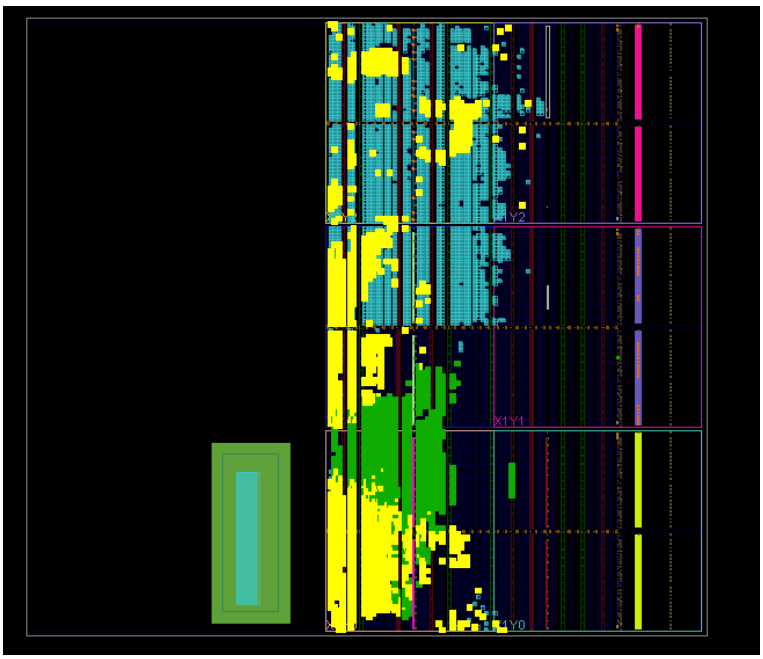

Figure 13: Kernel Mapping

From Figure 14, the worst case delay is 1.523 ns and can be discovered and is highlighted in Figure 15.

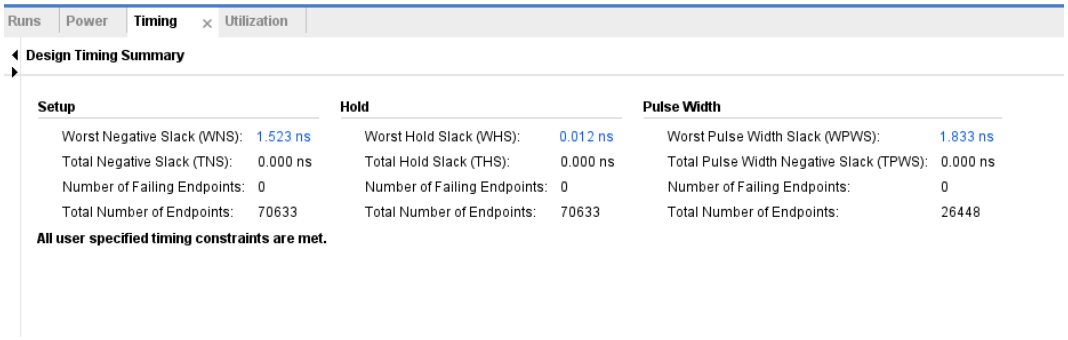

Figure 14: Timing Summary

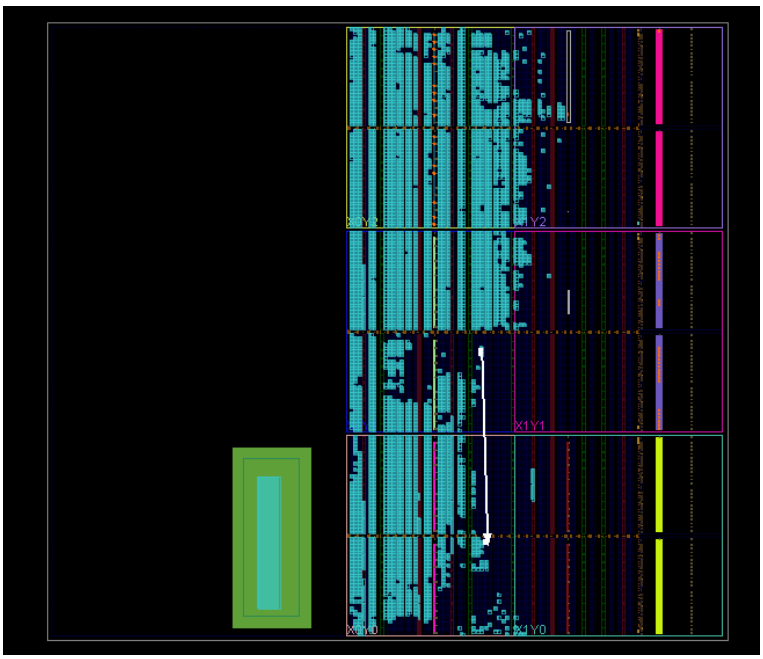

Figure 15: Worst Case Negative Slack

2.3 Current Bottleneck

Although we have accelerated CDC and SHA functions and a great increase of throughput is achieved, the predominent bottleneck lies on the LZW function. Depending on the number of duplicates found, LZW throughput may severly impact the overall throughput. In our current LZW algorithm design, loops like shown in Code 1 does not satisfy the prerequisite of using `#pragma HLS UNROLL` since the compiler is unable to identify the exact number of iterations.

```
for (uint16_t j = 256; j < dict_size; ++j) {
    if (hw_dictionary[j][0] == current_code && hw_dictionary[j][1] ==
        next_char) {
    current_code = j;
    found = true;
    break;
    }
}
```

Code 1: Relevant loop in LZW Hardware Implementation

Another significant bottleneck is the data transmission and reception from FPGA mapped memory and Host. Although we have tried implementing a hardware code to store unique chunk index and buffer contents in a large buffer and transmit data in bulks, we have failed due to incorrect array partitioning. Combined all the reasons stated above, the FPGA

mapped design achieved a throughput which is significantly slower than ARM processor mapped design.

3 Design Validation and Real-time Input

3.1 Function Description and Validation

We discovered many optimizations throughput the project, latest updated designs are discussed in this section.

- **CDC:**

  As shown in Code 3, the optimized Content-Defined Chunking (CDC) process focuses on pre-loading the gear table to reduce computational overhead and efficiently determine chunk boundaries using a rolling gear hash. The computation is also significantly simpler

  Initially, while testing full functionality of the CDC function, we created a main function to chunk the provided **LittlePrince.txt** and binary file was generated with gcc compiler. Chunks are created and multiple duplicates are discovered in `gdb`, proving that the CDC is chunking the input data based on the contents, are identical chunks are given same gear hashing value so that they are chunked to create more duplicates.

- **SHA256:**

  We explored SIMD acceleration for SHA256 function and gained a significant higher throughput. By leveraging ARMv8.2-A architecture-specific cryptographic extensions, such as the vsha256hq_u32 and vsha256h2q_u32 NEON intrinsics, which perform multiple rounds of SHA-256 compression in hardware, significantly reducing computation time compared to software-based implementations.

  We compiled the software version of `encoder.cpp` using the standard compilation process but included the `-march=armv8.2-a+crypto` flag in the host compiler options. This flag enables the use of ARMv8.2-A cryptographic extensions required for processing SHA256 using SIMD instructions. During the initial testing phase, the SHA function was validated using sample text inputs generated from online resources. Identical chunks produced consistent SHA hashing results, confirming the function's correctness.

- **Deduplication:**

  Our deduplication design is a simple search in the `unordered_map` with `sha_fingerprint`. It tried to insert the input `sha_fingerprint` to the existing hash dictionary and if failed (SHA fingerprint already exist), a duplicate chunk is found and value of -1 is returned. There was no specific validation alone for this function but we debugged regularly while overall functionality in final design.

- **LZW:** LZW was mapped on FPGA kernels. While exploring acceleration possible for hardware mapped LZW, we analyzed our loops and dictionary. By partitioning the dictionary array and binding it to BRAMs using `#pragma HLS BIND_STORAGE` and `#pragma HLS ARRAY_PARTITION`, the dictionary lookups can be efficiently parallelized. Additionally, our design leverages pipelining and unrolling, as indicated by `#pragma HLS PIPELINE` and `#pragma HLS UNROLL`, to exploit the inherent parallelism of hardware and reduce latency.

  After implementing our LZW design on the FPGA, we conducted a C/RTL Co-simulation to validate that the synthesized design's functionality aligns with the behavior observed in the ARM-based implementation. Figure 16 provides co-simulation verification result which indicates that the hardware version functions the same as software version.

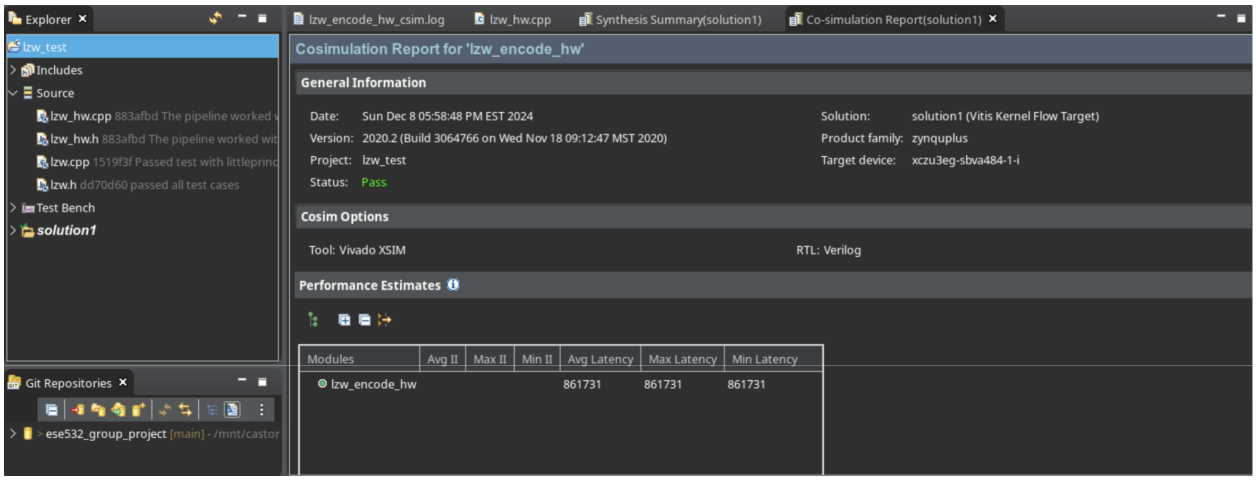

Figure 16: Co-simulation Result

3.2 FPGA Mapped Design Validation

When testing our design on FPGA, we used the following commands.

1. **On Ultra96**:

   $ ./encoder [compressedFilename]

2. **On Local Machine**:

   $ ./client -i 10.10.7.1 -f [TestFilename]

3. **On Ultra96**:

   $ ./decoder [compressedFilename] [decodedFilename]

$ diff [decodedFilename] [TestFilename]

In the final demo, we successfully compressed the provided test case **b_data.dat**, the overall throughput shown in Figure 17 is **119.15Mb/s**.

```
setting up sever...
server setup complete!
CDC function called: chunk_count = 29240
[ 4307.845339] [drm] bitstream 4d3ce153-a64b-4c4d-b461-25440533f50c locked, ref=1
[ 4312.958466] [drm] User buffer is not physical contiguous
All data written successfully.
Written file with 117259 bytes
Original Size: 409600 bytes
Compressed Size: 117259 bytes
Compression Ratio (original siz[ 4313.019553] [drm] bitstream 4d3ce153-a64b-4c4d-b461-25440533f50c unlocked, ref=0
e / compressed size): 3.49312

Total chunk size as dedup input[ 4313.023779] [drm] Pid 1541 closed device
: 409234 bytes in 29231 chunks.
Deduplication Contribution (dedup headers in Compressed size): 116924 bytes

Total chunk size as LZW input: 366 bytes in 9 chunks.
LZW Contribution (LZW header + LZ-compressed content in Compressed size): 335 bytes
INFO: Program completed successfully.
--------------- Key Throughputs ---------------
Input Throughput to Encoder: 329.599 Mb/s. (Latency: 0.0028461s).
-----------------------------------------------
Total Data Processed (input original size): 3276800 bits.
Total Time Taken of Encoder: 0.0275018 seconds or 27.5018ms.
Overall Throughput: 119.149 Mb/s.
-----------------------------------------------

CDC throughput 733.526 Mb/s.
SHA throughput 236.69 Mb/s.
Dedup throughput 388.211 Mb/s.
LZW throughput 3.90608 Mb/s.

Average time of CDC: 4.46719 (ms).
Average time of SHA: 0.00047347 (ms).
Average time of Dedup: 0.000288672 (ms).
Average time of LZW: 0.0832889 (ms).

Total time of CDC: 4.46719 (ms).
Total time of SHA: 13.8443 (ms).
Total time of Dedup: 8.44076 (ms).
Total time of LZW: 0.7496 (ms).
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./decoder out.bin out.dat
Decoding complete.
Total chunks in header: 29240
Successfully processed chunks: 29240
Failed chunks: 0
All chunks were successfully processed!
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# diff out.dat b_data.dat
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1#
```

Figure 17: Hardware Throughput for b_data.dat

3.3 Real-time Input and Processing

We guaranteed real-time input by providing the input data over Ethernet from local machines, we validated the ethernet functionality using `iperf`.

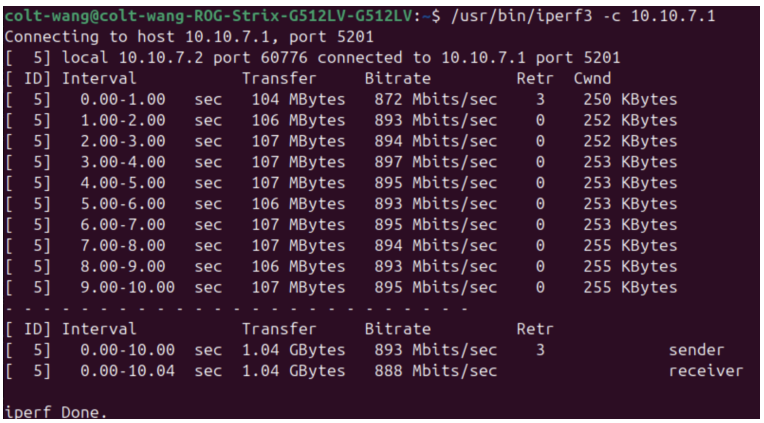

Figure 18: Ethernet Rate

All the terminal output screenshots shown above (such as Fig.2) indicate that we have successfully verify functionality of the entire encoder by comparing the decoded file with the original file. There are two crucial steps we adopted to ensure real-time data input and processing.

1. A potential area for parallelism lies in enabling real-time packet processing. For large files, our design transmits the file in multiple packets. Upon receiving the first packet, the algorithm initiates CDC on the existing file buffer while concurrently populating the buffer with subsequent packets. This approach effectively reduces latency during data reception, optimizing the overall processing efficiency. We used a while loop `while(!done)` to process received packets using `process_packet` function (Code 9). We initially added a 70Mb limit to the `file_buffer` to avoid buffer overflow in the program. We dynamically frees the `file_buffer` after processing each packet.

2. In our design, the functions are executed in a streamlined manner as defined in the `process_packet` function (Code 9). To ensure that data or chunks can be processed and propagated through the compression application in real time, we incorporated measures in the functional design before integrating the components. Given that the team's responsibilities were distributed among multiple members, maintaining consistent entry formats was critical to achieving uniform integration and efficient verification. To support seamless collaboration, we developed a program flowchart (Figure 19) that provides a clear and standardized framework for team coordination.

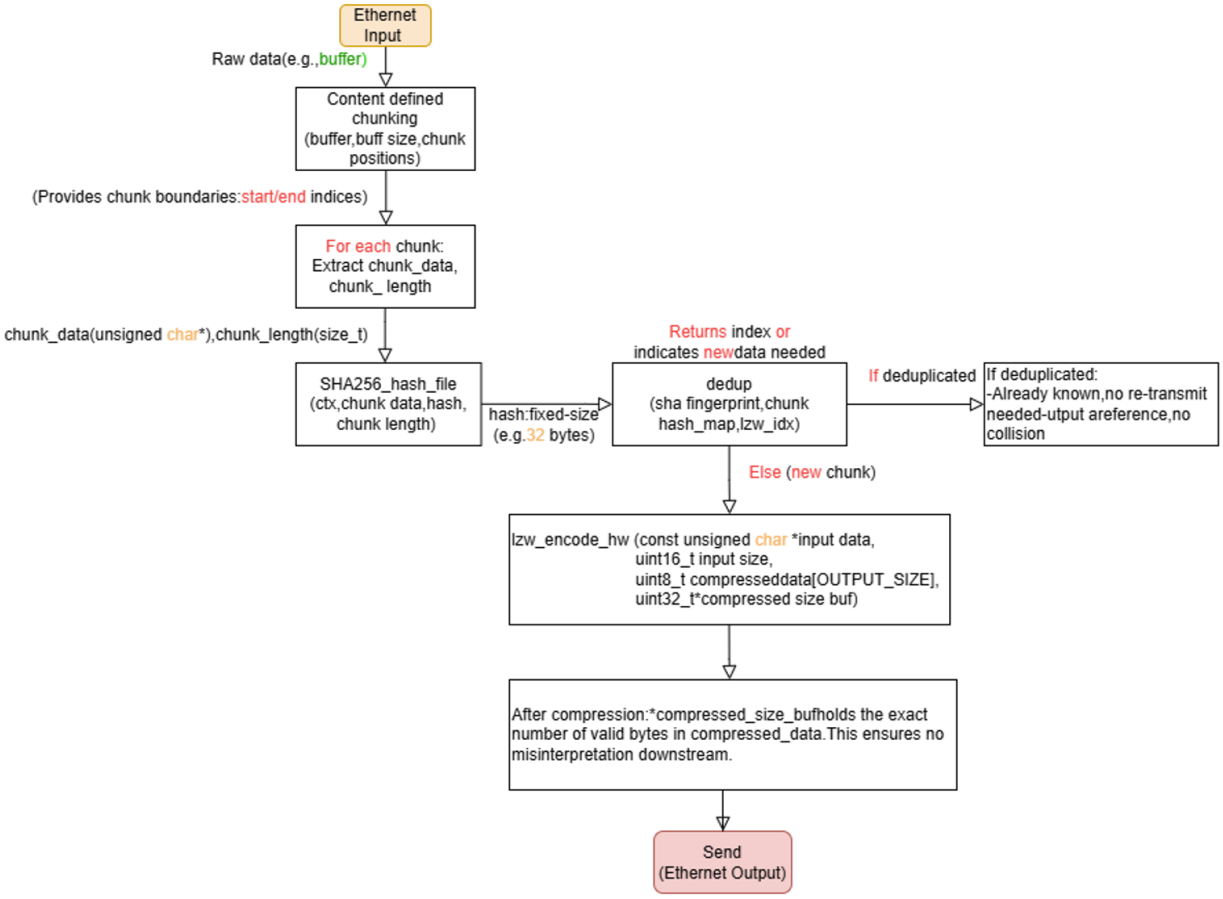

Figure 19: Design Consistency

4 Key Lessons

1. **CDC Implementation:**
    - Implemented an efficient rolling hash to enable rapid boundary detection while processing large datasets, reducing computational overhead compared to traditional fixed-block approaches.
    - Learned how to optimize the CDC function by carefully selecting parameters like the average chunk size and boundary markers to achieve a balance between deduplication efficiency and computational cost.
    - Addressed memory management issues, ensuring efficient handling of large buffers and avoiding bottlenecks caused by excessive memory allocation or data copying.
2. **SHA256 Implementation:**
    - Gained a deeper understanding of cryptographic hashing algorithms, particularly the mechanics of SHA-256, including message padding, word expansion, and the SIMD potential.
    - Debugging challenges, such as validating intermediate hash values against known test vectors, underscored the importance of incremental testing during implementation.
3. **LZW Compression:**
    - Gained an understanding of dictionary-based compression techniques, particularly how LZW dynamically builds a dictionary of patterns during compression processes.
    - Gained knowledge of using pragmas such as `#pragma HLS array partition` to accelerate the program by mapping and partitioning memories on FPGA.
    - Debugged edge cases, such as handling input data or chunk size data type that results in dictionary overflow or rare patterns that delay dictionary utilization, by implementing comprehensive test cases.

5 Design Space Exploration and Graphs

5.1 CDC

5.1.1 Baseline CDC

As shown in Code 2, our baseline CDC algorithm uses similar methodology as in previous homework submission. The baseline `content_defined_chunking` function implements a sliding-window-based Content-Defined Chunking (CDC) algorithm. The algorithm uses a *Rabin Fingerprint* hash to calculate a rolling signature (`sig`) over a sliding window of size `CHUNK_WIN_SIZE`. The initial `sig` is computed by iterating through the first window. For subsequent positions, `sig` is updated incrementally using modular arithmetic to subtract the contribution of the outgoing byte and add the incoming byte efficiently.

A chunk boundary is identified when the bitwise AND operation between `sig` and `CHUNK_MASK` equals zero, ensuring that boundaries are content-sensitive. Additionally, chunks are constrained by a minimum size (`CHUNK_MIN_SIZE`) and a maximum size (`CHUNK_MAX_SIZE`). If the window exceeds the maximum size without finding a boundary, a boundary is forced.

5.1.2 Optimization Description of CDC

Note that the total count of CDC chunks can be manipulated by changing the value of `MODULUS`. Therefore, we can determine a specific `MODULUS` value so that overall software throughput is optimized.

1. **Initialization:**
   - Initialize `hash` to 0 and set `chunk_start` and `i` to 0.
   - Append the starting position (0) to `chunk_positions`.
2. **Rolling Hash Computation:**
   - Iterate through the buffer (`i < buff_size`).
   - Update `hash` using:

$$\texttt{hash} = (\texttt{hash} \gg 1) + \texttt{gearTable[buffer[i]]}.$$

3. **Chunk Boundary Detection:**
   - Split chunks if:
     (a) `hash` matches the target value (`hash & (MODULUS - 1) == TARGET`),

(b) The chunk size exceeds `MIN_CHUNK_SIZE` or reaches `MAX_CHUNK_SIZE`, or

(c) End of the buffer is reached.

- Append the boundary to `chunk_positions` and reset `hash`.

Test Case - LittlePrince Text File (File Size: 14248B)

Modulus	Chunk Count	Total Throughput (Mb/s)	CDC Throughput (Mb/s)	SHA Throughput (Mb/s)	DEDUP Throughput (Mb/s)	LZW Throughput (Mb/s)
64	278	96.58	669.51	535.39	738.43	40.16
256	42	40.29	693.38	772.72	2545.99	11.36
512	17	27.33	813.24	826.87	3608.23	8.55

Table 5: Chunk Size Versus Throughput

Another reason of choosing a smaller modulus value is related to the functionality of the subsequent functions. Due to a greater `MODULUS` value, the average chunk size will be larger, which means the number of chunks will decrease. As the chunk size increases, there will be less chance for the Deduplication algorithm to find duplicate chunks and potentially cause incorrect output. As shown in Table 6, a `MODULUS` greater than 64 will result in nothing duplicated found in the deduplication function.

Test Case - LittlePrince Text File (File Size: 14248B)

Modulus	Contributed DEDUP (B)	Contributed LZW (B)	Compression Ratio
64	11022	3226	2.56
256	10739	3509	3.75
512	10120	4128	3.60

Table 6: Chunk Size Versus Compression Ratio - LittlePrince

The chart below shows the relation between chunk size and throughput/compression ratio. As the modulus increases, the total throughput decreases significantly, indicating that larger modulus values require more computational effort, due to increased chunk sizes and a reduced number of chunks. However, the compression ratio improves with higher modulus values up to a certain point, peaking at a modulus of 256. This improvement suggests that larger chunks allow better compression efficiency by capturing more repetitive patterns. Beyond

this point, the compression ratio slightly declines at a modulus of 512, indicating diminishing returns in compression efficiency for overly large chunks. Thus, the choice of modulus involves a trade-off between throughput and compression efficiency.

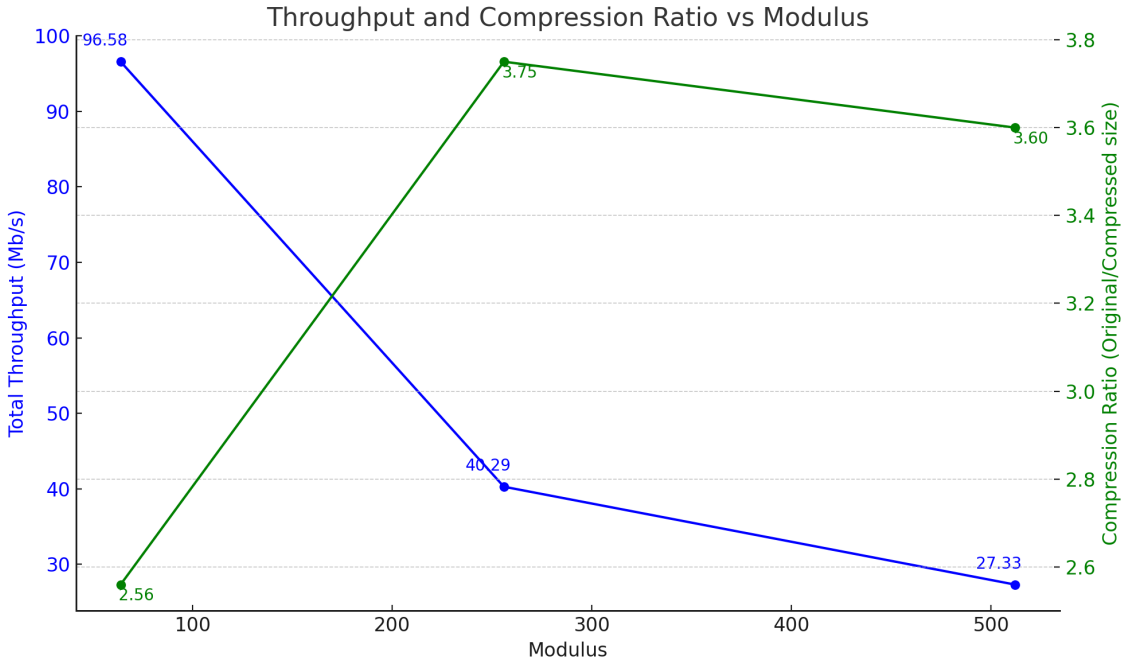

Figure 20: Design space axis: Modulus in CDC

5.2 SHA256

Initial State (Baseline): We began with a software implementation of SHA256 from the `mpsoc-crypto` library. While correct and portable, the scalar CPU implementation lacked the throughput required for real-time data processing.

Design Space Explored:

- *Scalar vs. NEON-Accelerated CPU:* The scalar approach was simpler but slower, while using ARM NEON intrinsics promised a significant speedup by exploiting SIMD parallelism.
- *Hardware Acceleration Consideration:* Although full hardware acceleration (e.g., FPGA) was considered, the complexity and overhead did not justify the gain compared to NEON optimization.

Chosen Approach: We adopted the NEON-accelerated SHA256 implementation from the `mpsoc-crypto` library. This yielded substantial performance gains with moderate complexity, leveraging the processor's SIMD units effectively. The following design flow shows the

streamline of the latest hashing algorithm implemented.

1. **Initialization:**
   - Set the initial hash state constants as defined by the SHA-256 standard.
   - Initialize internal counters (`bitlen`, `datalen`) to zero.
2. **Data Processing:**
   - For each input byte, add it to the internal data buffer.
   - Once the buffer reaches 64 bytes, call `sha256_transform`:

     ```
     transform(state, buffer)
     ```

     which uses NEON intrinsics (`vsha256hq_u32`, etc.) to process one 512-bit block.
   - Update the bit length count after processing each full block.
3. **Finalization:**
   - Pad the remaining data to a full 64-byte block, appending the bit length as a 64-bit value.
   - Call `sha256_transform` one last time to incorporate the final padded block.
   - Extract the resulting hash values from the internal state, reversing the byte order since SHA-256 is big-endian.

5.3 LZW Kernel

Initial State (Baseline): The initial LZW implementation suffered from long loops, associative memory lookups, and high overhead due to frequent kernel calls and hashing collisions.

Design Space Explored:

- *Loop Transformations:* Replaced `while` loops with `for` loops for known iteration counts, aiding Vitis HLS optimizations.
- *Associative Memory and Hashing:* Introduced the Murmur hash to reduce collisions and adopted a 2-bucket scheme per index, lowering the chance of multiple collisions and reducing associative memory usage.
- *Batch Processing and Reduced Arguments:* Batched multiple chunks (16KB) before sending to the kernel, reducing call overhead. Encoded parameters within arrays to

reduce the number of kernel arguments.

- *Bit-Packing Decisions:* Attempted to move bit-packing inside the kernel but found it detrimental to performance. Thus, reverted to host-side bit-packing.
- *Code Length Adjustments:* Moved to 13-bit packing for stability under larger code values.

Chosen Approach: A set of algorithmic and structural optimizations (loop transformations, improved hashing, batching) and careful parameter passing significantly improved LZW throughput. Ultimately, bit-packing remained on the host, and parameter complexity was minimized.

6 Team Contribution

We have thoroughly discussed team contributions prior to the project and adaptively adjusted roles and responsibilities throughout the process to ensure effective collaboration and successful completion of our goals.

Team Member	**Responsible**	**Contributions**
Truong (Winston) Nguyen	LZW & FPGA Mapping	Coordinated team meetings and lead the progress for FPGA design, collaborated on encoder.cpp, composed host.cpp, optimized LZW on FPGA, write-up and verified whole function implementation.
Ke Liu	Report Writing	Conducted background research and provided data analysis for the project report checking and write-up.
Willie Gu	CDC & Dedup	Designed and implemented core algorithms, optimized SHA256 NEON and write up ensured code quality, worked on the CDC and LZW portion of the project, collaborated on doing encoder.cpp and performed debugging.
Kun Wang	CDC & SHA256	Took the lead on the encoder.cpp section, collaborated on testing encoder.cpp, optimized CDC and SHA256, prepared technical documentation, created presentation slides, and maintained version control, report checking and write-up.

Table 7: Team Contributions Table

We, *Truong (Winston) Nguyen, Ke Liu, Willie Gu, and Kun Wang*, certify that we have complied with the University of Pennsylvania's Code of Academic Integrity in completing this final exercise.

7 Appendix

7.1 ARM Processor Mapped Design Validation

7.1.1 Test Case: Franklin.txt (Size: 399406B)

```
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./encoder_sw out.bin
Using out.bin as output file name.
setting up sever...
server setup complete!
Original Size: 399406 bytes
CDC function called: chunk_count = 6453
Compressed Size: 563009 bytes
Compression Ratio (original size/ compressed size): 0.709413

Total chunk size as dedup input: 1398 bytes in 538 chunks.
Deduplication Contribution (dedup headers in Compressed size): 2152 bytes

Total chunk size as LZW input: 398008 bytes in 5915 chunks.
LZW Contribution (LZW header + LZ-compressed content in Compressed size): 560857 bytes
Compressed data size: 563009 bytes.

All data written successfully.
Written file with 563009 bytes
--------------- Key Throughputs ---------------
Input Throughput to Encoder: 1591.99 Mb/s. (Latency: 0.00282921s).
-----------------------------------------------
Total Data Processed (input original size): 3195248 bits.
Total Time Taken of Encoder: 0.112176 seconds or 112.176ms.
Overall Throughput: 28.4842 Mb/s.
-----------------------------------------------

CDC throughput 944.512 Mb/s.
SHA throughput 479.506 Mb/s.
Dedup throughput 501.409 Mb/s.
LZW throughput 33.2515 Mb/s.

Average time of CDC: 3.38296 (ms).
Average time of SHA: 0.00103264 (ms).
Average time of Dedup: 0.000987531 (ms).
Average time of LZW: 0.0161888 (ms).

Total time of CDC: 3.38296 (ms).
Total time of SHA: 6.66362 (ms).
Total time of Dedup: 6.37254 (ms).
Total time of LZW: 95.757 (ms).
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./decoder out.bin out.txt
Decoding complete.
Total chunks in header: 6453
Successfully processed chunks: 6453
Failed chunks: 0
All chunks were successfully processed!
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# diff out.txt franklin.txt
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1#
```

Figure 21: Software throughput with Franklin

7.1.2 Test Case: b_data.dat (Size: 409600B)

```
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./encoder_sw out.bin
Using out.bin as output file name.
setting up sever...
server setup complete!
Original Size: 409600 bytes
CDC function called: chunk_count = 29240
Compressed Size: 117259 bytes
Compression Ratio (original size/ compressed size): 3.49312

Total chunk size as dedup input: 409234 bytes in 29231 chunks.
Deduplication Contribution (dedup headers in Compressed size): 116924 bytes

Total chunk size as LZW input: 366 bytes in 9 chunks.
LZW Contribution (LZW header + LZ-compressed content in Compressed size): 335 bytes
Compressed data size: 117259 bytes.

All data written successfully.
Written file with 117259 bytes
--------------- Key Throughputs ---------------
Input Throughput to Encoder: 335.196 Mb/s. (Latency: 0.00279858s).
-------------------------------------------------
Total Data Processed (input original size): 3276800 bits.
Total Time Taken of Encoder: 0.0261977 seconds or 26.1977ms.
Overall Throughput: 125.08 Mb/s.
-------------------------------------------------

CDC throughput 742.804 Mb/s.
SHA throughput 246.913 Mb/s.
Dedup throughput 386.963 Mb/s.
LZW throughput 62.0208 Mb/s.

Average time of CDC: 4.41139 (ms).
Average time of SHA: 0.000453868 (ms).
Average time of Dedup: 0.000289603 (ms).
Average time of LZW: 0.00524556 (ms).

Total time of CDC: 4.41139 (ms).
Total time of SHA: 13.2711 (ms).
Total time of Dedup: 8.468 (ms).
Total time of LZW: 0.04721 (ms).
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# ./decoder out.bin out.dat
Decoding complete.
Total chunks in header: 29240
Successfully processed chunks: 29240
Failed chunks: 0
All chunks were successfully processed!
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1# diff out.dat b_data.dat
root@u96v2-sbc-base-2020-2:/media/sd-mmcblk0p1#
```

Figure 22: Software throughput with b_data.dat

7.2 Content Defined Chunking

7.2.1 Baseline CDC

```
1 void content_defined_chunking(unsigned char* buffer, size_t buffer_size,
     std::vector<size_t>& chunk_positions) {
2     uint64_t power = 1;
3     for (int i = 1; i < CHUNK_WIN_SIZE; i++)
4         power = (power * 256) % CHUNK_PRIME;
5
6     uint64_t sig = 0;
7     for (int i = 0; i < CHUNK_WIN_SIZE; i++)
```

```
        sig = (sig * 256 + (unsigned char)buffer[i]) % CHUNK_PRIME;

    size_t last_idx = 0;
    for (size_t i = 1; i <= buffer_size - CHUNK_WIN_SIZE; i++) {
        sig = (sig + CHUNK_PRIME - power * (unsigned char)buffer[i - 1] %
           CHUNK_PRIME) % CHUNK_PRIME;
        sig = (sig * 256 + (unsigned char)buffer[i + CHUNK_WIN_SIZE - 1])
           % CHUNK_PRIME;

        if ((sig & CHUNK_MASK) == 0) {
            if (i + 1 - last_idx >= CHUNK_MIN_SIZE) {
                chunk_positions.push_back(i + 1);
                last_idx = i + 1;

            }
        } else if (i + 1 - last_idx >= CHUNK_MAX_SIZE) {
            chunk_positions.push_back(i + 1);
            last_idx = i + 1;
        }
    }

    if (last_idx < buffer_size - 1)
        chunk_positions.push_back(buffer_size);
}
```

Code 2: Baseline CDC Implementation

7.2.2 Optimized CDC

```
void content_defined_chunking(unsigned char* buffer, unsigned int
   buff_size, std::vector<size_t>& chunk_positions) {
    uint64_t hash = 0;
    size_t chunk_start = 0;
    size_t i = 0;

    chunk_positions.push_back(0);

    while (i < buff_size) {
        hash = (hash >> 1) + gearTable[buffer[i]];
        i++;
        size_t chunk_size = i - chunk_start;
        if (((hash & (MODULUS - 1)) == TARGET && chunk_size >=
           MIN_CHUNK_SIZE) || chunk_size >= MAX_CHUNK_SIZE || i ==
           buff_size) {
            chunk_positions.push_back(i);
            hash = 0;
```

```
            chunk_start = i;
        }
    }
}
```

Code 3: Optimized CDC Implementation

7.3 SHA256

7.3.1 Baseline SHA256

```
static const WORD k[64] = {
    // First 32 bits of the fractional parts of the cube roots of the
       first 64 primes 2..311
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
    0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
    0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
    0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
    0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
    0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
    0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
    0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
    0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
};

// Macros for SHA-256 operations
#define ROTLEFT(a,b)  (((a) << (b)) | ((a) >> (32-(b))))
#define ROTRIGHT(a,b) (((a) >> (b)) | ((a) << (32-(b))))

#define CH(x,y,z)  (((x) & (y)) ^ (~(x) & (z)))
#define MAJ(x,y,z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))
#define EP0(x)     (ROTRIGHT(x,2) ^ ROTRIGHT(x,13) ^ ROTRIGHT(x,22))
#define EP1(x)     (ROTRIGHT(x,6) ^ ROTRIGHT(x,11) ^ ROTRIGHT(x,25))
#define SIG0(x)    (ROTRIGHT(x,7) ^ ROTRIGHT(x,18) ^ ((x) >> 3))
#define SIG1(x)    (ROTRIGHT(x,17) ^ ROTRIGHT(x,19) ^ ((x) >> 10))

// SHA-256 transformation function
void sha256_transform(SHA256_CTX *ctx, const BYTE data[]) {
    WORD a, b, c, d, e, f, g, h, i, t1, t2, m[64];
```

```
35
36      // Prepare message schedule array
37      for (i = 0; i < 16; ++i)
38          m[i] = (data[i * 4] << 24) |
39                 (data[i * 4 + 1] << 16) |
40                 (data[i * 4 + 2] << 8) |
41                 (data[i * 4 + 3]);
42      for (; i < 64; ++i)
43          m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15]) + m[i - 16];
44
45      // Initialize working variables
46      a = ctx->state[0];
47      b = ctx->state[1];
48      c = ctx->state[2];
49      d = ctx->state[3];
50      e = ctx->state[4];
51      f = ctx->state[5];
52      g = ctx->state[6];
53      h = ctx->state[7];
54
55      // Compression function main loop
56      for (i = 0; i < 64; ++i) {
57          t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
58          t2 = EP0(a) + MAJ(a,b,c);
59          h = g;
60          g = f;
61          f = e;
62          e = d + t1;
63          d = c;
64          c = b;
65          b = a;
66          a = t1 + t2;
67      }
68
69      // Add compressed chunk to current hash value
70      ctx->state[0] += a;
71      ctx->state[1] += b;
72      ctx->state[2] += c;
73      ctx->state[3] += d;
74      ctx->state[4] += e;
75      ctx->state[5] += f;
76      ctx->state[6] += g;
77      ctx->state[7] += h;
78  }
79
80  // SHA-256 initialization function
```

```
81  void sha256_init(SHA256_CTX *ctx) {
82      ctx->datalen = 0;
83      ctx->bitlen = 0;
84      ctx->state[0] = 0x6a09e667;
85      ctx->state[1] = 0xbb67ae85;
86      ctx->state[2] = 0x3c6ef372;
87      ctx->state[3] = 0xa54ff53a;
88      ctx->state[4] = 0x510e527f;
89      ctx->state[5] = 0x9b05688c;
90      ctx->state[6] = 0x1f83d9ab;
91      ctx->state[7] = 0x5be0cd19;
92  }
93
94  // SHA-256 update function
95  void sha256_update(SHA256_CTX *ctx, const BYTE data[], size_t len) {
96      for (size_t i = 0; i < len; ++i) {
97          ctx->data[ctx->datalen] = data[i];
98          ctx->datalen++;
99          if (ctx->datalen == 64) {
100             sha256_transform(ctx, ctx->data);
101             ctx->bitlen += 512;
102             ctx->datalen = 0;
103         }
104     }
105 }
106
107 // SHA-256 finalization function
108 void sha256_final(SHA256_CTX *ctx, BYTE hash[]) {
109     WORD i = ctx->datalen;
110
111     // Pad whatever data is left in the buffer
112     if (ctx->datalen < 56) {
113         ctx->data[i++] = 0x80;
114         while (i < 56)
115             ctx->data[i++] = 0x00;
116     } else {
117         ctx->data[i++] = 0x80;
118         while (i < 64)
119             ctx->data[i++] = 0x00;
120         sha256_transform(ctx, ctx->data);
121         memset(ctx->data, 0, 56);
122     }
123
124     // Append the total message's length in bits
125     ctx->bitlen += ctx->datalen * 8;
126     ctx->data[63] = ctx->bitlen;
```

```
127     ctx->data[62] = ctx->bitlen >> 8;
128     ctx->data[61] = ctx->bitlen >> 16;
129     ctx->data[60] = ctx->bitlen >> 24;
130     ctx->data[59] = ctx->bitlen >> 32;
131     ctx->data[58] = ctx->bitlen >> 40;
132     ctx->data[57] = ctx->bitlen >> 48;
133     ctx->data[56] = ctx->bitlen >> 56;
134     sha256_transform(ctx, ctx->data);
135 
136     // Copy the final state to the output hash
137     for (i = 0; i < 4; ++i) {
138         hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
139         hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
140         hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
141         hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
142         hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
143         hash[i + 20] = (ctx->state[5] >> (24 - i * 8)) & 0x000000ff;
144         hash[i + 24] = (ctx->state[6] >> (24 - i * 8)) & 0x000000ff;
145         hash[i + 28] = (ctx->state[7] >> (24 - i * 8)) & 0x000000ff;
146     }
147 }
148 
149 // Helper function to compute SHA-256 hash of data
150 void sha256_hash(const BYTE data[], size_t length, BYTE hash_output[]) {
151     SHA256_CTX ctx;
152     sha256_init(&ctx);
153     sha256_update(&ctx, data, length);
154     sha256_final(&ctx, hash_output);
155 }
156 
157 // Helper function to compute SHA-256 hash over a specific range
158 void sha256_hash_range(const BYTE data[], uint32_t start_idx, uint32_t
        end_idx, BYTE hash_output[]) {
159     SHA256_CTX ctx;
160     sha256_init(&ctx);
161     sha256_update(&ctx, data + start_idx, end_idx - start_idx);
162     sha256_final(&ctx, hash_output);
163 }
164 
165 // Function to convert bytes to hexadecimal string
166 std::string bytes_to_hex_string(const BYTE* bytes, size_t length) {
167     std::stringstream ss;
168     ss << std::hex << std::setfill('0');
169     for (size_t i = 0; i < length; ++i) {
170         ss << std::setw(2) << static_cast<int>(bytes[i]);
171     }
```

```
172     return ss.str();
173 }
174 
175 // Helper function to compute SHA-256 hash and return as hexadecimal
    string
176 std::string sha256_hash_string(const BYTE data[], size_t length) {
177     BYTE hash[SHA256_BLOCK_SIZE];
178     sha256_hash(data, length, hash);
179     return bytes_to_hex_string(hash, SHA256_BLOCK_SIZE);
180 }
```

Code 4: Baseline SHA256 Implementation

7.3.2 Optimized SHA

```
1  #include <stdlib.h>
2  #include <memory.h>
3  #include <arm_neon.h>
4  #include "sha256.h"
5  
6  // #define ROLLED_UP_METHOD
7  
8  static const WORD k[64] = {
9      0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5,0x3956c25b,0x59f111f1,0
       x923f82a4,0xab1c5ed5,
10     0xd807aa98,0x12835b01,0x243185be,0x550c7dc3,0x72be5d74,0x80deb1fe,0
       x9bdc06a7,0xc19bf174,
11     0xe49b69c1,0xefbe4786,0x0fc19dc6,0x240ca1cc,0x2de92c6f,0x4a7484aa,0
       x5cb0a9dc,0x76f988da,
12     0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7,0xc6e00bf3,0xd5a79147,0
       x06ca6351,0x14292967,
13     0x27b70a85,0x2e1b2138,0x4d2c6dfc,0x53380d13,0x650a7354,0x766a0abb,0
       x81c2c92e,0x92722c85,
14     0xa2bfe8a1,0xa81a664b,0xc24b8b70,0xc76c51a3,0xd192e819,0xd6990624,0
       xf40e3585,0x106aa070,
15     0x19a4c116,0x1e376c08,0x2748774c,0x34b0bcb5,0x391c0cb3,0x4ed8aa4a,0
       x5b9cca4f,0x682e6ff3,
16     0x748f82ee,0x78a5636f,0x84c87814,0x8cc70208,0x90befffa,0xa4506ceb,0
       xbef9a3f7,0xc67178f2
17 };
18 
19 /****************************** TYPES ******************************/
20 struct hash_s {
21     uint32x4_t abcd, efgh;
22 };
23 typedef struct hash_s hash_state_t;
```

```
24
25 /********************** FUNCTION DEFINITIONS **********************/
26 void sha256_transform(SHA256_CTX *ctx, const BYTE data[])
27 {
28     WORD i;
29     uint32x4_t round_input;
30     hash_state_t cur_state, old_state;
31 #ifdef ROLLED_UP_METHOD
32     uint32x4_t sched[4];
33
34     // load state
35     cur_state.abcd = vld1q_u32(ctx->state);
36     cur_state.efgh = vld1q_u32(ctx->state+4);
37
38     // make the schedule
39     for (i = 0; i < 4; i++) {
40         // reverses the byte ordering
41         sched[i] = vreinterpretq_u32_u8(vrev32q_u8(vld1q_u8(data + 16*i)))
               ;
42         round_input = vaddq_u32(sched[i], vld1q_u32(k + i*4));
43         old_state.abcd = cur_state.abcd;
44         old_state.efgh = cur_state.efgh;
45         cur_state.abcd = vsha256hq_u32 (old_state.abcd, old_state.efgh,
               round_input);
46         cur_state.efgh = vsha256h2q_u32(old_state.efgh, old_state.abcd,
               round_input);
47     }
48     for (i = 4; i < 16; i++) {
49         sched[i%4] = vsha256su1q_u32(vsha256su0q_u32(
50                     sched[i%4], sched[(i+1)%4]),
51                     sched[(i+2)%4], sched[(i+3)%4]);
52         round_input = vaddq_u32(sched[i%4], vld1q_u32(k + i*4));
53         old_state.abcd = cur_state.abcd;
54         old_state.efgh = cur_state.efgh;
55         cur_state.abcd = vsha256hq_u32 (old_state.abcd, old_state.efgh,
               round_input);
56         cur_state.efgh = vsha256h2q_u32(old_state.efgh, old_state.abcd,
               round_input);
57     }
58 #else
59     uint32x4_t sched[16];
60
61     // make the schedule
62     for (i = 0; i < 4; i++) {
63         // reverses the byte ordering
```

```
64          sched[i] = vreinterpretq_u32_u8(vrev32q_u8(vld1q_u8(data + 16*i)))
                ;
65      }
66      for (i = 4; i < 16; i++) {
67          sched[i] = vsha256su1q_u32(vsha256su0q_u32(
68                  sched[i-4], sched[i-3]), sched[i-2], sched[i-1]);
69      }
70
71      // load state
72      cur_state.abcd = vld1q_u32(ctx->state);
73      cur_state.efgh = vld1q_u32(ctx->state+4);
74
75      // do the hashing
76      for (i = 0; i < 16; i++) {
77          round_input = vaddq_u32(sched[i], vld1q_u32(k + i*4));
78          old_state.abcd = cur_state.abcd;
79          old_state.efgh = cur_state.efgh;
80          cur_state.abcd = vsha256hq_u32 (old_state.abcd, old_state.efgh,
                round_input);
81          cur_state.efgh = vsha256h2q_u32(old_state.efgh, old_state.abcd,
                round_input);
82      }
83  #endif
84
85      // add in the state
86      vst1q_u32(ctx->state, vaddq_u32(cur_state.abcd, vld1q_u32(ctx->state))
            );
87      vst1q_u32(ctx->state+4, vaddq_u32(cur_state.efgh, vld1q_u32(ctx->state
            +4)));
88  }
89
90  void sha256_init(SHA256_CTX *ctx)
91  {
92      ctx->datalen = 0;
93      ctx->bitlen = 0;
94      ctx->state[0] = 0x6a09e667;
95      ctx->state[1] = 0xbb67ae85;
96      ctx->state[2] = 0x3c6ef372;
97      ctx->state[3] = 0xa54ff53a;
98      ctx->state[4] = 0x510e527f;
99      ctx->state[5] = 0x9b05688c;
100     ctx->state[6] = 0x1f83d9ab;
101     ctx->state[7] = 0x5be0cd19;
102 }
103
104 void sha256_update(SHA256_CTX *ctx, const BYTE data[], size_t len)
```

```
105 {
106     WORD i;
107 
108     for (i = 0; i < len; ++i) {
109         ctx->data[ctx->datalen] = data[i];
110         ctx->datalen++;
111         if (ctx->datalen == 64) {
112             sha256_transform(ctx, ctx->data);
113             ctx->bitlen += 512;
114             ctx->datalen = 0;
115         }
116     }
117 }
118 
119 void sha256_final(SHA256_CTX *ctx, BYTE hash[])
120 {
121     WORD i;
122 
123     i = ctx->datalen;
124 
125     // Pad whatever data is left in the buffer.
126     if (ctx->datalen < 56) {
127         ctx->data[i++] = 0x80;
128         while (i < 56)
129             ctx->data[i++] = 0x00;
130     }
131     else {
132         ctx->data[i++] = 0x80;
133         while (i < 64)
134             ctx->data[i++] = 0x00;
135         sha256_transform(ctx, ctx->data);
136         memset(ctx->data, 0, 56);
137     }
138 
139     // Append to the padding the total message length in bits and
        transform.
140     ctx->bitlen += (uint64_t)ctx->datalen * 8;
141     ctx->data[63] = (BYTE)(ctx->bitlen);
142     ctx->data[62] = (BYTE)(ctx->bitlen >> 8);
143     ctx->data[61] = (BYTE)(ctx->bitlen >> 16);
144     ctx->data[60] = (BYTE)(ctx->bitlen >> 24);
145     ctx->data[59] = (BYTE)(ctx->bitlen >> 32);
146     ctx->data[58] = (BYTE)(ctx->bitlen >> 40);
147     ctx->data[57] = (BYTE)(ctx->bitlen >> 48);
148     ctx->data[56] = (BYTE)(ctx->bitlen >> 56);
149     sha256_transform(ctx, ctx->data);
```

```
150
151     // Since this implementation uses little endian byte ordering and SHA
            uses big endian,
152     // reverse all the bytes when copying the final state to the output
            hash.
153     for (i = 0; i < 8; i += 4) {
154         vst1q_u8(hash + (i * 4),
155                  vrev32q_u8(vld1q_u8((BYTE*)&ctx->state[i])));
156     }
157 }
158
159 void sha256_hash(SHA256_CTX* ctx, const BYTE data[], BYTE hash[], size_t
        dataLen)
160 {
161     sha256_init(ctx);
162     sha256_update(ctx, data, dataLen);
163     sha256_final(ctx, hash);
164 }
165
166 void sha256_hash_file(SHA256_CTX* ctx, const BYTE data[], BYTE hash[],
        size_t dataLen)
167 {
168     sha256_init(ctx);
169     sha256_update(ctx, data, dataLen);
170     sha256_final(ctx, hash);
171 }
```

Code 5: Optimized SHA Implementation

7.4 Deduplication

```
1  int64_t dedup(const SHA256Hash& sha_fingerprint,
2                std::unordered_map<SHA256Hash, int64_t, SHA256HashHash>&
                     chunk_hash_map,
3                int64_t& lzw_chunk_index) {
4      auto it = chunk_hash_map.find(sha_fingerprint);
5
6      if (it == chunk_hash_map.end()) {
7          // New chunk
8          chunk_hash_map.emplace(sha_fingerprint, lzw_chunk_index);
9          ++lzw_chunk_index;
10         return -1;
11     } else {
12         // Duplicate chunk
13         return it->second;
14     }
```

```
}
```

Code 6: Deduplication Implementation

7.5 LZW ARM Mapped Design

```
#include "lzw.h"

// Constants
#define CODE_LENGTH 13                       // Code length in bits
#define CODE_MASK ((1 << CODE_LENGTH) - 1)  // Mask to extract CODE_LENGTH bits
#define MAX_DICT_SIZE (1 << CODE_LENGTH)    // Maximum dictionary size

uint16_t dictionary[MAX_DICT_SIZE][2] = {0};  // Global dictionary

void initialize_dictionary() {
    static bool initialized = false;  // Static flag to ensure one-time initialization
    if (!initialized) {
        for (uint16_t i = 0; i < 256; ++i) {
            dictionary[i][0] = 0xFFFF;  // No prefix for single characters
            dictionary[i][1] = i;       // Character itself
        }
        initialized = true;  // Set flag to true after initialization
    }
}

void lzw_encode(const unsigned char *input_data, uint16_t input_size, uint8_t compressed_data[OUTPUT_SIZE],
                uint32_t *compressed_size_buf) {
    if (input_size == 0) {
        // Handle empty input
        *compressed_size_buf = 0;
        return;
    }
    // Ensure the dictionary is initialized before using
    initialize_dictionary();

    uint16_t dict_size = 256;               // Next available code in the dictionary
    uint16_t current_code = input_data[0];  // Current prefix code
    uint16_t code_words[INPUT_SIZE];        // Store the encoded code words
    uint16_t code_word_count = 0;

```

```
36      // LZW Compression Algorithm
37      for (uint16_t i = 1; i < input_size; ++i) {
38          uint8_t next_char = input_data[i];
39          bool found = false;
40
41          // Search for the combined string in the dictionary
42          for (uint16_t j = 256; j < dict_size; ++j) {
43              if (dictionary[j][0] == current_code && dictionary[j][1] ==
                    next_char) {
44                  current_code = j;
45                  found = true;
46                  break;
47              }
48          }
49
50          if (!found) {
51              // Output the current_code
52              code_words[code_word_count++] = current_code;
53
54              if (dict_size >= MAX_DICT_SIZE - 1) {
55                  std::cerr << "Debug: Dictionary is near full. Current size
                        : " << dict_size << std::endl;
56              }
57              // Add new string to the dictionary if there's space
58              if (dict_size < MAX_DICT_SIZE) {
59                  dictionary[dict_size][0] = current_code;
60                  dictionary[dict_size][1] = next_char;
61                  ++dict_size;
62              }
63
64              current_code = next_char;  // Start new string with next_char
65          }
66      }
67
68      // Output the code for the last string
69      code_words[code_word_count++] = current_code;
70
71      // Pack code words into bytes (MSB-first)
72      uint32_t bit_buffer = 0;
73      uint16_t bit_count = 0;
74      uint32_t compressed_size = 0;
75
76      for (uint16_t i = 0; i < code_word_count; ++i) {
77          uint16_t code = code_words[i] & CODE_MASK;
78          bit_buffer = (bit_buffer << CODE_LENGTH) | code;
79          bit_count += CODE_LENGTH;
```

```
80
81          // Extract bytes from the bit buffer
82          while (bit_count >= 8) {
83              uint8_t byte = (bit_buffer >> (bit_count - 8)) & 0xFF;
84              compressed_data[compressed_size++] = byte;
85              bit_count -= 8;
86          }
87      }
88
89      // Handle remaining bits (if any)
90      if (bit_count > 0) {
91          uint8_t byte = (bit_buffer << (8 - bit_count)) & 0xFF;
92          compressed_data[compressed_size++] = byte;
93      }
94      if (compressed_size == 0) {
95          std::cerr << "Error: Compressed size is zero. This should not
                happen for valid input.\n";
96          *compressed_size_buf = 0;
97          return;
98      }
99      // Write compressed size back to global memory
100     *compressed_size_buf = compressed_size;
101 }
```

Code 7: LZW Software Implementation

7.6 LZW Hardware Implementation

```
1  #include "lzw_hw.h"
2  uint16_t hw_dictionary[MAX_DICT_SIZE][2] = {0};  // Global dictionary
3
4  void hw_initialize_dictionary() {
5  #pragma HLS INLINE
6      static bool initialized = false;  // Static flag to ensure one-time
           initialization
7      if (!initialized) {
8          for (uint16_t i = 0; i < 256; ++i) {
9  #pragma HLS UNROLL
10             hw_dictionary[i][0] = 0xFFFF;  // No prefix for single
                   characters
11             hw_dictionary[i][1] = i;       // Character itself
12         }
13         initialized = true;  // Set flag to true after initialization
14     }
15 }
16
```

```
17 void lzw_encode_hw(const unsigned char *input_data, uint8_t input_size,
       uint8_t compressed_data[OUTPUT_SIZE],
18                     uint32_t *compressed_size_buf) {
19     // if (input_size == 0) {
20     //     // Handle empty input
21     //     *compressed_size_buf = 0;
22     //     return;
23     // }
24     // Ensure the dictionary is initialized before using
25     hw_initialize_dictionary();
26
27 #pragma HLS ARRAY_PARTITION variable = hw_dictionary dim = 1 block factor
       = 8
28 #pragma HLS BIND_STORAGE variable = hw_dictionary type = ram_1p impl =
       bram
29
30     uint16_t dict_size = 256;               // Next available code in the
          dictionary
31     uint16_t current_code = input_data[0];  // Current prefix code
32     uint16_t code_words[INPUT_SIZE];        // Store the encoded code
          words
33     uint16_t code_word_count = 0;
34
35     // LZW Compression Algorithm
36     for (uint16_t i = 1; i < input_size; ++i) {
37         //#pragma HLS PIPELINE II = 1
38         uint8_t next_char = input_data[i];
39         bool found = false;
40
41         // Search for the combined string in the dictionary
42         for (uint16_t j = 256; j < dict_size; ++j) {
43             if (hw_dictionary[j][0] == current_code && hw_dictionary[j][1]
                   == next_char) {
44                 current_code = j;
45                 found = true;
46                 break;
47             }
48         }
49
50         if (!found) {
51             // Output the current_code
52             code_words[code_word_count++] = current_code;
53
54             // Add new string to the dictionary if there's space
55             if (dict_size < MAX_DICT_SIZE) {
56                 hw_dictionary[dict_size][0] = current_code;
```

```
                hw_dictionary[dict_size][1] = next_char;
                ++dict_size;
            }

            current_code = next_char;  // Start new string with next_char
        }
    }

    // Output the code for the last string
    code_words[code_word_count++] = current_code;

    // Pack code words into bytes (MSB-first)
    uint32_t bit_buffer = 0;
    int bit_count = 0;
    uint32_t compressed_size = 0;

    for (uint16_t i = 0; i < code_word_count; ++i) {
        uint16_t code = code_words[i] & CODE_MASK;
        bit_buffer = (bit_buffer << CODE_LENGTH) | code;
        bit_count += CODE_LENGTH;

        // Extract bytes from the bit buffer
        while (bit_count >= 8) {
            uint8_t byte = (bit_buffer >> (bit_count - 8)) & 0xFF;
            compressed_data[compressed_size++] = byte;
            bit_count -= 8;
        }
    }

    // Handle remaining bits (if any)
    if (bit_count > 0) {
        uint8_t byte = (bit_buffer << (8 - bit_count)) & 0xFF;
        compressed_data[compressed_size++] = byte;
    }
    // Write compressed size back to global memory
    *compressed_size_buf = compressed_size;
}
```

Code 8: LZW FPGA Implementation

7.7 Relevant Encoder Code

```
bool process_packet(unsigned char *file, size_t file_size, std::vector<
    unsigned char> &output_data) {
    // Step 1: Perform Content-Defined Chunking
    std::vector<size_t> chunk_positions;
```

```
4       cdc_time.start();
5       content_defined_chunking(file, file_size, chunk_positions);
6       cdc_time.stop();
7
8       size_t chunk_count = (chunk_positions.empty()) ? 0 : chunk_positions.
            size() - 1;
9       std::cout << "CDC function called: chunk_count = " << chunk_count <<
            std::endl;
10
11      size_t original_size = file_size;
12
13      size_t estimated_unique_chunks = 1000000; // Adjust based on expected
            data
14      std::unordered_map<SHA256Hash, int64_t, SHA256HashHash> chunk_hash_map
            ;
15      chunk_hash_map.reserve(estimated_unique_chunks);
16
17      int64_t lzw_chunk_index = 0;  // Starts from 0
18
19      for (size_t i = 0; i < chunk_count; i++) {
20          size_t start = chunk_positions[i];
21          size_t end = chunk_positions[i + 1];
22          uint16_t chunk_size = end - start;
23          unsigned char *chunk = file + start;
24
25          if (chunk_size == 0 || chunk_size > MAX_CHUNK_SIZE) {
26              std::cerr << "Error: Invalid chunk size " << chunk_size << "
                    for chunk " << i << std::endl;
27              continue;
28          }
29
30          BYTE hash_output[SHA256_BLOCK_SIZE];
31          sha_time.start();
32          SHA256_CTX ctx;
33          sha256_hash(&ctx, chunk, hash_output, chunk_size);
34          sha_time.stop();
35          SHA256Hash hash_value;
36          std::memcpy(hash_value.data(), hash_output, 32);
37
38          dedup_time.start();
39          int64_t duplicate_index = dedup(hash_value, chunk_hash_map,
                lzw_chunk_index);
40          dedup_time.stop();
41
42          if (duplicate_index != -1) {
43              uint32_t header = (duplicate_index << 1) | 0x1;  // Bit 0 is 1
```

```
44              append_uint32_le(output_data, header);
45              deduplication_contribution += 4;
46              total_compressed_size += 4;  // Only the header for duplicate
                    chunks
47              input_size_dedup += chunk_size;
48              dedup_chunk++;
49          } else {
50              uint8_t compressed_data[OUTPUT_SIZE];
51              uint32_t compressed_size_buf = 0;  // Initialize explicitly
52
53              lzw_time.start();
54              lzw_encode(chunk, chunk_size, compressed_data, &
                    compressed_size_buf);
55              lzw_time.stop();
56
57              // Assign the dereferenced value to a new variable
58              uint32_t compressed_size = compressed_size_buf;
59              if (compressed_size == 0) {
60                  std::cerr << "Error: Compressed size is zero for chunk "
                        << i << std::endl;
61                  // Print chunk information and content
62                  std::cout << "Chunk " << i << ": Start=" << start << ",
                        End=" << end << ", Size=" << chunk_size
63                            << " bytes, Data=\"" << std::string(chunk, chunk
                                + chunk_size) << "\"" << std::endl;
64              }
65              uint32_t header = (compressed_size << 1) | 0;
66              append_uint32_le(output_data, header);
67              output_data.insert(output_data.end(), compressed_data,
                    compressed_data + compressed_size);
68
69              lzw_contribution += 4 + compressed_size;
70              total_compressed_size += 4 + compressed_size;
71              input_size_lzw += chunk_size;
72              lzw_chunk++;
73          }
74      }
75      return true;
76  }
```

Code 9: Encoder.cpp (Only relevant part appended)