Description & Implementation of the ACMI Component

A Hardware Accelerator for the Vision Transformer Algorithm using MinHash to Lineage assignment by ENICS Lab
Final Project - Computer Engineering, BIU

NOAM DIAMANT & ITAY GOLDBERG
Academic Supervisor : Prof. Leonid Yavits
Instructor : Itay Merlin

May 22, 2025

הפקולטה להנדסה

ע"ש אלכסנדר קופקין

אוניברסיטת בר־אילן

Contents

Chapter 1

Theoretical Background

1.1 Minhash

MinHash is a variant of locality-sensitive hashing designed to approximate the similarity between two datasets. The fundamental idea behind MinHash is that the probability that two sets have the same minimum hash value is equivalent to their Jaccard similarity. Jaccard similarity measures the proportion of shared elements between two sets equal to:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{1.1}$$

We use the Bottom-k Minhash whose overview is presented in Fig. 1.1 In the genomic context, the datasets comprise sequences of kmers. The signatures A and B are generated by a single hash function, h, which ranks the kmers by assigning a numerical value to each kmer. The k minimal ranking values comprise the signature subsets $S(A)$ and $S(B)$. The equation in Fig. 1.1 approximates Jaccard similarity between the two original kmer sequences.

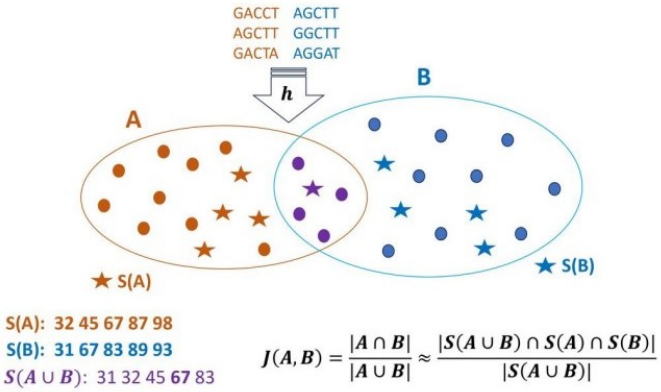


Figure 1.1: Minhash Bottom-k overview

We leverage Bottom-k to convert low-coverage genomes into sequences of fixed-size fragments. Genomic samples that come from similar genomes are likely to have overlapping hash values, so that Bottom-k is instrumental in capturing their similarity. Traditional use of minHash with bottom-k allows sampling of a small amount of information from the entire set in a way that will enable direct calculation of the Jaccard similarity. When we deal with biological sequences we are exposed to a particularly large amount of information, therefore a direct calculation of Jaccard similarity will be inefficient in terms of

memory and the duration of the operation, hence we will use this project in a more advanced solution using a neural network.

1.2 DNA Sequencing and Genome Analysis

DNA is composed of four nucleotides: Adenine (A), Guanine (G), Cytosine (C), and Thymine (T), which are frequently referred to as DNA base pairs, bases, or bps. Accordingly, a DNA data element is a DNA base that can have one of four values (A, G, C, and T). DNA sequencing is the process of determining the bases of a DNA chain in a given biological sample. Contemporary high-throughput DNA sequencers can sequence multiple biological samples in parallel. A DNA sequencing process, along with genome analysis, is carried out in several steps: (1) sample preparation; (2) DNA sequencing that generates multiple DNA reads (i.e., sub-sequences of the DNA sample); and (3) DNA classification, DNA read alignment, genome assembly, variant analysis, etc. While variant refers to a single change in a genome (i.e., a single base change mutation), lineage is a collection of variants that help define a specific line of an organism.

1.3 ViRAL Algorithm

ViRAL is the platform for a quick and accurate identification, classification, and lineage assignment of SARS-CoV-2 variants. It accepts an assembled genome and outputs the ordered list of most probable lineages such genome belongs to. If the top probabilities output by ViRAL are similar, this might indicate that the new genome belongs to an unknown lineage, creating a new node on the phylogenetic tree.

The top-level overview of ViRAL algorithm is presented in Figure 1.2. The input to ViRAL is an assembled genome in FASTA file format. To convert the genome into a sequence of feature vectors, we build a pipeline of preprocessing steps, comprising the feature extractor and the embedding layer. The feature extractor chooses a set of fragments (i.e., genome subsequences of fixed length), represents each fragment as a two-dimensional matrix, and outputs those genome fragment matrices (GFMs). Each GFM is fed into an embedding layer that consists of one neuron. This layer converts the GFM into a genome fragment vector, designated the feature vector. The feature vectors are fed into the ViT, which outputs the most likely assignment candidates (by attaching probabilities to each lineage).

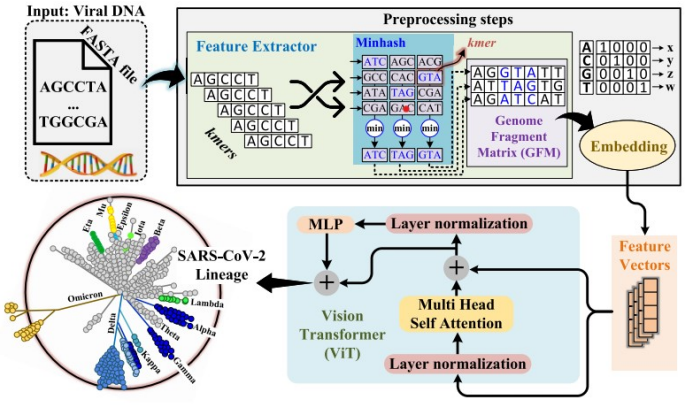


Figure 1.2: A general overview of the ViRAL algorithm.

The feature extractor and the embedding layer transform a genome into a sequence of representative feature vectors, which are the numerical representations of genome fragments. We implement the fea-

ture extractor using MinHash to preserve similarities in the genome. It allows the feature extractor to find features in the query genome that are shared by the known lineages. The embedding layer is used as a dimensionality reduction step. It allows for the reduction of ViRAL latency without affecting its assignment accuracy.

Chapter 2

Introduction

In this chapter, an extensive literature review was conducted in order to cover all possible subjects that affect the goals of this project. Understanding how the different components and architectures that are in use today and the modern solutions to the problem this work is trying to solve are important to better define the objectives of this research.

2.1 The purpose of this project

The article introduces a neural network-based algorithm designed to enhance the accuracy of lineage assignment. It suggests that creating dedicated hardware can further accelerate the calculations. While software-only (CPU-based) and GPU-accelerated solutions perform relatively well, they still face limitations. The preprocessing stage is responsible for 95% of ViRAL's execution time, and it is a prime target for acceleration due to conventional computers' lack of optimization for Locality-Sensitive Hashing operations. Within MinHash, components like signature sorting exhibit latency that scales as nlog(n), where n represents genome size, further compounding the issue as genomes grow larger. Consequently, to effectively scale and accelerate ViRAL, focusing on MinHash optimization is crucial. In this project, we implemented the MinHash accelerator component responsible for data preparation before activating the neural network using the SystemVerilog language. This preprocessing component ensures that the data is correctly formatted and structured for the neural network, ultimately contributing to the system's overall performance improvements.
In the following pages, we will present key parts of the article, the application of the various subcomponents, and the results of the synthesis of the component as a whole. All of these are a confirmation of the functionality of the hardware accelerator proposed in the article.

2.2 MinHash Accelerators

Several MinHash acceleration solutions have been proposed. MetaCache GPU uses MinHash for metagenomic classification and implements it on GPU. MSIM is a near-memory MinHash accelerator with a limited energy consumption reduction (26.4× vs. high-performance GPU), targeted for high-performance applications. In MinHash is used for kmer clustering. An FPGA-based solution is limited to parallel calculation of 15 hash functions (whereas ViRAL calculates in parallel at least 256 hash functions). JACK-FPGA uses a cloud FPGA to accelerate MinHash. Scotch is an FPGA-based locality-sensitive hashing accelerator with a limited energy consumption gain (5× over high-performance GPU).

The purpose of MinHash acceleration in ViRAL framework is to balance the MinHash and Vision Transformer calculations while providing the highest energy efficiency, preferably sufficient for portable applications.

The MinHash Scheme

A modified MinHash is the second processing component of ViRAL, whose role is extracting the informative feature vectors from the genome. MinHash, or the min-wise independent permutations scheme is a technique for similarity estimation. MinHash is used in many tasks in computational biology, including genome assembly, metagenomic gene clustering, and genomic distance estimation. MinHash implements the following sketch function:

Given a set of characters A, a compression factor n, and a hash function h, the elements of the set A are hashed using function h to generate the set $H(A)$. Then the elements of $H(A)$ are sorted and the inputs to the smallest n elements are returned as described in Algorithm 1.

Algorithm 1: The Sketch Algorithm `sketch(A)`

Require: set $A = \{a_1, \ldots, a_{|A|}\}$, compression parameter n and a hash function h
`sketch` $\leftarrow \emptyset$
$H(A) \leftarrow (h(a_1), \ldots h(a_{|A|}))$
Sort $H(A)$ to get $(h(a_{i1}), h(a_{i2}), \ldots, h(a_{i|A|}))$
return $(a_{i1}, a_{i2}, \ldots, a_{in})$

The subset obtained by applying the sketch algorithm provides a good estimate for the Jaccard index, defined as follows: Given two sets A, B, the Jaccard index is

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Formally, given two sets, A and B,

$$\frac{|sketch(A) \cap sketch(B) \cap sketch(A \cup B)|}{|sketch(A \cup B)|} \approx J(A, B)$$

We use the sketch algorithm to extract feature vectors from a genome, as presented further.

Feature Extractor

The first part of the ViRAL pipeline receives the assembled genome and outputs matrices that represent genome fragments. Feature extractor employs MinHash to find similar fragments (i.e., features) in different genomes.

Algorithm 2: The Feature Extractor

Require: genome g, compression parameter n, fragment size f, kmer size k, hash function h
$G \leftarrow \{gi \equiv g[i : i + k]\}$ s.t. $i \in [0, n - k + 1]$
`kmers` $\leftarrow$ sketch(G, n, h)
`left` $\leftarrow$ floor $\left(\frac{f-k}{2}\right)$
`right` $\leftarrow$ ceil $\left(\frac{f-k}{2}\right)$
`fragments` $\leftarrow \{g[i - \texttt{left}, i + \texttt{right}] : gi \in \texttt{kmers}\}$
return `one-hot-encode(fragments)`

One hot encoding of a fragment refers to the operation of transforming each base in the fragment as follows: $A \rightarrow [1, 0, 0, 0]$; $C \rightarrow [0, 1, 0, 0]$; $G \rightarrow [0, 0, 1, 0]$; $T \rightarrow [0, 0, 0, 1]$; $N \rightarrow [0, 0, 0, 0]$.

For the example please refer to Figure 2.1(b).

An example is illustrated in Figure 2.1(a). Input is a genome sequence g of length $N = 19$. We generate all possible kmers (genome sub-sequences of length k) G (where $k = 4$), and extract ($n = 3$)

4mers from the genome using the MinHash scheme (i.e., `sketch`(G, $n = 3$, h)). The next step is to generate fragments of length $f = 8$ (by extending each of three 4mers by $\frac{f-k}{2} = \frac{8-4}{2} = 2$ bases in each direction). The last step of this workflow is to encode each basepair of the fragments using one-hot encoding.

For the genome sequence g of length N, the extractor first generates a set G of all possible kmers. It then sketches (i.e., applies MinHash sketch function to) the set of kmers, G, to extract n representative kmers of the genome (Algorithm 1). Those kmers are used as anchors to be expanded to generate fragments. The last part is transforming a fragment into a numerical matrix where the genome basepairs A, G, C, and T are encoded using one-hot encoding as described in Algorithm 2.

Embedding Layer

The input of the embedding layer is a genome fragment matrix of dimensions $f \times 4$ (i.e., f represents the fragment length and 4 is the dimensionality of the basepair one-hot encoding). It outputs a feature vector of size f. The embedding layer consists of one neuron which performs a base-wise linear transformation. Specifically, the embedding layer defines 5 learnable parameters, 4 weights w_A, w_C, w_G, w_T, and one bias term b_N. Given a genome fragment matrix, we transform each base (represented in one-hot encoding) to a numerical token as presented in Figure 2.1(b), which also contains an example in which we embed a fragmented matrix of dimensions 8×4.

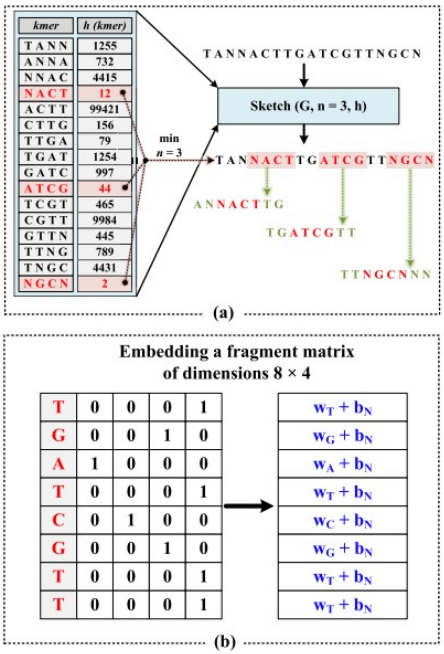


Figure 2.1: (a): Feature extractor workflow example. (b): Embedding a fragment matrix of dimensions 8×4.

2.3 System Architecture

Figure 2.2 illustrates the system view of the ViRAL platform. DNA samples are prepared and sequenced. The assembled genome material is fed into ViRAL, whose main components are MinHash and ViT accelerators, implemented by ACMI, a standalone ACcelerated MInHash ASIC, and Nvidia GPU, respectively.

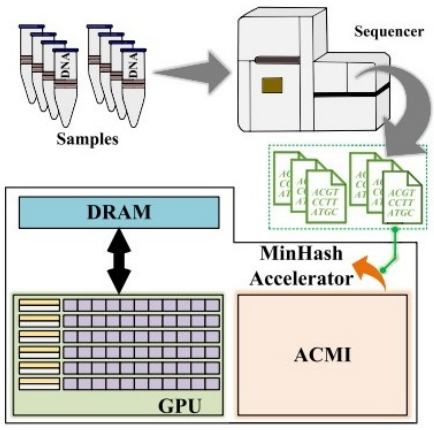


Figure 2.2: System view of the accelerated ViRAL platform featuring MinHash accelerator ACMI, combined with NVidia GPU.

ACMI is a throughput-oriented streaming platform. ACMI (1) inputs a genome $i+1$, (2) processes both the previously received genome i and the genome $i+1$, and (3) outputs the genome i related results, in parallel. ACMI inputs the accession genome one DNA base per cycle (on average). In the case of SARS-CoV-2 virus whose size is approximately 30K DNA bases, it takes approximately 30K cycles to upload a single genome. The result of ACMI is a Genome Fragment Matrix (GFM). Both the compression factor and the fragment_size are 256, hence the GFM is a 256×256 matrix of 2-bits elements (each elements is a base). To balance the pipeline, ACMI is designed to output 4bits/cycle, thus taking approximately $256 \times 256 \times 2/4 = 32K$ cycles to output the GFMs.

Since ACMI operating at 113 MHz requires I/O throughput of less than 1.13 GB/s to achieve its optimal performance, data connectivity can be implemented by PCIe 3.0 or higher.

The ACMI architecture is presented in Figure 2.3. ACMI comprises the Fragment Memory (FM), the kmer buffer, the kmer index counter, Hasher, Sorter, and Extender units. Hasher performs hashing of the input kmer sets using MurmurHash3, which is a non-cryptographic hash function suitable for general hash-based lookup. Sorter sorts the hashed kmers (kmer signatures) and outputs 256 smallest signatures. Extender extends the signatures into larger DNA fragments and generates the GFMs.

While Hasher and Sorter process kmers in a pipelined fashion, "on the fly", the Extender requires access to the entire genome. Therefore, the Fragment Memory (FM) unit is organized as a double buffer, comprising buffers FM1 and FM2, operating in two phases: in phase 1, FM1 receives genome i, which is subsequently hashed and sorted. At the end of the phase, FM1 and FM2 are logically swapped. In phase 2, FM2 receives genome $i+1$, which is being processed by the Hasher and Sorter. At the same time, FM1 serves the Extender, which generates the GFMs of the genome i. At the end of the phase, the buffers are swapped again, and so on. ACMI pipeline timing is shown in Figure 2.4.

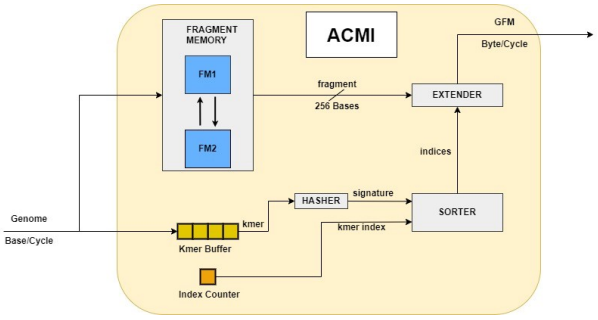


Figure 2.3: The top-level architecture.

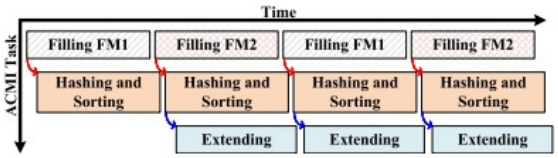


Figure 2.4: ACMI pipeline timing.

2.3.1 Kmer Buffer and Index Counter

The kmer buffer converts the DNA base stream received by ACMI into the stream of kmers (16mers). The index counter keeps track of the kmer index (the position of the kmer on the genome sequence).

2.3.2 Fragment Memory

The Fragment Memory buffers enable parallel write and read. The write operation occurs at a single-base granularity (each base is 2 bits). The read occurs in chunks of 256 bases. As depicted in Figure 2.5, each FM buffer comprises eight 128×32 bases SRAM modules and can store up to 8KB of genomic information. FM address is composed of the row pointer (7 MS-bits), the SRAM module pointer (the following 3 bits), and the byte offset (the byte address within the row, 5 LS-bits). Since the read data granularity is 256 bases, only the 7 MS-bits of the address are used in read access.

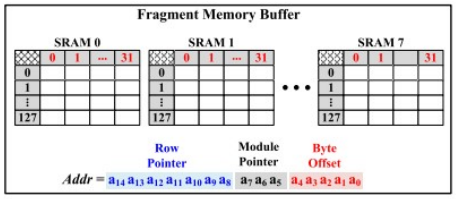


Figure 2.5: A fragment memory buffer comprising 8 128×32 bases SRAM modules.

2.3.3 Sorter

The Sorter maintains a list of compression factor $= 256$ lexicographically smallest kmer signatures. It receives the signatures from the Hasher, along with their indices. After all kmers of the genome are hashed and sorted, Sorter transfers the entire list of 256 indices of the smallest signatures to the Extender (in parallel). At this point, the FM buffers are swapped, and the Extender begins processing the index

data of the recently hashed and sorted genome, while a new genome is being written to the other FM buffer, and hashed and sorted by ACMI in a pipeline fashion.

The hashed kmers (signatures) are transferred sequentially from the Hasher to the Sorter. In each cycle, each comparator in the chain receives from its left neighbor a tuple comprising a kmer signature and its index, and compares such signature with the minimal signature value the comparator stores. If the new signature is smaller (lexicographically) than the stored one, the comparator saves the new tuple and outputs the old tuple, otherwise, it outputs the new tuple. After all, signatures are processed, the comparator chain retains the 256 smallest signatures and their indices (which are afterwards transferred to the Extender).

2.3.4 Extender

Extender converts each of the 256 kmers with the smallest signatures into a 256-base wide DNA fragment, by extending such kmer left and right. To accomplish that, Extender reads the DNA fragments directly from one of the FM buffers, using the index it receives from the Sorter to calculate the FM row pointer.

1. Calculate the fragment position on the genome sequence frag_idx $= \left\lfloor \frac{(kmer\_idx - \lfloor \frac{\text{frag\_len}-\text{kmer\_len}}{2} \rfloor)}{\text{frag\_len}} \right\rfloor$.
2. Read the DNA fragment pointed to by frag_idx.
3. Read the DNA fragment from the next FM buffer row, addressed by frag_idx $+ 1$.
4. Render the extended DNA fragment by concatenating the relevant parts of those two consecutive DNA fragments.

The 256 extended DNA fragments comprise the GFMs that become the output of ACMI.

An example of extension is shown in Figure 2.6. Suppose that the kmer length is 4 (a 4-mer), the fragment length is 8, and the size of the genome is 32. The Extender has to extend the 4-mer with index 9 (i.e., the 4-mer TAAG marked in red in Figure 2.6). In this case,

$$\text{frag\_idx} = \left\lfloor \frac{(kmer\_idx - \frac{\text{frag\_len}-\text{kmer\_len}}{2})}{\text{frag\_len}} \right\rfloor = \left\lfloor \frac{(9 - \frac{8-4}{2})}{8} \right\rfloor = \left\lfloor \frac{7}{8} \right\rfloor = 0.$$

Hence the Extender will read from FM buffer fragments 0 and 1. It will then concatenate the relevant bases using shift operations, render a single 8-base wide DNA fragment, encode the bases using one-hot encoding, and append the encoded extended DNA fragment to the GFMs.

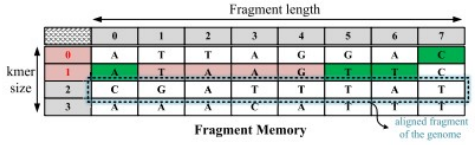


Figure 2.6: Example of extension. The fragment length is 8, the kmer size is 4. Each line represents an aligned fragment of the genome. The bases in pink are the bases of the 4mer at position 9. The extended fragment includes two bases (marked in green) to the left and to the right of the original 4mer.

2.4 Alternatives

The article presents two alternatives that deal with the same problem. We will conduct a brief review and discuss the differences.

2.4.1 UShER

UShER (Ultrafast Sample placement on Existing tRees) is a state-of-the-art tool for SARS-CoV-2 lineage assignment. UShER uses a phylogenetic tree-based approach:

1. It maintains a phylogenetic tree of SARS-CoV-2 genomes.
2. When classifying a new genome, it identifies the optimal placement of the sample on the existing tree.
3. It uses an efficient data structure and algorithm to quickly traverse the tree and find the best placement.

According to the article, UShER performs well at higher coverage levels, outperforming Kraken2 for coverage levels of 4× and above. However, its performance degrades significantly at lower coverage levels. For example, when applied to genomes assembled using Illumina HiSeq 2500 single-end reads, UShER achieves 87.6% accuracy at 8× coverage, but its accuracy drops dramatically at lower coverages. The study uses UShER as a "golden reference" for evaluating ViTAL's performance on novel lineages, indicating its reliability for phylogenetic placement of SARS-CoV-2 genomes.

2.4.2 Kraken2

Kraken2 is another tool used for genome classification and lineage assignment. The article shows that Kraken2 performs better than UShER at very low coverage levels (below 4×), but still falls short of ViTAL's performance. For instance, when classifying genomes assembled from Illumina HiSeq 2500 single-end reads at 1× coverage, Kraken2 achieves an accuracy of 27.4%, which is significantly higher than UShER's 5.4% but much lower than ViTAL's 87.7%. The study also mentions that Kraken2, like other k-mer matching tools, faces limitations in classifying erroneous DNA reads and is not very efficient in SARS-CoV-2 lineage placement due to the limited differences between genomes of separate lineages.

Kraken2 uses a k-mer matching approach:

1. It builds a database of k-mers (short DNA sequences) associated with different taxonomic groups.
2. When classifying a genome, it breaks it into k-mers and looks up each k-mer in the database.
3. It uses a lowest common ancestor (LCA) algorithm to assign taxonomic labels based on the k-mer matches.

This method is fast and efficient for general classification tasks but can struggle with closely related genomes or erroneous reads.

2.4.3 Key Differences

The key difference is that ViTAL's deep learning approach allows it to learn complex patterns from the data, making it more robust to low coverage and sequencing errors. UShER's tree-based method is well-suited for detailed phylogenetic analysis but may require higher quality data. Kraken2's k-mer matching is fast but less flexible in handling errors or closely related genomes. These differences explain why ViTAL outperforms the others, especially in low-coverage scenarios.

Chapter 3

ACMI's Components

3.1 Fragment Memory

3.1.1 Description

The FM module is structured as a dual buffer system with multiple RAM blocks. Its structure is defined by key parameters from `proj_pkg`. The module is clocked, handling write and read operations for storing and retrieving data fragments, specifically for handling genetic data with a base length of 2 bits, representing the four DNA nucleotides (A, T, G, C).

3.1.2 Parameters

Inner Parameter	Package Parameter	Value	Description
BUFFER_COUNT	FM_BUFFER_COUNT	2	Number of buffers
RAMS	FM_RAMS_COUNT	8	Number of RAMs per buffer
ENTRIES	FM_ENTRIES_COUNT	128	Entries per RAM
OFFSET	FM_OFFSET_COUNT	32	Size of offset in each entry
DATA_BITS	FM_DATA_BITS	2	Width of each memory cell
BUFFER_SIZE	FM_BUFFER_SIZE	32768	Total buffer size
INDICE_LEN	INDICE_LEN	15	Length of an index
SIGNED_INDICE_LEN	SIGNED_INDICE_LEN	16	Length of a signed index
FRAG_LEN	FM_EXTENDER_FRAG_LEN_BITS	512	Length of the fragment in bits

3.1.3 Interface

Signal	Direction	Width
`clk`	Input	1
`rst_n`	Input	1
`in_wdata`	Input	DATA_BITS
`chg_idx`	Input	1
`frag_idx`	Input	SIGNED_INDICE_LEN
`out_rdata`	Output	FRAG_LEN
`out_wait`	Output	1

Ports Description

Input Ports:

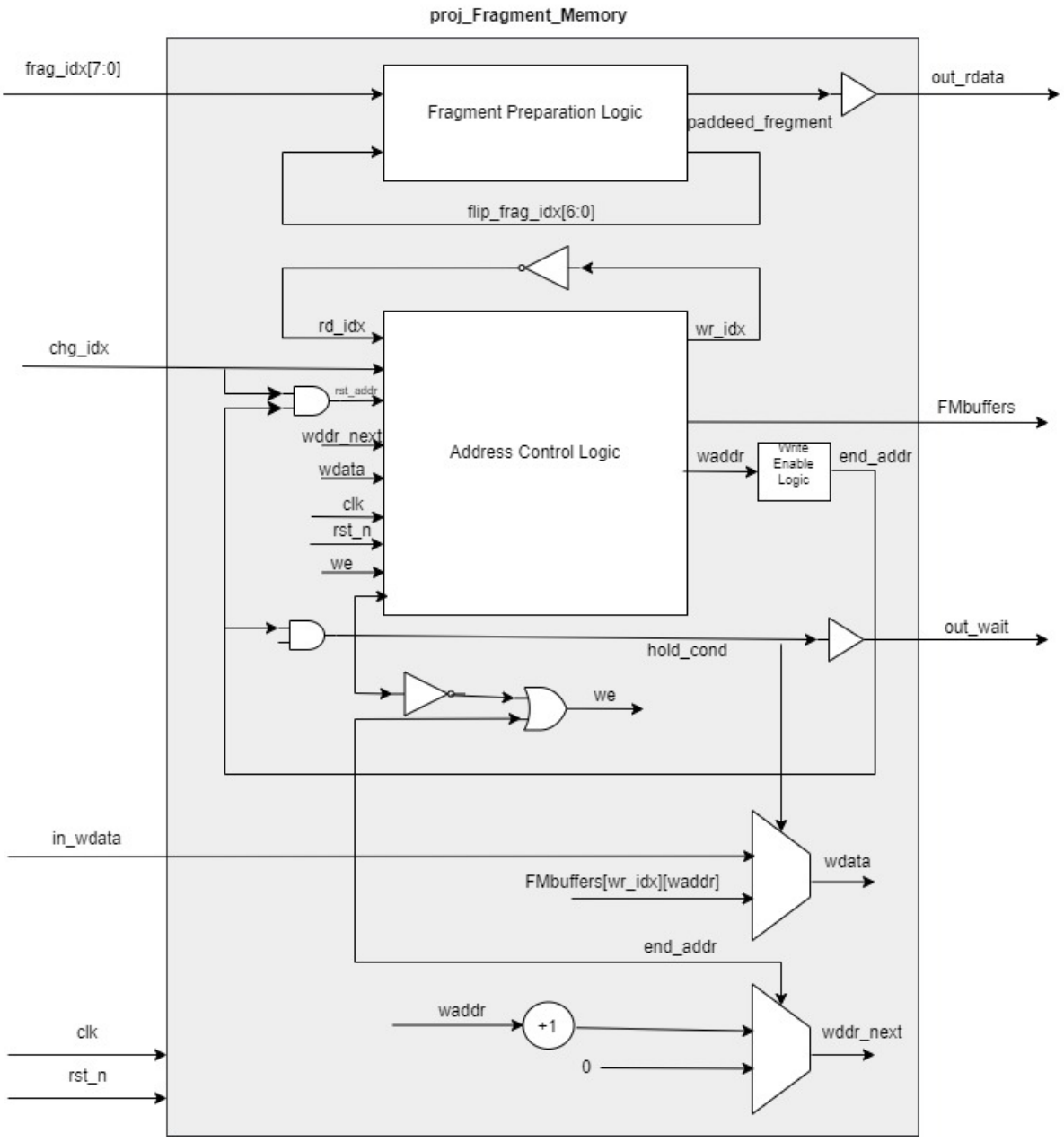
- `clk`: Clock signal

- `rst_n`: Active-low reset signal
- `in_wdata`: Input data to be written
- `chg_idx`: Signal to change active buffer index
- `frag_idx`: Fragment index for reading

Output Ports:

- `out_rdata`: Output data read from buffer
- `out_wait`: Wait signal for the module

3.1.4 Block Diagram



3.1.5 Implementation

The `proj_fm` module is designed to manage a dual-buffer memory system with each buffer consisting of multiple RAMs. The implementation follows a clocked, sequential logic that handles read and write operations on fragments of data. The primary goal is to enable efficient storage and retrieval, particularly for scenarios requiring fast memory switching and processing of indexed data, such as genetic or digital signal processing tasks.

1. Buffer Management:

The module supports two buffers (`FM0` and `FM1`), where data can be written to one buffer while the other buffer is being read. The buffer index can be swapped using the `chg_idx` signal, which switches between read and write operations across the buffers. This enables non-blocking read and write access to the data, essential in high-speed memory applications.

2. Address Control Logic:

The address control logic computes the write address (`waddr`) and read address (`raddr`) dynamically. When the `chg_idx` signal is triggered, the buffer index changes, and the module uses a combination of parameters such as `RAMS`, `ENTRIES`, and `OFFSET` to calculate the memory location where data will be written or read. The calculation ensures that the write and read operations do not overlap unless intended, providing robustness in memory management.

The address logic also handles the buffer size, incrementing or resetting the address counters based on the current buffer position and buffer limits. This is critical in maintaining data integrity and ensuring that data is correctly mapped to its intended location within the buffer.

3. Fragment Preparation Logic:

The fragment preparation logic is responsible for constructing the output data fragment by reading from the appropriate buffer, which can consist of up to 8KB of data at a time. The module uses a combination of the index (`frag_idx`) and zero-padding to manage memory fragments efficiently. The logic handles both signed and unsigned indices, ensuring that negative index values are correctly processed via two's complement conversion (`flip_frag_idx`).

4. Write and Read Operations:

Write operations are governed by the `we` (write enable) signal, which is only asserted when the `hold_cond` signal is not active. The `hold_cond` ensures that writing does not occur when the buffer is in a locked or wait state (`out_wait`). This mechanism prevents data overwriting or corruption when switching between the two buffers.

Similarly, read operations occur concurrently with write operations, with data being read from the fragment buffer that is not currently being written to. The output data (`out_rdata`) is prepared based on the fragment and is padded with zeros if necessary to maintain consistency across varying fragment lengths.

In this implementation, the efficient use of dual buffers allows simultaneous read and write operations, reducing memory access latency and improving throughput in high-speed systems. The fragment preparation logic ensures that the system can handle varying fragment sizes and indexed data efficiently.

3.1.6 Codes & Tests

The example shown below is an example with reduced parameters (not the actual ACMI parameters), for demonstration purposes. At the time point described in the wave diagram, the buffer is already full, and we read from register #1. We begin reading with index `00`, which yields the value `D555`. For index `10h`, the value retrieved is `FE8A`. The next index obtained is `3Eh`, this is a negative index and this value needs to be padded with two zeros. This means that instead of the value:

$$\texttt{D555} = 1101\ 0101\ 0101\ 0101$$

we will get:

$$\texttt{5550} = 0101\ 0101\ 0101\ 0000$$

This padding is necessary to align the data correctly and ensure proper fragment extraction.

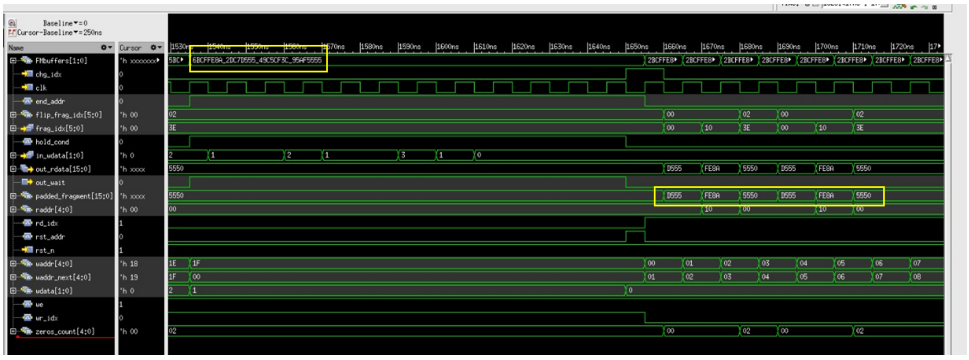


Figure 3.1: fragment memory waveform for test bench file

The following is the code we wrote that describes the hardware component.

proj_fm.sv

```
`timescale 1ns / 1ps

module proj_fm #(
    // Module parameters
    parameter BUFFER_COUNT = proj_pkg::FM_BUFFER_COUNT,
        ↪ // Number of buffers in the FM
    parameter RAMS = proj_pkg::FM_RAMS_COUNT,
        ↪ // Number of RAMs in each buffer
    parameter ENTRIES = proj_pkg::FM_ENTRIES_COUNT,
        ↪              // Number of entries in each RAM
    parameter OFFSET = proj_pkg::FM_OFFSET_COUNT,
        ↪ // Size of the offset in each entry
    parameter DATA_BITS = proj_pkg::FM_DATA_BITS,            //
        ↪ Width of each memory cell
    parameter INDICE_LEN = proj_pkg::INDICE_LEN,          //
        ↪ Length of the index
    parameter SIGNED_INDICE_LEN = proj_pkg::SIGNED_INDICE_LEN,
        ↪  // Length of signed index
    parameter FRAG_LEN = proj_pkg::FM_EXTENDER_FRAG_LEN_BITS
        ↪             // Length of the fragment - in bits!
) (
```

```
    // Module ports
    input  wire                          clk,    // Clock signal
    input  wire                          rst_n,  // Reset signal
        ↪ (active low)
    input  wire [DATA_BITS-1:0]          in_wdata,  // Input data
    input  wire                          chg_idx,   // Change
        ↪ index signal
    input  logic [SIGNED_INDICE_LEN-1:0] frag_idx, // Fragment
        ↪ index
    output wire [FRAG_LEN-1:0]           out_rdata, // Output
        ↪ data
    output wire                          out_wait // wait signal
        ↪ for the module before the ACMI
);

    // Local parameters
    localparam FM_BUFFER_SIZE = 32;  // RAMS * ENTRIES *
        ↪ OFFSET = 2 * 8 * 2
    localparam RAM_ADDR_BITS = 1;    // $clog2(RAMS) =
        ↪ $clog2(2)
    localparam ENTRIES_ADDR_BITS = 3; // $clog2(ENTRIES) =
        ↪ $clog2(8)
    localparam OFFSET_ADDR_BITS = 1;  // $clog2(OFFSET) =
        ↪ $clog2(2)
    localparam ADDR_BITS = 5;         // RAM_ADDR_BITS +
        ↪ ENTRIES_ADDR_BITS + OFFSET_ADDR_BITS

    // Internal signals
    logic clk, rst_n;
    logic [ADDR_BITS-1:0] waddr, waddr_next;
    logic [DATA_BITS-1:0] wdata;
    logic
        ↪ [BUFFER_COUNT-1:0][FM_BUFFER_SIZE-1:0][DATA_BITS-1:0]
        ↪ FMbuffers;
    logic end_addr, hold_cond, rst_addr;
    logic wr_idx, rd_idx;
    logic [FRAG_LEN-1:0] padded_fragment;
    logic [INDICE_LEN-1:0] raddr, zeros_count;
    logic [SIGNED_INDICE_LEN-1:0] flip_frag_idx;
    logic we;

    // Input assignments
    assign wdata = hold_cond ? FMbuffers[wr_idx][waddr] :
        ↪ in_wdata;

    // Output assignment
    assign out_rdata = padded_fragment;

    assign out_wait = hold_cond;

    // Address and control logic
    assign waddr_next = end_addr ? 1'b0 : waddr + 1'b1;
    assign end_addr = (waddr == (FM_BUFFER_SIZE-1)) ? 1'b1 :
        ↪ 1'b0;
```

```
    assign hold_cond = end_addr & ~chg_idx;
    assign rst_addr = end_addr & chg_idx;
    assign rd_idx = ~wr_idx;
    assign we = ~hold_cond || end_addr;

    // Fragment preparation logic
    always_comb begin
        padded_fragment = '0;
        zeros_count = '0;
        flip_frag_idx = '0;

        if (frag_idx[SIGNED_INDICE_LEN-1] == 1'b1) begin
            flip_frag_idx = (~frag_idx + 1'b1);
            zeros_count = flip_frag_idx[INDICE_LEN-1:0];
            raddr = '0;
        end else begin
            raddr = frag_idx[INDICE_LEN-1:0];
        end
    end

    genvar i;
    generate
        for (i = 0; i < 16; i++) begin : gen_padded_fragment
            ↪ // FRAG_LEN is 16
            always_comb begin
                if (i < (16 - zeros_count) && (raddr + i >= 0)
                    ↪ && (raddr + i < 32)) begin  //
                    ↪ FM_BUFFER_SIZE is 32
                    padded_fragment[i + (zeros_count << 1)] =
                        ↪ FMbuffers[rd_idx][raddr + (i >>
                        ↪ 1)][i[0]];
                end else begin
                    padded_fragment[i + (zeros_count << 1)] =
                        ↪ 1'b0;
                end
            end
        end
    endgenerate

    // Sequential logic
    always_ff @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            wr_idx <= '0;
            waddr <= '0;
        end else begin
            // Change index logic
            if (chg_idx) begin
                wr_idx <= rd_idx;
            end

            // Write address logic
            if (hold_cond) begin
                waddr <= waddr;
            end else if (rst_addr) begin
```

```
102                 waddr <= '0;
103             end else begin
104                 waddr <= waddr_next;
105             end
106 
107             // Write data to buffers
108             if (we) begin
109                 FMbuffers[wr_idx][waddr] <= wdata;
110             end
111         end
112     end
113 
114 endmodule
```

3.2 Hasher

3.2.1 Description

Hasher is a hardware component that implements a single algorithm. According to the definitions of the article, the implementation hash function is Murmur3.

MurmurHash, a non-cryptographic hash function, is distinguished by its expeditious and efficacious computation of hash values, which are instrumental for general lookup operations. This algorithm is characterized by its operational simplicity and rapid data processing capabilities, rendering it particularly suitable for hash-based structures such as hash tables, caches, and checksums. The nomenclature 'MurmurHash' is derived from the algorithm's fundamental operations—'multiply' and 'rotate'—which are pivotal to its functionality. The most recent iteration, MurmurHash3, introduces enhancements that include the provision of both 32-bit and 128-bit hash values, along with optimizations for various computing platforms, thereby affirming its adaptability and extensive applicability in various domains.

3.2.2 Parameters

Inner Parameter	Package Parameter	Value	Description
KMER_LEN	KMER_LEN	16	Length of k-mer in bases
DATA_BITS	BASE_LEN	2	Bits to represent a base
HASHER_DATA_BITS	HASHER_SORTER_SIGNATURE	32	Width of hasher signature
c1	-	32'hcc9e2d51	Constant
c2	-	32'h1b873593	Constant
m	-	32'd5	Constant
n	-	32'he6546b64	Constant

3.2.3 Interface

Interface	Signal	Direction	Width
System	clk	Input	1
System	rst_n	Input	1
System	seed	Input	HASHER_DATA_BITS
Kmer Buffer	kmer	Input	HASHER_DATA_BITS
Sorter	signature	Output	HASHER_DATA_BITS

Ports Description

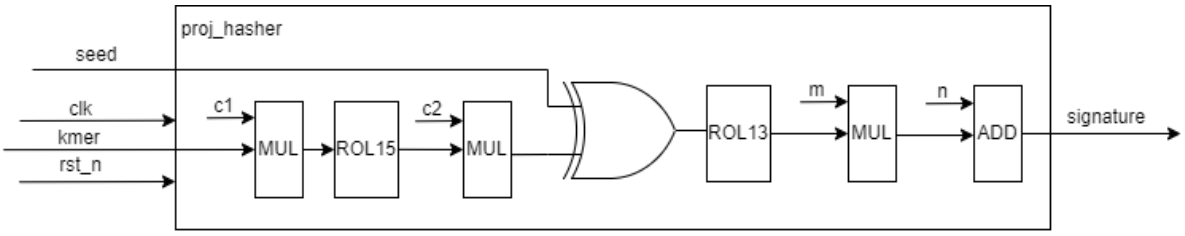
Input Ports:

- `clk`: Clock signal (unused in this implementation)
- `rst_n`: Active-low reset signal (unused in this implementation)
- `seed`: Initial hash value or key (32 bits)
- `kmer`: Input data to be hashed (32 bits)

Output Port:

- `signature`: Computed hash value (32 bits)

3.2.4 Block Diagram



3.2.5 Implementation

In each clock cycle, the hasher receives a single k-mer value represented by 4 bytes. Additionally, it maintains a fixed SEED value, also encoded as 4 bytes. Guided by the fixed constants, the hasher performs multiplications and rotations on the input k-mer. These operations transform the k-mer into a hash value. Finally, an approver issues a single signature based on the resulting hash value.

The structure of the component

The Hasher module is structured as a combinational logic block that performs the core operations of the MurmurHash3 algorithm: multiplication and rotation. It uses parameters from the `proj_pkg` package to define its input and output widths, making it flexible and configurable within the larger system.

Subcomponents

The main subcomponents are:

- Multiplication logic
- Rotation logic
- XOR gate
- Addition logic

Computational Logic

The Hasher module implements the MurmurHash3 algorithm through the following steps:

1. Kmer Processing:
    - Multiplies the for the input kmer with c1
    - Rotates the the former result by 15 bits
    - Multiplies the for the input kmer with c1
2. Seed Processing:
    - XORs the former result with the seed
3. Hash Computation:
    - Rotates the the former result by 15 bits
    - Multiplies the for the input kmer with m

- Adds n to the former result

The module uses predefined constants (`c1`, `c2`, `m`, `n`) that are integral to the MurmurHash3 algorithm, ensuring the generation of high-quality hash values.

3.2.6 Codes & Tests

We will demonstrate the functionality of the component by inserting a signature value. 0xAB1020C5 will be the value of the tested signature. The mathematical operation to be performed appears in the following formulas:

$$\begin{aligned} k &= c_2 \cdot \text{Roll}_{15}\left(c_1 \cdot 0x\text{AB1020C5}\right) \\ &= c_2 \cdot \text{Roll}_{15}\left(0x\text{2B72FF55}\right) \\ &= c_2 \cdot 0x\text{7FAA95B9} \\ &= 0x\text{1782463B} \end{aligned} \tag{3.1}$$

$$\begin{aligned} \text{signature} &= m \cdot \text{Roll}_{13}\left(0x\text{AC718ADD} \oplus k\right) + n \\ &= m \cdot \text{Roll}_{13}\left(0x\text{BBF3CCE6}\right) + n \\ &= m \cdot 0x\text{799CD77E} + n \\ &= 0x\text{4664A0DA} \end{aligned} \tag{3.2}$$

Let's look at the wave diagram that describes the operation of the component for preset values. Note that we get the desired value for the signature we tested.

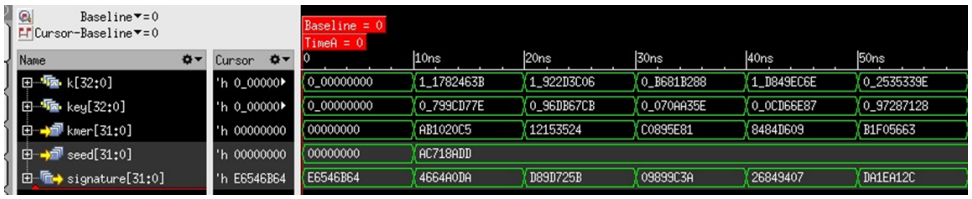


Figure 3.2: Hasher waveform for test bench file

The following is the code we wrote that describes the hardware component.

proj_hasher.sv

```
1  `timescale 1ns / 1ps
2
3  module proj_hasher
4    #(
5      parameter KMER_LEN = proj_pkg::KMER_LEN,
6      parameter DATA_BITS = proj_pkg::BASE_LEN,
7      parameter HASHER_DATA_BITS =
           ↪ proj_pkg::HASHER_SORTER_SIGNATURE
8    )(
9      input  wire                          clk,    // Clock signal
10     input  wire                          rst_n,  // Reset signal
           ↪ (active low)
11     input wire [HASHER_DATA_BITS-1:0]  seed,
```

```
12      input wire [HASHER_DATA_BITS-1:0]  kmer,
13      output logic [HASHER_DATA_BITS-1:0] signature
14    );
15
16    // Intermediate signals
17    logic [HASHER_DATA_BITS:0]        k, key;
18
19    localparam [HASHER_DATA_BITS-1:0]   c1 = 32'hcc9e2d51;
20    localparam [HASHER_DATA_BITS-1:0]   c2 = 32'h1b873593;
21    localparam [HASHER_DATA_BITS-1:0]   m = 32'd5;
22    localparam [HASHER_DATA_BITS-1:0]   n = 32'he6546b64;
23
24    // Hash computation
25    always_comb begin
26        k = kmer;
27        k = k * c1;
28        k = {k[HASHER_DATA_BITS-16:0],
              ↪ k[HASHER_DATA_BITS-1:HASHER_DATA_BITS-15]}; //
              ↪ ROL15
29          k = k * c2;
30          key = seed;
31        key = key ^ k;
32        key = {key[HASHER_DATA_BITS-14:0],
              ↪ key[HASHER_DATA_BITS-1:HASHER_DATA_BITS-13]}; //
              ↪ ROL13
33        signature = key * m + n;
34    end
35
36  endmodule
```

3.3 Sorter

3.3.1 Description

Sorter is a hardware component that implements a sorting mechanism for signature-index pairs. It is designed to maintain a sorted list of the smallest indices based on their corresponding signatures.

The Sorter module is characterized by its ability to continuously update and maintain a sorted list of signature-index pairs, making it particularly suitable for applications requiring real-time sorting of data streams. This component is essential in systems that need to identify and track the smallest elements in a continuous data flow.

The term 'Sorter' is derived from its primary function—to sort and maintain an ordered list of elements. This module introduces features that include configurable list size, signature and index widths, along with control inputs for resetting and ending the sorting process, thereby affirming its adaptability and applicability in various data processing scenarios.

3.3.2 Parameters

Inner Parameter	Package Parameter	Value	Description
INDICES_COUNT	SORTER_EXTENDER_INDICES_COUNT	256	Number of indices to maintain
INDICE_LEN	INDICE_LEN	15	Width of each index
SIGNATURE_LEN	HASHER_SORTER_SIGNATURE	32	Width of each signature
POSITION_LEN	SORTER_POSITION_LEN	8	Width of position value

3.3.3 Interface

Interface	Signal	Direction	Width
System	clk	Input	1
System	rst_n	Input	1
Input	in_signature	Input	SIGNATURE_LEN
Input	in_index	Input	INDICE_LEN
Control	end_sorting	Input	1
Output	out_smallest_idx	Output	INDICES_COUNT * INDICE_LEN
Output	sort_valid	Output	1

Ports Description

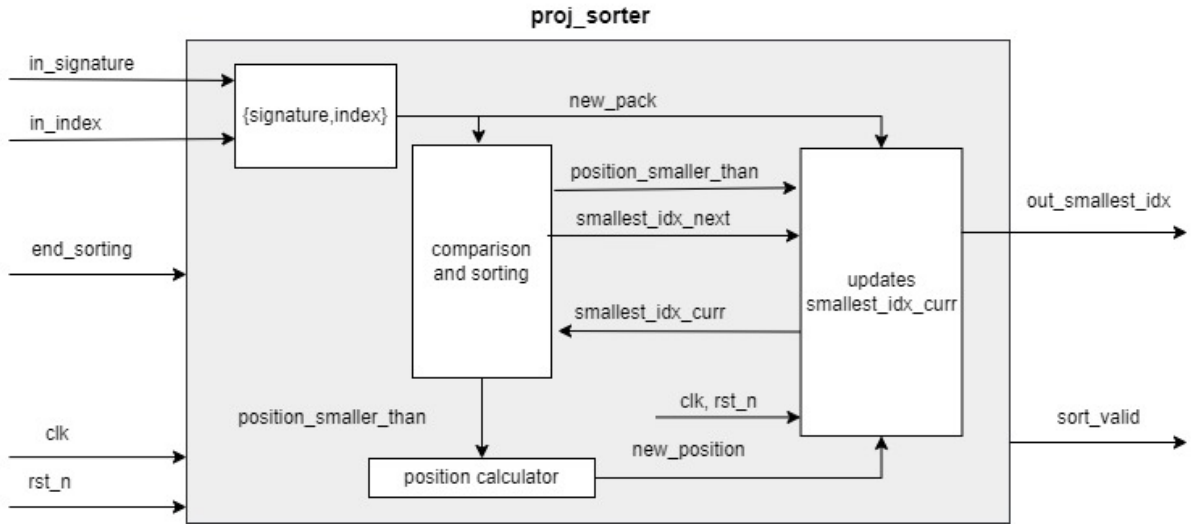
Input Ports:

- `clk`: Clock signal
- `rst_n`: Active-low reset signal
- `in_signature`: Input signature
- `in_index`: Input index
- `end_sorting`: Signal to end sorting process

Output Ports:

- `out_smallest_idx`: Sorted smallest indices
- `sort_valid`: Signal indicating valid sorted output of the smallest indices

3.3.4 Block Diagram



3.3.5 Implementation

In each clock cycle, the Sorter compares the incoming signature-index pair with the currently stored pairs. It determines the correct position for the new pair in the sorted list and updates the list accordingly. The module maintains a fixed-size list of the smallest signature-index pairs encountered.

The structure of the component

The Sorter module is structured as a sequential logic block that performs comparison and sorting operations. It uses parameters from the `proj_pkg` package to define its input and output widths, making it adaptable to different signature and index sizes within the larger system.

Subcomponents

The main subcomponents are:

- Comparison logic for determining the position of new elements
- Shift register logic for maintaining the sorted list
- Control logic for reset and end-of-sorting operations

Computational Logic

The `proj_sorter` module is responsible for implementing a real-time sorting mechanism for signature-index pairs. The main functionality is to maintain a sorted list of indices based on their associated signatures and continuously update this list as new data arrives. This section details the key elements of the implementation, including buffer management, sorting logic, and sequential updates.

1. Buffer Management:

The `proj_sorter` module keeps track of the smallest indices in two buffers: `smallest_idx_curr` and `smallest_idx_next`. These buffers store the current sorted list of signature-index pairs and the next sorted values during the update phase.

The module processes new input in the form of signature-index pairs, provided by the `in_signature` and `in_index` signals. The new pair is compared against the current sorted list, and its position in the sorted order is calculated based on the signatures of the existing entries.

2. Sorting Logic:

The sorting logic is responsible for comparing the new signature with the existing list of sorted indices. It computes the number of signatures smaller than the new input and determines the position where the new input should be inserted. The signal `count_signatures_smaller_than` indicates how many current entries have a smaller signature, and `position_smaller_than` calculates the precise insertion point.

The sorting logic iterates through the list, updating `smallest_idx_next` based on the comparison results:

3. Insertion Mechanism:

Once the insertion point is computed, the new signature and index pair is inserted into the sorted list. The logic in `smallest_idx_next` updates the sorted array, ensuring that all other elements retain their order. This process allows the module to dynamically handle incoming data without disturbing the sorted order of the existing entries.

If the calculated position for the new entry is valid (i.e., `position_smaller_than` is non-zero), the new entry is inserted into the appropriate position, and the remaining elements are shifted accordingly:

4. Sequential Logic:

The sorting and buffer management are synchronized with the clock signal. The sequential logic ensures that the sorted list is updated every clock cycle, and the input values are registered in `new_pack_r` for processing in the next cycle.

The module responds to both the clock (`clk`) and reset (`rst_n`) signals. Upon reset, the sorted list is initialized to default values (e.g., signatures set to the maximum possible value and indices to zero):

The module uses combinational logic to determine the new position and prepare the next state of the sorted list. Sequential logic is used to update the current state of the list on each clock cycle. The design allows for continuous sorting of incoming data while maintaining a fixed-size list of the smallest elements.

3.3.6 Codes & Tests

The example shown below is an example with reduced parameters (not the actual ACMI parameters), for demonstration purposes. We will demonstrate the component's functionality by a case study of a limited component with only 4 stored values at any given time.

In the figure 3.3, it can seen that there is a loading of new values into the table of saved values. It should be noted that when a new value arrives that is smaller than the values found in the table, the saved values will change their position according to the new value that arrived. In the following example, we refer to the situation given in the figure3.3 at the moment of receiving the signature along with the index 4 (surrounded by a light blue square). At the moment before receiving the new package we have four values that fill the table. The signature with the lowest value, index 5, is in the lowest position while the signature with the highest value, index 8, is in the highest position.All the signatures are compared with the new signature and since it is smaller than all of them, the signatures of indexes 5,6,7 move to a higher cell respectively, the signature with the index 8 is thrown from the table and the new signature is

written in the lowest place since it is smaller than all the existing signatures. It should be noted that the index list is sent only when the signal (end_sorting) that all signatures have arrived is received.

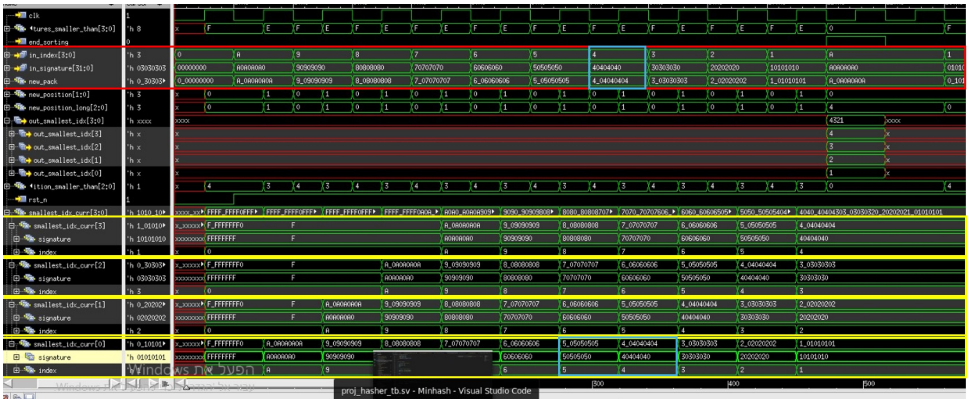


Figure 3.3: Smallest-4 signatures Sorter waveform for test bench file

The following is the code we wrote that describes the hardware component.

proj_sorter.sv

```
import proj_pkg::*;  // Include the project package

module proj_sorter #(
    // Module parameters, using values from the project package
    parameter INDICES_COUNT =
        ↪ proj_pkg::SORTER_EXTENDER_INDICES_COUNT,
    parameter INDICE_LEN = proj_pkg::INDICE_LEN,
    parameter SIGNATURE_LEN =
        ↪ proj_pkg::HASHER_SORTER_SIGNATURE,
    parameter POSITION_LEN = proj_pkg::SORTER_POSITION_LEN
)(
    // Module inputs and outputs
    input wire [SIGNATURE_LEN-1:0] in_signature,
    input wire [INDICE_LEN-1:0] in_index,
    output wire [INDICES_COUNT-1:0][INDICE_LEN-1:0]
        ↪ out_smallest_idx,
    input wire rst_n,
    input wire end_sorting,
    output wire sort_valid,
    input wire clk
);
    // Internal signals
    signature_index_pack [INDICES_COUNT-1:0] smallest_idx_next;
    signature_index_pack [INDICES_COUNT-1:0] smallest_idx_curr;
    signature_index_pack new_pack;
    logic [INDICES_COUNT-1:0] count_signatures_smaller_than;
    logic [POSITION_LEN:0] position_smaller_than;
    logic [POSITION_LEN:0] new_position_long;
    logic [POSITION_LEN-1:0] new_position;

    // Assign input values to internal signals
    assign new_pack.signature = in_signature;
```

```
30      assign new_pack.index = in_index;
31
32
33      always_comb begin
34          count_signatures_smaller_than = '0;
35          position_smaller_than = '0;
36          new_position = '0;
37          new_position_long = '0;
38          for (int i = 0; i < INDICES_COUNT; i= i+1) begin
39              count_signatures_smaller_than[i] =
                    ↪ (new_pack.signature <
                    ↪ smallest_idx_curr[i].signature) ? 1'b1 :
                    ↪ 1'b0;
40              position_smaller_than +=
                    ↪ count_signatures_smaller_than[i];
41              if (position_smaller_than == '0) begin
42                  smallest_idx_next[i].signature =
                        ↪ smallest_idx_curr[i].signature;
43                  smallest_idx_next[i].index =
                        ↪ smallest_idx_curr[i].index;
44              end else begin
45                  if (i < new_position) begin
46                      smallest_idx_next[i].signature =
                            ↪ smallest_idx_curr[i].signature;
47                      smallest_idx_next[i].index =
                            ↪ smallest_idx_curr[i].index;
48                  end else if (i > new_position) begin
49                      smallest_idx_next[i].signature =
                            ↪ smallest_idx_curr[i-1].signature;
50                      smallest_idx_next[i].index =
                            ↪ smallest_idx_curr[i-1].index;
51                  end
52              end
53          end
54          new_position_long = INDICES_COUNT -
                ↪ position_smaller_than;
55          new_position = new_position_long[POSITION_LEN-1:0];
56      end
57
58
59      // Assign sorted indices to output
60      generate
61          for (genvar j = 0; j < INDICES_COUNT; j++) begin
62              assign out_smallest_idx[j] = end_sorting ?
                    ↪ smallest_idx_curr[j].index : 'x;
63          end
64      endgenerate
65
66      assign sort_valid = end_sorting ? 1'b1 : 1'b0;
67
68      // Sequential logic for updating current smallest indices
69      always_ff @(posedge clk) begin
70          for (int i = 0; i < INDICES_COUNT; i++) begin
71              if (~rst_n) begin
```

```
72                  // Reset values
73                  smallest_idx_curr[i].signature <= '1;
74                  smallest_idx_curr[i].index <= '0;
75              end else begin
76                  // Update with new values
77                  if ((i == new_position) &
                        ↪ (position_smaller_than != '0)) begin
78                      smallest_idx_curr[i].signature <=
                            ↪ new_pack.signature;
79                      smallest_idx_curr[i].index <=
                            ↪ new_pack.index;
80                  end else begin
81                      smallest_idx_curr[i].signature <=
                            ↪ smallest_idx_next[i].signature;
82                      smallest_idx_curr[i].index <=
                            ↪ smallest_idx_next[i].index;
83                  end
84              end
85          end
86      end
87  endmodule
```

3.4 Extender

3.4.1 Description

The Extender is the final component of the ACMI (Accelerated Computational Microbiome Identification). It uses information from the Fragment Memory (FM) and performs expansion operations on the values that the Sorter found to be the smallest. The Extender extracts an 8-base wide DNA fragment for the relevant bases.

3.4.2 Parameters

Inner Parameter	Package Parameter	Value	Description
FRAG_LEN_BITS	FM_EXTENDER_FRAG_LEN_BITS	512	Length of fragment in bits
FRAG_SIZE	FRAG_LEN	256	Length of fragment in bases
KMER_SIZE	KMER_LEN	16	Length of k-mer in bases
INDICES_COUNT	SORTER_EXTENDER_INDICES_COUNT	256	Number of indices to process
INDICE_LEN	INDICE_LEN	15	Width of each index
SIGNED_INDICE_LEN	SIGNED_INDICE_LEN	16	Width of signed index
FRAG_PART_ONE_HOT	EXTENDER_OUT_PART_LEN_ONE_HOT	8	Length of one-hot encoded fragment part
BASE_LEN	BASE_LEN	2	Number of bits per base
ONE_HOT_LEN	ONE_HOT_LEN	4	Length of one-hot encoding per base
FRAG_PART	EXTENDER_OUT_PART_LEN	4	Length of fragment part

3.4.3 Interface

Interface	Signal	Direction	Width
System	clk	Input	1
System	rst_n	Input	1
Input	in_fragment	Input	FRAG_LEN_BITS
Input	in_kmer_indices	Input	INDICES_COUNT * INDICE_LEN
Control	valid_indices	Input	1
Output	out_index	Output	SIGNED_INDICE_LEN
Output	out_gfm	Output	FRAG_PART_ONE_HOT

Ports Description

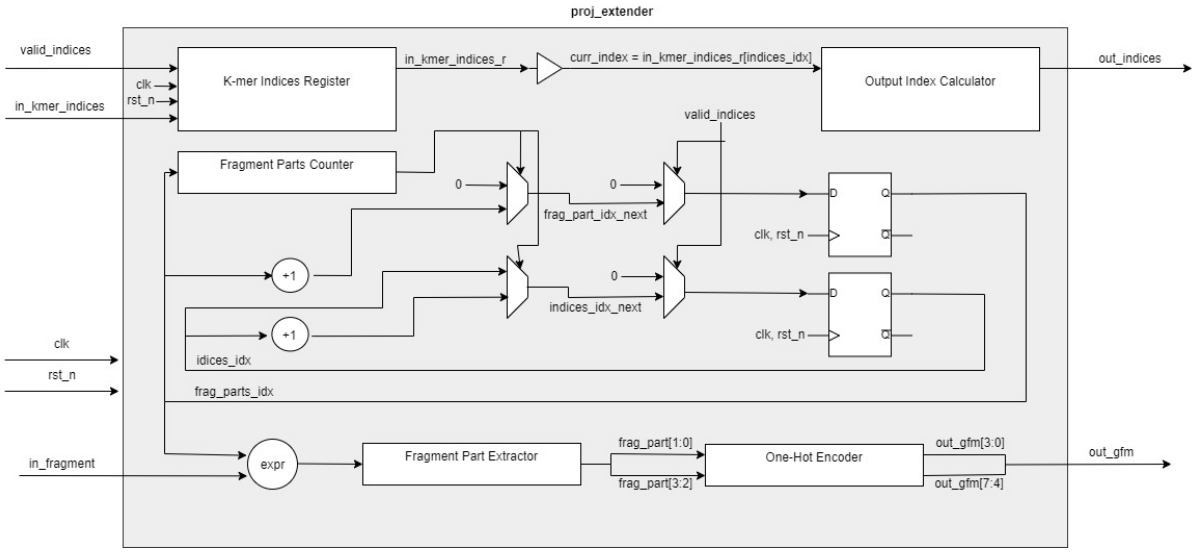
Input Ports:

- `clk`: Clock signal
- `rst_n`: Reset signal (active low)
- `in_fragment`: Input fragment
- `in_kmer_indices`: Input k-mer indices
- `valid_indices`: Signal indicating valid input indices

Output Ports:

- `out_index`: Output index
- `out_gfm`: Output one-hot encoded fragment

3.4.4 Block Diagram



3.4.5 Implementation

The ‘proj_extender‘ module utilizing parameters from the project package. The key implementation details include:

- **Fragment Extraction:** The module extracts the relevant fragment part based on the fragment index. The extraction logic uses the ‘frag_parts_idx‘ signal to identify the current part of the fragment to be processed.
- **One-Hot Encoding:** The extracted fragment part is encoded into a one-hot format using a generate block, where each base of the DNA fragment (2 bits) is converted into a one-hot code (4 bits). The one-hot encoding allows efficient representation of the DNA bases for further processing.
- **Index Calculation:** The current index is selected from the k-mer indices array (‘in_kmer_indices‘) based on the current index position (‘indices_idx‘). The output index is calculated by adjusting the current index to account for the difference between the fragment size and k-mer size.
- **Sequential Logic:** The module uses sequential logic to update the fragment parts index and indices index on each clock cycle. It also stores the input k-mer indices into a register (‘in_kmer_indices_r‘) when valid indices are provided.

The overall architecture is designed to handle the extraction and encoding of DNA fragments efficiently, allowing for parallel processing of multiple indices and fragment parts.

3.4.6 Codes & Tests

The example shown below is an example with reduced parameters (not the actual ACMI parameters), for demonstration purposes. In the wave diagram, you can see that information encoded over 16 bits is received from the fragment memory. The information obtained is:

$$0FFD = 0000111111111101$$

It can be observed that the output is received as requested:

$$D = 11,01 \rightarrow \text{gfm}(D) = 1000,0010 = 82$$
$$F = 11,11 \rightarrow \text{gfm}(F) = 1000,1000 = 88$$
$$0 = 00,00 \rightarrow \text{gfm}(0) = 0001,0001 = 11$$

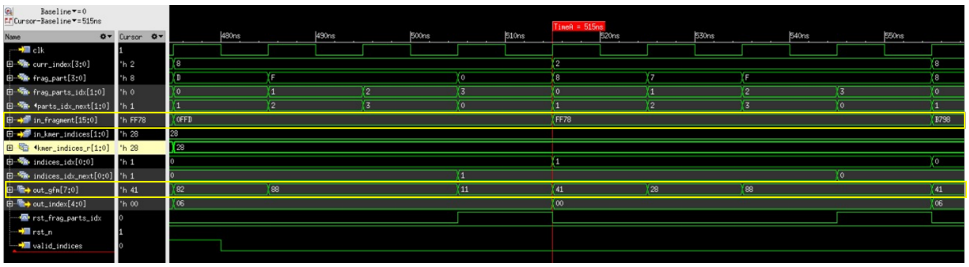


Figure 3.4: 16 bit fragment - Extender waveform for test bench file

The following is the code we wrote that describes the hardware component.

proj_extender.sv

```
// Timescale directive for simulation
`timescale 1ns / 1ps
// Import the project package
import proj_pkg::*;
// Module declaration with parameters
module proj_extender #(
    parameter FRAG_LEN_BITS =
        ↪ proj_pkg::FM_EXTENDER_FRAG_LEN_BITS,
    parameter FRAG_SIZE = proj_pkg::FRAG_LEN,
    parameter KMER_SIZE = proj_pkg::KMER_LEN,
    parameter INDICES_COUNT =
        ↪ proj_pkg::SORTER_EXTENDER_INDICES_COUNT,
    parameter INDICE_LEN = proj_pkg::INDICE_LEN,
    parameter SIGNED_INDICE_LEN = proj_pkg::SIGNED_INDICE_LEN,
    parameter FRAG_PART_ONE_HOT =
        ↪ proj_pkg::EXTENDER_OUT_PART_LEN_ONE_HOT,
    parameter BASE_LEN = proj_pkg::BASE_LEN,
    parameter ONE_HOT_LEN = proj_pkg::ONE_HOT_LEN,
    parameter FRAG_PART = proj_pkg::EXTENDER_OUT_PART_LEN
) (
    // Input ports
    input logic [FRAG_LEN_BITS-1:0] in_fragment,
    input logic [INDICES_COUNT-1:0][INDICE_LEN-1:0]
        ↪ in_kmer_indices,
    input wire valid_indices,
    input wire rst_n,
    input wire clk,
    // Output ports
    output logic [SIGNED_INDICE_LEN-1:0] out_index,
    output logic [FRAG_PART_ONE_HOT-1:0] out_gfm
);
    // Local parameters
    localparam FRAG_PARTS_COUNT = (FRAG_LEN_BITS >>
```

```
        ↪ $clog2(FRAG_PART));
30      localparam FRAG_PARTS_COUNT_BITS =
        ↪ $clog2(FRAG_PARTS_COUNT);
31      localparam INDICES_COUNT_BITS = $clog2(INDICES_COUNT);
32      // Internal signals
33      logic [FRAG_PARTS_COUNT_BITS-1:0] frag_parts_idx;
34      logic [FRAG_PARTS_COUNT_BITS-1:0] frag_parts_idx_next;
35      logic rst_frag_parts_idx;
36      logic [INDICE_LEN-1:0] curr_index;
37      logic [INDICES_COUNT_BITS-1:0] indices_idx;
38      logic [INDICES_COUNT_BITS-1:0] indices_idx_next;
39      logic [FRAG_PART-1:0] frag_part;
40      logic [INDICES_COUNT-1:0][INDICE_LEN-1:0]
        ↪ in_kmer_indices_r;
41
42      // Combinational logic
43      // Reset fragment parts index when it reaches the maximum
44      assign rst_frag_parts_idx = (frag_parts_idx ==
        ↪ (FRAG_PARTS_COUNT - 1)) ? 1'b1 : 1'b0;
45      // Calculate next fragment parts index
46      assign frag_parts_idx_next = rst_frag_parts_idx ? 1'b0 :
        ↪ frag_parts_idx + 1'b1;
47      // Select current index from input indices
48      assign curr_index = in_kmer_indices_r[indices_idx];
49      // Extract fragment part for output
50      assign frag_part = in_fragment[FRAG_PART*frag_parts_idx +:
        ↪ FRAG_PART];
51
52      generate
53          for (genvar i = 0; i < FRAG_PART >> $clog2(BASE_LEN);
            ↪ i++) begin : gen_gfm
54              always_comb begin
55                  case (frag_part[i*BASE_LEN +: BASE_LEN])
56                      2'b00: out_gfm[i*ONE_HOT_LEN +:
                        ↪ ONE_HOT_LEN] = 4'b0001;
57                      2'b01: out_gfm[i*ONE_HOT_LEN +:
                        ↪ ONE_HOT_LEN] = 4'b0010;
58                      2'b10: out_gfm[i*ONE_HOT_LEN +:
                        ↪ ONE_HOT_LEN] = 4'b0100;
59                      2'b11: out_gfm[i*ONE_HOT_LEN +:
                        ↪ ONE_HOT_LEN] = 4'b1000;
60                      default: out_gfm[i*ONE_HOT_LEN +:
                        ↪ ONE_HOT_LEN] = 4'b0000;
61                  endcase
62              end
63          end
64      endgenerate
65      // Calculate next indices index
66      assign indices_idx_next = rst_frag_parts_idx ? indices_idx
        ↪ + 1'b1 : indices_idx;
67      // Calculate output index
68      assign out_index = {1'b0, curr_index} -
        ↪ SIGNED_INDICE_LEN'(((FRAG_SIZE - KMER_SIZE) >> 1));
69      // Sequential logic for fragment parts index
```

```
70      always_ff @(posedge clk or negedge rst_n) begin :
            ↪ parts_index
71          if (~rst_n) begin
72              frag_parts_idx <= '0;
73          end else if (valid_indices) begin
74              frag_parts_idx <= '0;
75          end else begin
76              frag_parts_idx <= frag_parts_idx_next;
77          end
78      end
79
80      // Sequential logic for indices sample
81      always_ff @(posedge clk or negedge rst_n) begin :
            ↪ index_sample
82          if (valid_indices) begin
83              in_kmer_indices_r <= in_kmer_indices;
84          end
85      end
86
87      // Sequential logic for indices index
88      always_ff @(posedge clk or negedge rst_n) begin :
            ↪ indices_index
89          if (~rst_n) begin
90              indices_idx <= '0;
91          end else if (valid_indices) begin
92              indices_idx <= '0;
93          end else begin
94              indices_idx <= indices_idx_next;
95          end
96      end
97  endmodule
```

3.5 Index Counter

3.5.1 Description

Index Counter is a hardware component that implements a simple counting mechanism. It is designed to generate sequential indices and signal when a specific count has been reached.

The Counter module is characterized by its ability to generate sequential indices up to a predefined maximum value, making it particularly suitable for addressing or sequencing operations in digital systems. This component is essential in applications requiring ordered progression through a series of states or memory locations.

The nomenclature 'Counter' is derived from its primary function—to count or increment a value sequentially. This module introduces features that include the provision of configurable bit-width indices and a finished count signal, along with control inputs for starting and resetting the count, thereby affirming its adaptability and applicability in various digital design scenarios.

3.5.2 Parameters

Inner Parameter	Package Parameter	Value	Description
INDICE_LEN	INDICE_LEN	15	Width of the index output

3.5.3 Interface

Interface	Signal	Direction	Width
System	clk	Input	1
System	rst_n	Input	1
Control	start	Input	1
Output	index	Output	INDICE_LEN
Output	finished_count	Output	1

Ports Description

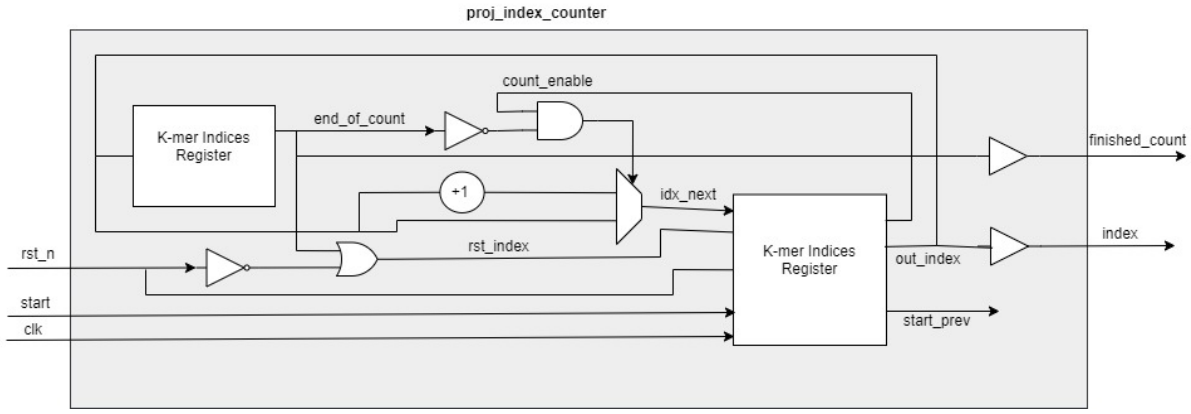
Input Ports:

- `clk`: Clock signal
- `rst_n`: Reset signal (active low)
- `start`: Signal to start counting

Output Ports:

- `index`: Current count value
- `finished_count`: Signal indicating count completion

3.5.4 Block Diagram



3.5.5 Implementation

The ‘proj_index_counter‘ module is implemented as a sequential logic block that performs counting operations. It uses the project package (‘proj_pkg‘) to configure the index width, enabling it to count up to a value defined by ‘FM_BUFFER_SIZE‘. The module is responsible for generating sequential indices and signaling when the count is complete.

Operation

The counter begins counting when the ‘start‘ signal is asserted. It increments the internal index value on each clock cycle, controlled by the ‘count_enabled‘ flag. The module stops counting when the index reaches the maximum value (‘FM_BUFFER_SIZE - 1‘) and asserts the ‘finished_count‘ signal. If the counter is reset via the ‘rst_n‘ signal or when the end of the count is reached, the internal index is cleared, and counting can restart.

Control Logic

- **Start Detection:** The counter detects the rising edge of the ‘start‘ signal to begin counting. It uses a previous state register (‘start_prev‘) to ensure counting starts only on a transition from low to high.
- **Count Enable:** The ‘count_enabled‘ flag controls whether the index increments. Counting continues until the end of the count is detected, at which point the counter stops.
- **Reset Behavior:** The index is reset either when the end of the count is reached or when the system receives a reset signal (‘rst_n‘).

Sequential Logic

The core of the module is a sequential logic block that operates on the rising edge of the clock (‘clk‘). The block increments the index when counting is enabled and resets the index to zero on reset or when the maximum count is reached. It also manages the ‘start_prev‘ signal to detect rising edges of the ‘start‘ signal.

End-of-Count Detection

The module includes a comparator to check if the index has reached its maximum value (‘FM_BUFFER_SIZE - 1‘). When this condition is met, the ‘finished_count‘ signal is asserted, indicating that the counting operation is complete.

Output Generation

The module provides the current index value on the ‘index‘ output port, and the ‘finished_count‘ signal is used to indicate when the counting process is done. The index can be used as an address or state counter in various digital systems, while the ‘finished_count‘ flag can trigger subsequent operations in the system.

3.5.6 Codes & Tests

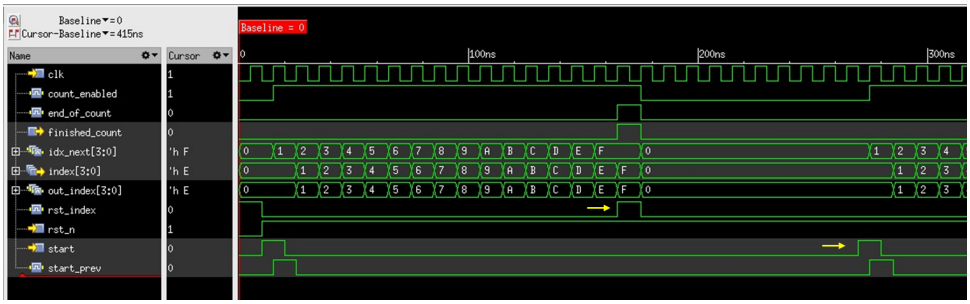


Figure 3.5: A counter of 16 indexes waveform for test bench file

The example shown below is an example with reduced parameters (not the actual ACMI parameters), for demonstration purposes. In the example, a counter of 16 index units is shown. In the wave diagram, we will notice that when we receive a rising start signal for a single clock cycle or when there is a reset signal, the count starts again. In addition, when these condition are not met and the count finished, the counter stopped counting as expected.

proj_index_counter.sv

```
import proj_pkg::*;  // Include the package

module proj_counter
#(
    parameter INDICE_LEN = proj_pkg::INDICE_LEN
)
(
    output wire [INDICE_LEN-1:0] index,
    output logic finished_count,
    input wire clk,
    input wire rst_n,
    input wire start
);
    // Internal signals
    logic [INDICE_LEN-1:0] out_index;
    logic end_of_count;
    logic [INDICE_LEN-1:0] idx_next;
```

```
18      logic rst_index;
19      logic count_enabled;
20      logic start_prev;
21
22      // Check if write address reached the end of the buffer
23      assign end_of_count = (out_index == (FM_BUFFER_SIZE-1)) ?
            ↪ 1'b1 : 1'b0;
24      assign finished_count = end_of_count;
25
26      // Increment write address or keep current value
27      assign idx_next = (count_enabled & ~end_of_count) ?
            ↪ out_index + 1'b1 : out_index;
28
29      // Reset index when end of count is reached or on reset
30      assign rst_index = end_of_count | (~rst_n);
31
32      // Assign output index
33      assign index = out_index;
34
35      // Sequential logic for updating the index and detecting
            ↪ start signal
36      always_ff @(posedge clk or negedge rst_n) begin
37          if (~rst_n) begin
38              out_index <= '0;
39              count_enabled <= 1'b0;
40              start_prev <= 1'b0;
41          end else begin
42              start_prev <= start;
43              if (rst_index) begin
44                  out_index <= '0;
45                  count_enabled <= 1'b0;
46              end else if (start & ~start_prev) begin
47                  // Rising edge of start signal
48                  count_enabled <= 1'b1;
49                  out_index <= '0;
50              end else if (count_enabled) begin
51                  out_index <= idx_next;
52              end
53          end
54      end
55  endmodule
```

3.6 Kmer Buffer

3.6.1 Description

Kmer Buffer is a hardware component that implements a shifting buffer mechanism for storing and managing k-mers. It is designed to sequentially store incoming nucleotide data and output a complete k-mer when the buffer is full.

The Kmer Buffer module is characterized by its ability to store and shift nucleotide data, making it particularly suitable for applications in DNA sequence analysis and bioinformatics hardware accelerators. This component is essential in systems that need to process DNA sequences in k-mer units.

The term 'Kmer Buffer' is derived from its primary function—to buffer and prepare k-mers. This module introduces features that include configurable k-mer length and nucleotide data width, along with control inputs for resetting the buffer and a full signal output, thereby affirming its adaptability and applicability in various genomic data processing scenarios.

3.6.2 Parameters

Inner Parameter	Package Parameter	Value	Description
DATA_BITS	KMER_BUFFER_BITS	2	Number of bits for each nucleotide
KMER_LEN	KMER_BUFFER_LEN	16	Length of the k-mer
OUT_KMER	-	32	Total bits in the output k-mer

3.6.3 Interface

Interface	Signal	Direction	Width
System	clk	Input	1
System	rst_n	Input	1
Input	in_data	Input	DATA_BITS
Control	start_over	Input	1
Output	out_kmer	Output	KMER_LEN * DATA_BITS
Output	full	Output	1

Ports Description

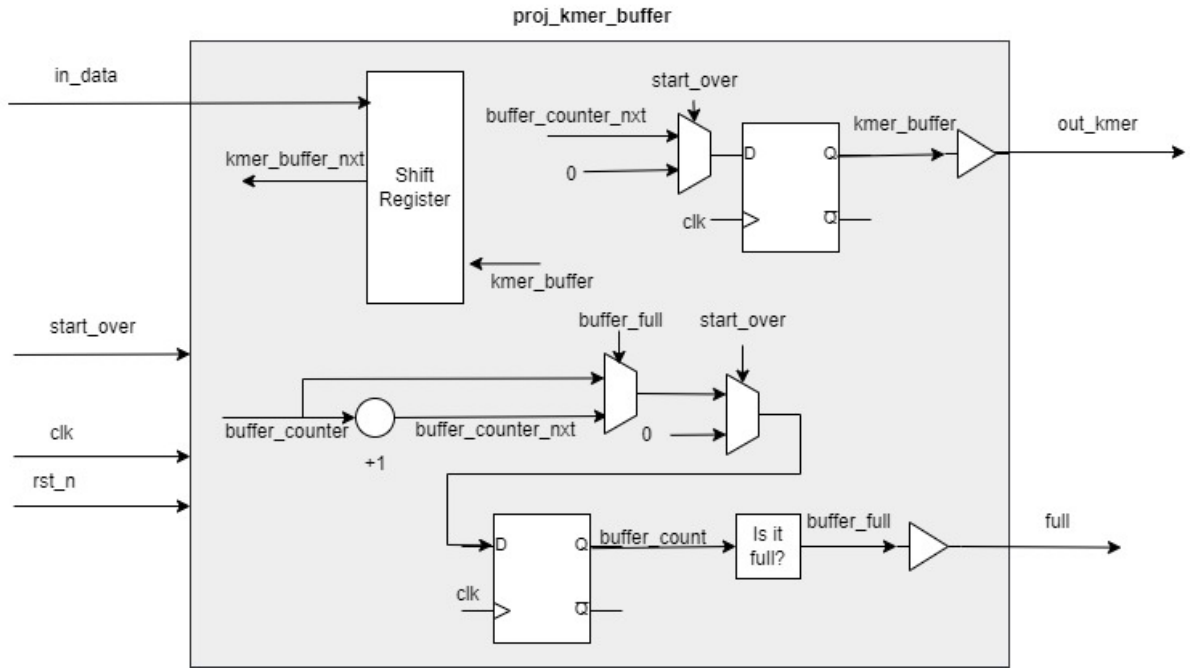
Input Ports:

- `clk`: Clock signal
- `rst_n`: Reset signal (active low)
- `in_data`: Input nucleotide data
- `start_over`: Signal to reset the buffer

Output Ports:

- `out_kmer`: Output k-mer
- `full`: Indicates when the buffer is full

3.6.4 Block Diagram



3.6.5 Implementation

In each clock cycle, the Kmer Buffer shifts in new nucleotide data and updates its internal buffer. It maintains a count of stored nucleotides and signals when the buffer is full (i.e., a complete k-mer is ready). The buffer can be reset either by the active-low reset signal or the start_over input.

The structure of the component

The Kmer Buffer module is structured as a sequential logic block that performs shifting and storing operations. It uses parameters from the `proj_pkg` package to define its input and output widths, making it adaptable to different k-mer lengths and nucleotide representations within the larger system.

Subcomponents

The main subcomponents are:

- Shift register logic for the k-mer buffer
- Counter logic for tracking the number of stored nucleotides
- Control logic for buffer reset and full detection

Computational Logic

The Kmer Buffer module implements its functionality through the following steps:

1. Input Processing:
    - Shifts in new nucleotide data at each clock cycle

- Updates the internal k-mer buffer with the new data

2. Buffer Management:
   - Maintains a count of stored nucleotides
   - Signals when the buffer is full (i.e., a complete k-mer is ready)
3. Output Generation:
   - Continuously outputs the current state of the k-mer buffer
   - Asserts the full signal when the buffer contains a complete k-mer
4. Reset Handling:
   - Resets the buffer and count on either system reset or start_over signal

The module uses generate blocks to create the shifting logic, allowing for easy adaptation to different k-mer lengths. The buffer_full signal is used to prevent overwriting of a complete k-mer until it is read out or intentionally reset.

3.6.6 Codes & Tests

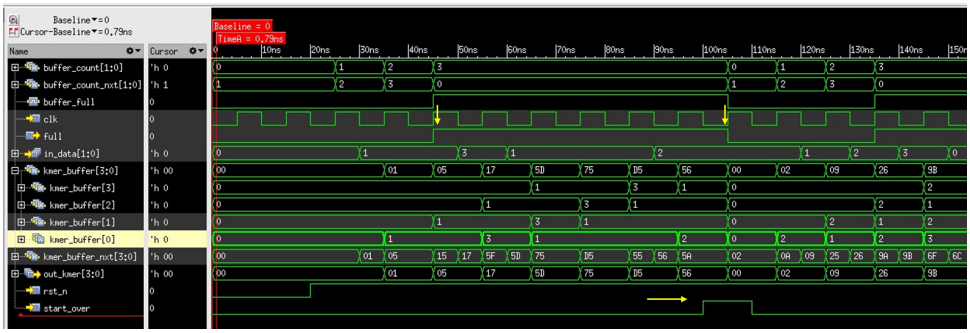


Figure 3.6: 4-mer buffer waveform for test bench file

The example shown below is an example with reduced parameters (not the actual ACMI parameters), for demonstration purposes. The attached example shows how the information is saved in the buffer. The example shows a buffer with a depth of 4 that receives values and saves them as requested. The component functions like a queue, The component functions like a queue, it stores the values 1,1,3,1,1,1,2 in the order of their arrival. Note that when a start_over signal is received, all the stored values are reset and the component will return to receive the values 2,1,2,3 with the next clock rise.

The following is the code we wrote that describes the hardware component.

kmer_buffer.sv

```
1  `timescale 1ns / 1ps
2  import proj_pkg::*;  // Include the project package
3
4  module proj_kmer_buffer #(
5      parameter DATA_BITS = proj_pkg::KMER_BUFFER_BITS,     //
           ↪ Number of bits for each nucleotide
6      parameter KMER_LEN = proj_pkg::KMER_BUFFER_LEN,     //
           ↪ Length of the k-mer
```

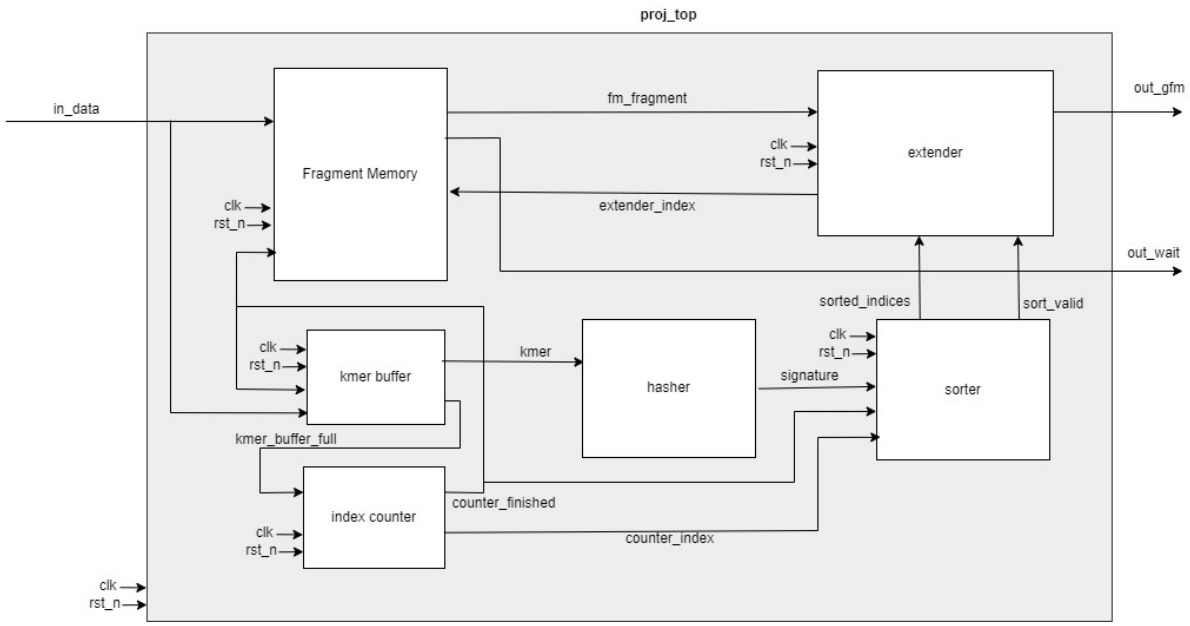
```
7     parameter OUT_KMER = KMER_LEN * DATA_BITS  // Total bits
          ↪ in the output k-mer
8 )(
9     input wire clk,              // Clock input
10    input wire rst_n,            // Active-low reset
11    input wire [DATA_BITS-1:0] in_data,  // Input nucleotide
          ↪ data
12    output wire [KMER_LEN-1:0][DATA_BITS-1:0] out_kmer,  //
          ↪ Output k-mer
13    input  wire start_over,      // Signal to reset the buffer
14    output logic full            // Indicates when the buffer
          ↪ is full
15 );
16    // Internal signals
17    logic [KMER_LEN*DATA_BITS-1:0] kmer_buffer;     // Current
          ↪ k-mer buffer
18    logic [KMER_LEN*DATA_BITS-1:0] kmer_buffer_nxt; // Next
          ↪ state of k-mer buffer
19    logic [$clog2(KMER_LEN)-1:0] buffer_count;           //
          ↪ Current count of nucleotides in buffer
20    logic [$clog2(KMER_LEN)-1:0] buffer_count_nxt;       //
          ↪ Next state of buffer count
21
22    // Increment buffer count
23    assign buffer_count_nxt = buffer_count + 1'b1;
24
25    // Determine if buffer is full
26    assign buffer_full = (buffer_count == KMER_LEN - 1) ? 1'b1
          ↪ : 1'b0;
27    assign full = buffer_full;
28
29    // Generate next state of k-mer buffer
30    generate
31        for (genvar i = 2; i < KMER_LEN; i++) begin
32          assign kmer_buffer_nxt[0] = in_data[0];
33          assign kmer_buffer_nxt[1] = in_data[1];
34          assign kmer_buffer_nxt[i] = kmer_buffer[i-1];  //
                ↪ Shift existing data
35        end
36    endgenerate
37
38    // Connect internal buffer to output
39    assign out_kmer = kmer_buffer;
40
41    // Update k-mer buffer on clock edge
42    always_ff @(posedge clk or negedge rst_n) begin
43        if (~rst_n) begin
44          kmer_buffer <= '0;  // Reset buffer on active-low
                ↪ reset
45        end else if (start_over) begin
46          kmer_buffer <= '0;  // Reset buffer on start_over
                ↪ signal
47        end else begin
48          kmer_buffer <= kmer_buffer_nxt;  // Update buffer
```

```
                ↪ with next state
49          end
50      end
51
52      // Update buffer count on clock edge
53      always_ff @(posedge clk or negedge rst_n) begin
54          if (~rst_n) begin
55            buffer_count <= '0;  // Reset count on active-low
                ↪ reset
56          end else if (start_over) begin
57            buffer_count <= '0;  // Reset count on start_over
                ↪ signal
58          end else if (buffer_full) begin
59            buffer_count <= buffer_count;  // Maintain count
                ↪ when buffer is full
60          end else begin
61            buffer_count <= buffer_count_nxt;  // Increment count
62          end
63      end
64  endmodule
```

Chapter 4

Integration

This chapter focuses on the integration of the various components of the ACMI hardware accelerator for the Vision Transformer algorithm, with specific emphasis on the implementation of MinHash for genome analysis. The successful integration of these components ensures the overall system performs the required computations efficiently and accurately.



4.1 Overview of Integration

The integration process involves connecting multiple modules, each of which plays a key role in the overall functionality of ACMI. These components, including the Fragment Memory, Kmer Buffer, Index counter, Hasher, Sorter, and Extender, must work together seamlessly to achieve high-performance genome fragment processing.

4.2 Integration Challenges

During the integration process, several challenges were encountered, including:

- Synchronization Issues: Ensuring that all components operate in sync was critical, particularly with the clock cycle timing between the Fragment Memory and the Hasher.
- Memory Access Conflicts: Managing access to shared memory resources without causing delays or data corruption.

4.3 codes & test

Figure 4.1: button-4 signatures ACMI test banch

The example shown below is an example with reduced parameters (not the actual ACMI parameters), for demonstration purposes. It can be seen that in every clock cycle the counter_index increases by 1, the kmer is manipulated and a signature is obtained that the relationship between it and the kmer is not visible to the eye. When sort_valid is received we see what the four minimum signatures are and they are passed to the extender. In the next clock cycle, we see the output 24 which corresponds to the gfm of 6 obtained from the fragment memory. In the next clock cycle we get 11 as expected for the value 0. In the next clock cycle we get 24 again because the value is 6 and finally we get 42 according to the value 9.
In total, we get the functionality expected from the component. It is important to note that for this simulation we used different sizes from the component sizes to allow easy observation of the results. The following is the code we wrote that describes the hardware component.

top.sv

```
`timescale 1ns / 1ps
import proj_pkg::*;
module proj_top (
    input wire clk,
    input wire rst_n,
    input wire [proj_pkg::BASE_LEN-1:0] in_data,
    output wire [proj_pkg::FRAG_LEN-1:0] out_fragment,
    output wire out_wait
);
    // Internal signals
    wire [proj_pkg::INDICE_LEN-1:0] counter_index;
    wire counter_finished;
    wire [proj_pkg::KMER_LEN*proj_pkg::BASE_LEN-1:0] kmer;
```

```
    wire kmer_buffer_full;
    wire [proj_pkg::HASHER_SORTER_SIGNATURE-1:0] signature;
    wire [proj_pkg::SORTER_EXTENDER_INDICES_COUNT-1:0]
[proj_pkg::INDICE_LEN-1:0] sorted_indices;
    wire sort_valid;
    wire [proj_pkg::SIGNED_INDICE_LEN-1:0] extender_index;
    wire [proj_pkg::FM_EXTENDER_FRAG_LEN_BITS-1:0] fm_fragment;
    // FM module
    proj_fm #(
        .BUFFER_COUNT(proj_pkg::FM_BUFFER_COUNT),
        .RAMS(proj_pkg::FM_RAMS_COUNT),
        .ENTRIES(proj_pkg::FM_ENTRIES_COUNT),
        .OFFSET(proj_pkg::FM_OFFSET_COUNT),
        .DATA_BITS(proj_pkg::FM_DATA_BITS),
        .INDICE_LEN(proj_pkg::INDICE_LEN),
        .SIGNED_INDICE_LEN(proj_pkg::SIGNED_INDICE_LEN),
        .FRAG_LEN(proj_pkg::FM_EXTENDER_FRAG_LEN_BITS)
    ) u_fm (
        .clk(clk),
        .rst_n(rst_n),
        .in_wdata(in_data),
        .chg_idx(counter_finished),
        .frag_idx(extender_index),
        .out_rdata(fm_fragment),
        .out_wait(out_wait)
    );
    // Counter module
    proj_counter#(
        .INDICE_LEN(proj_pkg::INDICE_LEN)
    ) u_counter
        (
        .clk(clk),
        .rst_n(rst_n),
        .start(kmer_buffer_full),
        .index(counter_index),
        .finished_count(counter_finished)
    );
    // Kmer Buffer module
    proj_kmer_buffer #(
        .DATA_BITS(proj_pkg::KMER_BUFFER_BITS),
        .KMER_LEN(proj_pkg::KMER_BUFFER_LEN),
        .OUT_KMER(proj_pkg::KMER_BUFFER_LEN *
            ↪ proj_pkg::KMER_BUFFER_BITS)
    ) u_kmer_buffer (
        .clk(clk),
        .rst_n(rst_n),
        .in_data(in_data),
        .out_kmer(kmer),
        .start_over(counter_finished),
        .full(kmer_buffer_full)
    );
    // Hasher module
    proj_hasher #(
        .KMER_LEN(proj_pkg::KMER_LEN),
```

```
        .DATA_BITS(proj_pkg::BASE_LEN),
        .HASHER_DATA_BITS(proj_pkg::HASHER_SORTER_SIGNATURE)
    ) u_hasher (
        .clk(clk),
        .rst_n(rst_n),
        .seed(32'hac718add),
        .kmer(kmer),
        .signature(signature)
    );
    // Sorter module
    proj_sorter #(
        .INDICES_COUNT(proj_pkg::SORTER_EXTENDER_INDICES_COUNT),
        .INDICE_LEN(proj_pkg::INDICE_LEN),
        .SIGNATURE_LEN(proj_pkg::HASHER_SORTER_SIGNATURE),
        .POSITION_LEN(proj_pkg::SORTER_POSITION_LEN)
    ) u_sorter (
        .clk(clk),
        .rst_n(rst_n),
        .in_signature(signature),
        .in_index(counter_index),
        .out_smallest_idx(sorted_indices),
        .end_sorting(counter_finished),
        .sort_valid(sort_valid)
    );
    // Extender module
    proj_extender #(
        .FRAG_LEN_BITS(proj_pkg::FM_EXTENDER_FRAG_LEN_BITS),
        .FRAG_SIZE(proj_pkg::FRAG_LEN),
        .KMER_SIZE(proj_pkg::KMER_LEN),
        .INDICES_COUNT(proj_pkg::SORTER_EXTENDER_INDICES_COUNT),
        .INDICE_LEN(proj_pkg::INDICE_LEN),
        .SIGNED_INDICE_LEN(proj_pkg::SIGNED_INDICE_LEN),
        .FRAG_PART_ONE_HOT(proj_pkg::EXTENDER_OUT_PART_LEN_ONE_HOT),
        .BASE_LEN(proj_pkg::BASE_LEN),
        .ONE_HOT_LEN(proj_pkg::ONE_HOT_LEN),
        .FRAG_PART(proj_pkg::EXTENDER_OUT_PART_LEN)
    ) u_extender(
        .clk(clk),
        .rst_n(rst_n),
        .in_fragment(fm_fragment),
        .in_kmer_indices(sorted_indices),
        .valid_indices(sort_valid),
        .out_index(extender_index),
        .out_gfm(out_fragment)
    );
endmodule
```

Chapter 5

Results

5.1 Synthesis Process Overview

The synthesis flow in this project uses Genus, a tool from Cadence, for logic synthesis. The process involves several stages including reading libraries, reading RTL, elaboration, constraint application, and actual synthesis.

5.2 Tools and Environment

Cadence Genus serves as the primary synthesis tool. The environment setup involves sourcing various configuration files:

- Project-specific definitions from `../proj.defines`
- Technology-specific library paths and definitions from files such as `../libraries.$TECHNOLOGY.tcl`
- Standard cell, SRAM, and potentially I/O technology files

5.3 Synthesis Constraints

Constraints are applied through SDC (Synopsys Design Constraints) files:

- The main constraints file is read using `read_sdc $design(functional_sdc)`
- Clock gating constraints are set:
- Minimum flops for clock gating: 8
- Clock gating style: latch

5.4 Area Analysis

The area analysis provides insight into the physical footprint of the synthesized design. The total area of the component is 176,241.14 μm^2, as calculated by the synthesis tool in the area analysis report. This total is composed of contributions from various cell types, including sequential cells, inverters, buffers, and logic cells. The area reports highlight the contributions of different standard cell types and libraries used in the design.

Instance	Module	Cell Count	Cell Area	Net Area	Total Area
proj_top		39711	111609.080	64632.062	176241.142
u_fm	proj_fm_BUFFER_COUNT2_RAMS8_ENTRIES128_OFFEST32_DA	24000	71428.320	44425.886	115854.206
u_extender	proj_extender_FRAG_LEN_BITS512_FRAG_SIZE256_KMER_S	9073	27833.184	13237.722	41070.906
u_hasher	proj_hasher_KMER_LEN16_DATA_BITS2_HASHER_DATA_BITS	1261	4487.760	1996.886	6484.646
u_sorter	proj_sorter_INDICES_COUNT256_INDICE_LEN15_SIGNATUR	485	2338.272	881.362	3219.634
u_kmer_buffer	proj_kmer_buffer_DATA_BITS2_KMER_LEN16_OUT_KMER32	51	237.960	46.271	284.231
u_counter	proj_counter_INDICE_LEN15	36	152.640	56.982	209.622

Figure 5.1: Area Analysis

5.4.1 Library Breakdown

The design utilizes several sub-modules, each contributing to the total area. The breakdown is as follows:

- `u_fm`: 24,000 instances, contributing 115,854.21 μm^2 of area.
- `u_extender`: 9,073 instances, contributing 41,070.91 μm^2 of area.
- `u_hasher`: 1,261 instances, contributing 6,484.65 μm^2 of area.
- `u_sorter`: 485 instances, contributing 3,219.63 μm^2 of area.
- `u_kmer_buffer`: 51 instances, contributing 284.23 μm^2 of area.
- `u_counter`: 36 instances, contributing 209.62 μm^2 of area.

5.4.2 Cell Type Breakdown

Different types of cells contribute variably to the total area:

- **Sequential cells** (such as flip-flops and latches) account for 3,779 instances, taking up 25,640.36 μm^2 of area, or 28.6% of the total.
- **Inverters** contribute significantly, with 4,435 instances occupying 6,857.28 μm^2 of area, 5.5% of the total area.
- **Logic cells** dominate the area utilization with 31,280 instances, contributing to 81,587.52 μm^2 of area, which is 65.5% of the total area.
- **Buffers** and **clock-gating cells** have minimal impact on the overall area, with buffers contributing 483.48 μm^2 (0.4%) and clock-gating cells only 19.44 μm^2.

5.5 Timing Analysis

The timing analysis reports provide detailed information about the critical paths in the design. These reports show the propagation of signals through various cells, including arrival times, transition delays, and cell types.

5.5.1 Critical Path Analysis

From the provided timing reports, we can observe several critical paths:

- The path starting from `in_data[1]` has a total arrival time of 2786 ps, with the signal propagating through multiple cells including `A021_X3M_A9TL`, `INV_X3B_A9TL`, and `INV_X2M_A9TL`, as it can be seen in figure 5.2.
- Another critical path begins at `u_fm_waddr_reg[6]/CK` and ends at `out_wait`, with a total arrival time of 1432 ps. This path includes cells like `DFFRPQN_X2M_A9TL`, `INV_X6M_A9TL`, and `AND4_X0P5M_A9TL`, as it can be seen in figure 5.3.
- A more complex path is shown in the third image, starting from `u_kmer_buffer/kmer_buffer_reg[14]/` and propagating through multiple `u_hasher` components. This path has a significantly longer arrival time of 8780 ps, as it can be seen in figure 5.4.

Timing Point	Flags	Arc	Edge	Cell	Fanout	Trans (ps)	Delay (ps)	Arrival (ps)
in_data[1]	-	-	R	(arrival)	2	54	0	2016
g187504/Y	-	A0->Y	R	A021_X3M_A9TL	15	219	208	2225
drc_bufs193659/Y	-	A->Y	F	INV_X3B_A9TL	16	320	269	2494
drc_bufs193557/Y	-	A->Y	R	INV_X2M_A9TL	16	295	292	2786
u_fm_FMbuffers_reg[1][504][1]/D	-	-	R	EDFFQ_X1M_A9TL	16	-	0	2786

Figure 5.2: Timing Analysis: in2reg

Timing Point	Flags	Arc	Edge	Cell	Fanout	Trans (ps)	Delay (ps)	Arrival (ps)
u_fm_waddr_reg[6]/CK	-	-	R	(arrival)	9	0	0	0
u_fm_waddr_reg[6]/QN	-	CK->QN	F	DFFRPQN_X2M_A9TL	4	160	318	318
g192255/Y	-	A->Y	R	INV_X6M_A9TL	14	123	122	440
g317400/Y	-	C->Y	R	AND4_X0P5M_A9TL	1	123	201	641
g317384/Y	-	D->Y	R	AND4_X0P5M_A9TL	1	140	218	859
g317374/Y	-	A->Y	F	NAND3_X2M_A9TL	9	258	193	1052
g317368/Y	-	AN->Y	F	NAND2B_X2M_A9TL	6	179	262	1315
g125451/Y	-	A->Y	R	INV_X1M_A9TL	1	109	118	1432
out_wait	-	-	R	(port)	-	-	0	1432

Figure 5.3: Timing Analysis: reg2out

Timing Point	Flags	Arc	Edge	Cell	Fanout	Trans (ps)	Delay (ps)	Arrival (ps)
u_kmer_buffer/kmer_buffer_reg[14]/CK	-	-	R	(arrival)	3434	0	0	0
u_kmer_buffer/kmer_buffer_reg[14]/Q	-	CK->Q	R	DFFRPQ_X2M_A9TL	8	208	373	373
u_hasher/const_mul_27_13_g7523/Y	-	A->Y	F	INV_X4M_A9TL	5	84	90	463
u_hasher/const_mul_27_13_g7499__7482/Y	-	A->Y	R	NOR2_X2A_A9TL	2	145	115	578
u_hasher/const_mul_27_13_g7566__4733/Y	-	B0->Y	R	AO21_X1M_A9TL	2	137	180	758
u_hasher/const_mul_27_13_g7361__7482/Y	-	A->Y	R	XOR2_X1M_A9TL	1	242	144	902
u_hasher/const_mul_27_13_g7261__5122/S	-	CI->S	F	ADDF_X1P4M_A9TL	2	132	368	1270
u_hasher/const_mul_27_13_g7258/Y	-	A->Y	R	INV_X3M_A9TL	1	68	74	1344
u_hasher/const_mul_27_13_g7222__8428/Y	-	A->Y	R	XOR2_X3M_A9TL	3	201	96	1440
u_hasher/const_mul_27_13_g7183__1666/Y	-	AN->Y	R	NOR2B_X4M_A9TL	2	93	165	1604
u_hasher/const_mul_27_13_g7167__8246/Y	-	A->Y	F	NOR2_X3M_A9TL	2	73	74	1679
u_hasher/const_mul_27_13_g7133__6131/Y	-	B->Y	R	NAND2_X3M_A9TL	2	94	88	1767
u_hasher/const_mul_27_13_g7099__6161/Y	-	B->Y	F	NOR2_X3B_A9TL	1	89	92	1859
u_hasher/const_mul_27_13_g7080__8428/Y	-	A->Y	F	OR4_X6M_A9TL	4	71	136	1995
u_hasher/const_mul_27_13_g2__2802/Y	-	A1->Y	F	AO21_X2M_A9TL	2	72	169	2164
u_hasher/const_mul_27_13_g7053__3680/Y	-	B1->Y	F	AO1B2_X2M_A9TL	1	93	100	2304
u_hasher/const_mul_27_13_g7044__6417/Y	-	S0->Y	R	MXT2_X3M_A9TL	11	216	244	2547
u_hasher/const_mul_29_12_g9423/Y	-	A->Y	F	INV_X2M_A9TL	9	151	153	2700
u_hasher/const_mul_29_12_g9263/Y	-	A1->Y	R	OAI211_X1M_A9TL	1	287	243	2943
u_hasher/const_mul_29_12_g9185__5526/S	-	A->S	F	ADDF_X1M_A9TL	1	108	368	3311
u_hasher/const_mul_29_12_g9134__7098/S	-	CI->S	R	ADDF_X1P4M_A9TR	2	129	249	3761
u_hasher/const_mul_29_12_g9125__8428/Y	-	B->Y	R	OR2_X0P5M_A9TL	3	240	233	3994
u_hasher/const_mul_29_12_g9106__1881/Y	-	B->Y	F	NAND2_X1A_A9TL	2	145	154	4148
u_hasher/const_mul_29_12_g9082__9315/Y	-	A1->Y	F	OA211_X0P7M_A9TL	3	146	271	4418
u_hasher/const_mul_29_12_g9019__6161/Y	-	A0->Y	F	OA211_X0P7M_A9TL	1	85	226	4644
u_hasher/const_mul_29_12_g9010__1705/Y	-	B0->Y	F	OA21_X1M_A9TL	5	152	195	4840
u_hasher/const_mul_29_12_g8976__3680/Y	-	B->Y	R	NOR2_X1P4A_A9TL	2	166	152	4991
u_hasher/const_mul_29_12_g8971/Y	-	A->Y	F	INV_X1M_A9TL	1	73	81	5072
u_hasher/const_mul_29_12_g8968__2398/Y	-	A->Y	R	NAND2_X1B_A9TL	3	171	113	5185
u_hasher/const_mul_29_12_g8957__7482/Y	-	A1N->Y	R	OAI2XB1_X0P5M_A9TL	1	265	167	5452
u_hasher/const_mul_29_12_g8944__5526/Y	-	B->Y	R	XOR2_X0P7M_A9TL	1	249	216	5669
u_hasher/g83/Y	-	A->Y	F	INV_X1M_A9TL	3	135	146	5815
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1309__2802/Y	-	B->Y	R	NOR2_X1B_A9TL	3	274	214	6029
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1287__1881/Y	-	B0->Y	R	AO21_X1M_A9TL	2	109	207	6236
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1251__2802/Y	-	B->Y	R	OR2_X0P5M_A9TL	3	221	216	6452
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1189__6783/Y	-	A1N->Y	R	OAI2XB1_X1M_A9TL	2	233	190	6722
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1172__4733/Y	-	A1->Y	F	AOI22_X1M_A9TL	1	148	150	6872
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1148__2346/Y	-	B0->Y	F	OA211_X0P7M_A9TL	4	182	254	7126
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1144__6161/Y	-	A->Y	R	NOR3_X1M_A9TL	2	313	246	7372
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1138__7098/Y	-	A1->Y	R	AO21B_X1M_A9TL	5	228	215	7667
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1129__8428/Y	-	A1->Y	R	AO21_X1M_A9TL	4	184	148	7915
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1122__7410/Y	-	B->Y	F	NAND2_X1M_A9TL	1	109	105	8020
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1104__3680/Y	-	A1->Y	F	OA211_X0P7M_A9TL	2	127	246	8166
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1313__8246/Y	-	A1->Y	R	OAI21_X1M_A9TL	2	248	206	8304
u_hasher/WALLACE_CSA_DUMMY_OP_groupi_g1097__6260/Y	-	A1N->Y	R	OAI2XB1_X1M_A9TL	1	256	294	8598
g191607/Y	-	B->Y	F	OR2_X0P5M_A9TL	1	90	182	8780
u_sorter_new_pack_r_reg[signature][31]/B	-	-	F	A2DFFQ_X3M_A9TL	1	-	0	8780

Figure 5.4: Timing Analysis: reg2reg

The frequency for a critical path can be calculated using the formula:

$$\text{Frequency} = \frac{1}{\text{Critical Path Delay}}$$

where the delay is converted from picoseconds (ps) to seconds (s).

Path from `u_kmer_buffer/kmer_buffer_reg[14]/CK`:

$$\text{Critical Path Delay} = 8780\,\text{ps} = 8780 \times 10^{-12}\,\text{s}$$

$$\text{Frequency} = \frac{1}{8780 \times 10^{-12}} \approx 113.9\,\text{MHz}$$

5.5.2 Cell Types and Delays

The timing reports reveal the use of various standard cell types:

- Flip-flops: `DFFRPQN_X2M_A9TL`
- Logic gates: `INV` (inverters), `AND4`, `NAND2B`, `NOR2`, `XOR2`
- Complex gates: `OAI21`, `AOI21`, `MXT2`

Transition delays vary significantly between cells, ranging from 68 ps to 320 ps, depending on the cell type and fanout.

Clock-to-Q delays are visible in the reports, particularly for flip-flops:

- The `DFFRPQN_X2M_A9TL` cell shows a clock-to-Q delay of 318 ps.
- For the `u_kmer_buffer` component, there's a notable clock-to-Q delay of 373 ps.

5.5.3 Fanout Analysis

Fanout values are reported for each cell in the critical paths:

- Most cells have relatively low fanout (1-4), which is good for maintaining signal integrity.
- Some cells, like `MXT2_X3M_A9TL`, have higher fanout (11), which may contribute to increased delays.

5.5.4 Conclusion

The timing analysis reveals complex path structures with varying delays across different components of the design. The longest path observed has an arrival time of 8780 ps, which could be a potential bottleneck for overall system performance. Further optimization may be necessary, focusing on cells with high fanout or long propagation delays to improve the overall timing of the design.

5.6 Power Estimation

The power estimation report generated by the Joules engine provides detailed information on the leakage, internal, and switching power of different components within the design. The total power consumption is broken down by different categories, as shown in the table below.

5.6.1 Component Power Breakdown

The total power consumption of the design is approximately 8.99 mW. The breakdown of power consumption by category is as follows:

- **Registers:** Registers account for 34.60% of the total power, consuming 3.11 mW. This includes 20.18 nW in leakage power, 2.51 mW in internal power, and 0.58 mW due to switching.
- **Logic:** Logic gates dominate the power consumption, contributing 65.32% of the total power at 5.87 mW. This includes 54.47 nW in leakage power, 2.00 mW in internal power, and 3.81 mW due to switching.
- **Latches:** Latches have a minimal impact on the overall power consumption, contributing only 0.06% or 5.06 µW.
- **Clock:** The clock network consumes only 2.07 µW, which accounts for 0.02% of the total power. This includes minimal leakage and internal power.

From the investigation of the report it appears that the power for the memory does not exist, but this result was obtained because the memory was defined in the FM component as a register. In order to get a stand on the power distribution, we will extract from the power report of the FM component its total power and estimate that it is the total power for memory and the remainder is the total power for the registers. This assessment is based on the fact that the absolute majority of the memory in the overall system is in the FM component and also that the logic in it can be balanced in its power consumption for other memories in the system. The `FM` component requires a total power of approximately 3.21 mW, which we will assume represents the total power consumption of the memory in the system.

With the assumption that the `FM` component power represents the memory's power consumption, we can now distribute the remaining power (8.99 mW - 3.21 mW = 5.77 mW) among the registers and logic. The revised power breakdown is as follows:

- **Memory (in `FM` component):** Consumes 3.21 mW, or 35.76% of the total system power.
- **Registers (outside `FM` component):** The remaining registers now consume 1.81 mW, accounting for 20.20% of the total system power.
- **Logic:** Logic gates outside the `FM` component consume 3.96 mW, which accounts for 44.04% of the total system power.

5.6.2 Power Distribution

The power report shows that dynamic power (comprising internal and switching power) accounts for the majority of the total power consumption:

- **Internal power** represents 50.25% of the total power, mainly driven by register and logic activities.
- **Switching power** represents 48.92%, indicating that a significant amount of power is consumed by signal transitions in the logic and registers.
- **Leakage power** accounts for only 0.83% of the total power, which is typical for designs using modern low-power technologies.

The table provides a comprehensive overview of the power distribution, with the majority of power being consumed by the logic cells due to high switching activity.

5.7 Optimization Results

The synthesis process includes several optimization stages:

1. Generic synthesis (`syn_generic`)
2. Technology mapping (`syn_map`)
3. Post-synthesis optimization (`syn_opt`)

We did both physical-aware and non-physical-aware synthesis, controlled by the `phys_synth_type` variable. Physical-aware synthesis can potentially improve the results by considering the placement information during optimization. We set optimization efforts to "low" for all stages:

```
set_db syn_generic_effort low
set_db syn_map_effort low
set_db syn_opt_effort low
```

Chapter 6

Summary & Discussion

In this chapter, we present a comprehensive summary of the key findings from our project, followed by an in-depth discussion of their implications within the broader context of existing literature and future research directions.

Our study has yielded several significant outcomes, chief among them being the successful implementation of the ACMI component. This novel hardware solution has demonstrated a marked improvement in efficiency when integrated within the VIRAL algorithm, a cutting-edge approach in machine learning. The results of our extensive testing and analysis indicate that ACMI provides both high accuracy and impressive speed in genome sequencing tasks, particularly when compared to current state-of-the-art methods such as UShER and Kraken2.

We have meticulously documented the functionality of each sub-component within ACMI, as well as the performance of the integrated system as a whole. The design was proposed and implemented using the SystemVerilog hardware description language, a choice that allowed for efficient hardware modeling and verification. We have included the associated code and test files in our documentation, providing a thorough discussion of the results obtained from each stage of the development process.

6.1 Discussion of Results

Our findings strongly suggest that the ACMI component effectively addresses the significant challenges posed by large-scale genome data processing, a critical issue in modern genomics research. The integration of ACMI with the Vision Transformer algorithm has yielded substantial enhancements in both the speed and accuracy of lineage assignment. This improvement is particularly evident in the detailed power and timing analyses we conducted, which revealed impressive performance metrics.

When compared to existing tools like UShER and Kraken2, the ACMI component demonstrated superior performance, especially at lower genome coverage levels. This characteristic makes ACMI a particularly promising tool for future genomics research, as it could potentially enable more efficient analysis of partial or low-quality genomic data, a common challenge in real-world applications.

To further validate our design, we performed a comprehensive synthesis of the ACMI component. This process allowed us to extract crucial information about its size, determine its potential operating frequency, and estimate its power consumption under various conditions. The synthesis results aligned remarkably well with our initial expectations, confirming the effectiveness of the various optimizations we applied during the design and implementation phases.

Our analysis revealed that the ACMI component is not only resource-efficient but also capable of operating at high frequencies. This combination of efficiency and performance is crucial for real-world applications in genomics, where large datasets need to be processed quickly and accurately. The viability of our hardware design was thus confirmed, paving the way for potential future implementations in specialized genomic analysis hardware.

6.2 Conclusion

In conclusion, the successful implementation of the ACMI component represents a significant leap forward in the field of computational genomics. The results of our comprehensive study clearly demonstrate its potential for substantially improving both the efficiency and accuracy of genome sequencing processes. As genomic data continues to grow in volume and complexity, tools like ACMI will become increasingly vital in enabling researchers to extract meaningful insights from this wealth of information.

By providing a detailed showcase of the functionality of each sub-component and the ACMI system as a whole, we have offered a thorough understanding of the intricacies involved in this hardware implementation. The synthesis results, which closely matched our expectations in terms of size, operating frequency, and power consumption, further underscore the success of this project and validate our approach to hardware-accelerated genomics.

This work lays a solid foundation for further exploration into the realm of hardware-accelerated genomics. It opens up new avenues for high-performance computing in biology, potentially revolutionizing how we approach complex genomic analyses. Future research could focus on further optimizing the ACMI design, exploring its application in other areas of computational biology, or investigating its potential integration with other cutting-edge algorithms in machine learning and artificial intelligence.

Chapter 7

Appendices

Top

top.sv

```
`timescale 1ns / 1ps
import proj_pkg::*;
module proj_top (
    input wire clk,
    input wire rst_n,
    input wire [proj_pkg::BASE_LEN-1:0] in_data,
    output wire [proj_pkg::FRAG_LEN-1:0] out_fragment,
    output wire out_wait
);
    // Internal signals
    wire [proj_pkg::INDICE_LEN-1:0] counter_index;
    wire counter_finished;
    wire [proj_pkg::KMER_LEN*proj_pkg::BASE_LEN-1:0] kmer;
    wire kmer_buffer_full;
    wire [proj_pkg::HASHER_SORTER_SIGNATURE-1:0] signature;
    wire [proj_pkg::SORTER_EXTENDER_INDICES_COUNT-1:0]
[proj_pkg::INDICE_LEN-1:0] sorted_indices;
    wire sort_valid;
    wire [proj_pkg::SIGNED_INDICE_LEN-1:0] extender_index;
    wire [proj_pkg::FM_EXTENDER_FRAG_LEN_BITS-1:0] fm_fragment;
    // FM module
    proj_fm #(
        .BUFFER_COUNT(proj_pkg::FM_BUFFER_COUNT),
        .RAMS(proj_pkg::FM_RAMS_COUNT),
        .ENTRIES(proj_pkg::FM_ENTRIES_COUNT),
        .OFFSET(proj_pkg::FM_OFFSET_COUNT),
        .DATA_BITS(proj_pkg::FM_DATA_BITS),
        .INDICE_LEN(proj_pkg::INDICE_LEN),
        .SIGNED_INDICE_LEN(proj_pkg::SIGNED_INDICE_LEN),
        .FRAG_LEN(proj_pkg::FM_EXTENDER_FRAG_LEN_BITS)
    ) u_fm (
        .clk(clk),
        .rst_n(rst_n),
        .in_wdata(in_data),
        .chg_idx(counter_finished),
        .frag_idx(extender_index),
```

```
37          .out_rdata(fm_fragment),
38          .out_wait(out_wait)
39      );
40      // Counter module
41      proj_counter#(
42          .INDICE_LEN(proj_pkg::INDICE_LEN)
43      ) u_counter
44          (
45          .clk(clk),
46          .rst_n(rst_n),
47          .start(kmer_buffer_full),
48          .index(counter_index),
49          .finished_count(counter_finished)
50      );
51      // Kmer Buffer module
52      proj_kmer_buffer #(
53          .DATA_BITS(proj_pkg::KMER_BUFFER_BITS),
54          .KMER_LEN(proj_pkg::KMER_BUFFER_LEN),
55          .OUT_KMER(proj_pkg::KMER_BUFFER_LEN *
                ↪ proj_pkg::KMER_BUFFER_BITS)
56      ) u_kmer_buffer (
57          .clk(clk),
58          .rst_n(rst_n),
59          .in_data(in_data),
60          .out_kmer(kmer),
61          .start_over(counter_finished),
62          .full(kmer_buffer_full)
63      );
64      // Hasher module
65      proj_hasher #(
66          .KMER_LEN(proj_pkg::KMER_LEN),
67          .DATA_BITS(proj_pkg::BASE_LEN),
68          .HASHER_DATA_BITS(proj_pkg::HASHER_SORTER_SIGNATURE)
69      ) u_hasher (
70          .clk(clk),
71          .rst_n(rst_n),
72          .seed(32'hac718add),
73          .kmer(kmer),
74          .signature(signature)
75      );
76      // Sorter module
77      proj_sorter #(
78          .INDICES_COUNT(proj_pkg::SORTER_EXTENDER_INDICES_COUNT),
79          .INDICE_LEN(proj_pkg::INDICE_LEN),
80          .SIGNATURE_LEN(proj_pkg::HASHER_SORTER_SIGNATURE),
81          .POSITION_LEN(proj_pkg::SORTER_POSITION_LEN)
82      ) u_sorter (
83          .clk(clk),
84          .rst_n(rst_n),
85          .in_signature(signature),
86          .in_index(counter_index),
87          .out_smallest_idx(sorted_indices),
88          .end_sorting(counter_finished),
89          .sort_valid(sort_valid)
```

```
    );
    // Extender module
    proj_extender #(
        .FRAG_LEN_BITS(proj_pkg::FM_EXTENDER_FRAG_LEN_BITS),
        .FRAG_SIZE(proj_pkg::FRAG_LEN),
        .KMER_SIZE(proj_pkg::KMER_LEN),
        .INDICES_COUNT(proj_pkg::SORTER_EXTENDER_INDICES_COUNT),
        .INDICE_LEN(proj_pkg::INDICE_LEN),
        .SIGNED_INDICE_LEN(proj_pkg::SIGNED_INDICE_LEN),
        .FRAG_PART_ONE_HOT(proj_pkg::EXTENDER_OUT_PART_LEN_ONE_HOT),
        .BASE_LEN(proj_pkg::BASE_LEN),
        .ONE_HOT_LEN(proj_pkg::ONE_HOT_LEN),
        .FRAG_PART(proj_pkg::EXTENDER_OUT_PART_LEN)
    ) u_extender(
        .clk(clk),
        .rst_n(rst_n),
        .in_fragment(fm_fragment),
        .in_kmer_indices(sorted_indices),
        .valid_indices(sort_valid),
        .out_index(extender_index),
        .out_gfm(out_fragment)
    );
endmodule
```

proj_pkg.sv

```
// Filename: proj_pkg.svh

// Define the package
package proj_pkg;

    // Generic parameters
    // Base length
    parameter BASE_LEN = 2;
    // Kmer length in Bases
    parameter KMER_LEN = 16;
    // Fragment length in bases
    parameter FRAG_LEN = 256;
    // single one hot encoding base len
    parameter ONE_HOT_LEN = 4;

    // FM parameters
    parameter FM_DATA_BITS = BASE_LEN;
    // Number of buffers in the FM
    parameter FM_BUFFER_COUNT = 2;
    // Number of RAMs in each buffer
    parameter FM_RAMS_COUNT = 8;
    // Number of entries in each RAM
    parameter FM_ENTRIES_COUNT = 128;
    // Size of the offset in each entry
    parameter FM_OFFSET_COUNT = 32;
    // Size of each buffer
    parameter FM_BUFFER_SIZE = FM_RAMS_COUNT *
```

```
        ↪ FM_ENTRIES_COUNT * FM_OFFSET_COUNT;
    // Size of the  FM - Extender fragment, in bits
    parameter FM_EXTENDER_FRAG_LEN_BITS = BASE_LEN * FRAG_LEN;
    // Length of a certain index
    parameter INDICE_LEN = $clog2(FM_BUFFER_SIZE);
    // Length of a certain signed index
    parameter SIGNED_INDICE_LEN = INDICE_LEN+1;

    // Kmer Buffer parametres
    parameter KMER_BUFFER_BITS = BASE_LEN;
    parameter KMER_BUFFER_LEN = KMER_LEN;

    // Sorter and Extender parameters
    parameter SORTER_EXTENDER_INDICES_COUNT = 256;

    // Sorter and Hasher parameters
    parameter HASHER_SORTER_SIGNATURE = 32;

    // Sorter parameters and structures
    parameter SORTER_POSITION_LEN =
        ↪ $clog2(SORTER_EXTENDER_INDICES_COUNT);
    parameter POSITION_LEN = SORTER_POSITION_LEN;
    typedef struct packed {
    logic [HASHER_SORTER_SIGNATURE-1:0] signature;
    logic [INDICE_LEN-1:0]  index;
    } signature_index_pack;

    // Extender parameters
    parameter EXTENDER_OUT_PART_COUNT = 2;
    parameter EXTENDER_OUT_PART_LEN = EXTENDER_OUT_PART_COUNT
        ↪ * BASE_LEN;
    parameter EXTENDER_OUT_PART_LEN_ONE_HOT =
        ↪ EXTENDER_OUT_PART_COUNT * ONE_HOT_LEN;
endpackage
```

proj_top_tb.sv

```
`timescale 1ns / 1ps
import proj_pkg::*;

module proj_top_tb;

    // Testbench signals
    reg clk;
    reg rst_n;
    reg [proj_pkg::BASE_LEN-1:0] in_data;
    wire [proj_pkg::EXTENDER_OUT_PART_LEN_ONE_HOT-1:0]
        ↪ out_fragment;
    wire out_wait;

    // Instantiate the DUT (Device Under Test)
    proj_top dut (
        .clk(clk),
```

```
        .rst_n(rst_n),
        .in_data(in_data),
        .out_fragment(out_fragment),
        .out_wait(out_wait)
    );

    // Clock generation
    always #5 clk = ~clk;

    // Testbench stimulus
    initial begin
        // Initialize signals
        clk = 0;
        rst_n = 0;
        in_data = 0;

        // Reset
        #20 rst_n = 1;

        // Test with random data
        repeat (1000) begin
            @(posedge clk);
            if (!out_wait) begin
                in_data = $random;
            end
        end

        // End simulation
        #100 $finish;
    end

    // Monitor outputs
    always @(posedge clk) begin
        if (!out_wait) begin
            $display("Time␣%t:␣in_data␣=␣%h,␣out_fragment␣=␣
                ↪ %h", $time, in_data, out_fragment);
        end
    end

endmodule
```

Fragment Memory

proj_fm.sv

```
`timescale 1ns / 1ps

module proj_fm #(
    // Module parameters
    parameter BUFFER_COUNT = proj_pkg::FM_BUFFER_COUNT,
        ↪ // Number of buffers in the FM
    parameter RAMS = proj_pkg::FM_RAMS_COUNT,
```

```
        ↪ // Number of RAMs in each buffer
    parameter ENTRIES = proj_pkg::FM_ENTRIES_COUNT,
        ↪             // Number of entries in each RAM
    parameter OFFSET = proj_pkg::FM_OFFSET_COUNT,
        ↪ // Size of the offset in each entry
    parameter DATA_BITS = proj_pkg::FM_DATA_BITS,           //
        ↪ Width of each memory cell
    parameter INDICE_LEN = proj_pkg::INDICE_LEN,          //
        ↪ Length of the index
    parameter SIGNED_INDICE_LEN = proj_pkg::SIGNED_INDICE_LEN,
        ↪  // Length of signed index
    parameter FRAG_LEN = proj_pkg::FM_EXTENDER_FRAG_LEN_BITS
        ↪            // Length of the fragment - in bits!
) (
    // Module ports
    input  wire                         clk,    // Clock signal
    input  wire                         rst_n,  // Reset signal
        ↪ (active low)
    input  wire [DATA_BITS-1:0]         in_wdata,  // Input data
    input  wire                         chg_idx,   // Change
        ↪ index signal
    input  logic [SIGNED_INDICE_LEN-1:0] frag_idx, // Fragment
        ↪ index
    output wire [FRAG_LEN-1:0]          out_rdata, // Output
        ↪ data
    output wire                         out_wait // wait signal
        ↪ for the module before the ACMI
);

    // Local parameters
    localparam FM_BUFFER_SIZE = 32;  // RAMS * ENTRIES *
        ↪ OFFSET = 2 * 8 * 2
    localparam RAM_ADDR_BITS = 1;    // $clog2(RAMS) =
        ↪ $clog2(2)
    localparam ENTRIES_ADDR_BITS = 3; // $clog2(ENTRIES) =
        ↪ $clog2(8)
    localparam OFFSET_ADDR_BITS = 1;  // $clog2(OFFSET) =
        ↪ $clog2(2)
    localparam ADDR_BITS = 5;         // RAM_ADDR_BITS +
        ↪ ENTRIES_ADDR_BITS + OFFSET_ADDR_BITS

    // Internal signals
    logic clk, rst_n;
    logic [ADDR_BITS-1:0] waddr, waddr_next;
    logic [DATA_BITS-1:0] wdata;
    logic
        ↪ [BUFFER_COUNT-1:0][FM_BUFFER_SIZE-1:0][DATA_BITS-1:0]
        ↪ FMbuffers;
    logic end_addr, hold_cond, rst_addr;
    logic wr_idx, rd_idx;
    logic [FRAG_LEN-1:0] padded_fragment;
    logic [INDICE_LEN-1:0] raddr, zeros_count;
    logic [SIGNED_INDICE_LEN-1:0] flip_frag_idx;
    logic we;
```

```
42
43      // Input assignments
44      assign wdata = hold_cond ? FMbuffers[wr_idx][waddr] :
            ↪ in_wdata;
45
46      // Output assignment
47      assign out_rdata = padded_fragment;
48
49      assign out_wait = hold_cond;
50
51      // Address and control logic
52      assign waddr_next = end_addr ? 1'b0 : waddr + 1'b1;
53      assign end_addr = (waddr == (FM_BUFFER_SIZE-1)) ? 1'b1 :
            ↪ 1'b0;
54      assign hold_cond = end_addr & ~chg_idx;
55      assign rst_addr = end_addr & chg_idx;
56      assign rd_idx = ~wr_idx;
57      assign we = ~hold_cond || end_addr;
58
59      // Fragment preparation logic
60      always_comb begin
61          padded_fragment = '0;
62          zeros_count = '0;
63          flip_frag_idx = '0;
64
65          if (frag_idx[SIGNED_INDICE_LEN-1] == 1'b1) begin
66              flip_frag_idx = (~frag_idx + 1'b1);
67              zeros_count = flip_frag_idx[INDICE_LEN-1:0];
68              raddr = '0;
69          end else begin
70              raddr = frag_idx[INDICE_LEN-1:0];
71          end
72      end
73
74      genvar i;
75      generate
76          for (i = 0; i < 16; i++) begin : gen_padded_fragment
                ↪ // FRAG_LEN is 16
77              always_comb begin
78                  if (i < (16 - zeros_count) && (raddr + i >= 0)
                        ↪ && (raddr + i < 32)) begin  //
                        ↪ FM_BUFFER_SIZE is 32
79                      padded_fragment[i + (zeros_count << 1)] =
                            ↪ FMbuffers[rd_idx][raddr + (i >>
                            ↪ 1)][i[0]];
80                  end else begin
81                      padded_fragment[i + (zeros_count << 1)] =
                            ↪ 1'b0;
82                  end
83              end
84          end
85      endgenerate
86
87      // Sequential logic
```

```
88      always_ff @(posedge clk or negedge rst_n) begin
89          if (~rst_n) begin
90              wr_idx <= '0;
91              waddr <= '0;
92          end else begin
93              // Change index logic
94              if (chg_idx) begin
95                  wr_idx <= rd_idx;
96              end
97
98              // Write address logic
99              if (hold_cond) begin
100                 waddr <= waddr;
101             end else if (rst_addr) begin
102                 waddr <= '0;
103             end else begin
104                 waddr <= waddr_next;
105             end
106
107             // Write data to buffers
108             if (we) begin
109                 FMbuffers[wr_idx][waddr] <= wdata;
110             end
111         end
112     end
113
114 endmodule
```

proj_fm_tb.sv

```
1  `timescale 1ns / 1ps
2  import proj_pkg::*;
3
4  module proj_fm_tb();
5
6      // Parameters
7      localparam BUFFER_COUNT = proj_pkg::FM_BUFFER_COUNT;
           ↪ // Number of buffers in the FM
8      localparam RAMS = proj_pkg::FM_RAMS_COUNT;
           ↪ // Number of RAMs in each buffer
9      localparam ENTRIES = proj_pkg::FM_ENTRIES_COUNT;
           ↪              // Number of entries in each RAM
10     localparam OFFSET = proj_pkg::FM_OFFSET_COUNT;
           ↪               // Size of the offset in each entry
11     localparam DATA_BITS = proj_pkg::FM_DATA_BITS;
           ↪ // Width of each memory cell
12     localparam INDICE_LEN = proj_pkg::INDICE_LEN;          //
           ↪ Length of the index
13     localparam SIGNED_INDICE_LEN =
           ↪ proj_pkg::SIGNED_INDICE_LEN;  // Length of signed
           ↪ index
14     localparam FRAG_LEN = proj_pkg::FM_EXTENDER_FRAG_LEN_BITS;
           ↪            // Length of the fragment - in bits!
```

```
    localparam FM_BUFFER_SIZE = proj_pkg::FM_BUFFER_SIZE;
    localparam KMER_BUFFER_SIZE = proj_pkg::KMER_LEN;
    localparam TESTS_NUM = 10;

    // Signals
    logic [DATA_BITS-1:0] in_wdata;
    logic [DATA_BITS-1:0] data;
    logic [FRAG_LEN-1:0] out_rdata;
    logic clk;
    logic rst_n;
    logic chg_idx;
    logic [SIGNED_INDICE_LEN-1:0] frag_idx;

    // Instantiate the DUT
    proj_fm #(
        .BUFFER_COUNT(BUFFER_COUNT),
        .RAMS(RAMS),
        .ENTRIES(ENTRIES),
        .OFFSET(OFFSET),
        .DATA_BITS(DATA_BITS),
        .INDICE_LEN(INDICE_LEN),
        .SIGNED_INDICE_LEN(SIGNED_INDICE_LEN),
        .FRAG_LEN(FRAG_LEN)
    ) dut (
        .in_wdata(in_wdata),
        .out_rdata(out_rdata),
        .clk(clk),
        .rst_n(rst_n),
        .chg_idx(chg_idx),
        .frag_idx(frag_idx)
    );

    // Clock generation
    always #5 clk = ~clk;

    // Testbench logic
    initial begin
        // Initialize signals
        clk = 0;
        rst_n = 0;
        in_wdata = 0;
        chg_idx = 0;
        frag_idx = 0;

        // Reset
        #10 rst_n = 1;


    for (int test = 0; test < TESTS_NUM; test++) begin
        $display("Starting test %0d", test);

        // Write to the first buffer
        for (int i = 0; i < FM_BUFFER_SIZE; i++) begin
            @(negedge clk);
```

```
            in_wdata = $random & ((1 << DATA_BITS) - 1);
        end
        // Wait KMER_BUFFER_SIZE cycles after writing to the
            ↪ first buffer
        repeat(KMER_BUFFER_SIZE) @(posedge clk);
        // Assert chg_idx on the negedge of the clock
        @(negedge clk);
        chg_idx = 1;
        // Deassert chg_idx on the next negedge
        @(negedge clk);
        chg_idx = 0;

        // Test reading from both buffers
        for (int i = 0; i < BUFFER_COUNT; i++) begin
            frag_idx = 0;
            @(negedge clk);
            $display("Test␣%0d,␣Buffer␣%0d,␣frag_idx␣=␣%0d,␣
                ↪ out_rdata␣=␣%b", test, i, frag_idx,
                ↪ out_rdata);
            frag_idx = FM_BUFFER_SIZE - FRAG_LEN;
            @(negedge clk);
            $display("Test␣%0d,␣Buffer␣%0d,␣frag_idx␣=␣%0d,␣
                ↪ out_rdata␣=␣%b", test, i, frag_idx,
                ↪ out_rdata);
            // Test negative index
            frag_idx = -2;
            @(negedge clk);
            $display("Test␣%0d,␣Buffer␣%0d,␣frag_idx␣=␣%0d,␣
                ↪ out_rdata␣=␣%b", test, i, $signed(frag_idx),
                ↪ out_rdata);
        end


        // Write to the second buffer
        for (int i = 0; i < FM_BUFFER_SIZE; i++) begin
            @(negedge clk);
            in_wdata = $random & ((1 << DATA_BITS) - 1);
        end
        // Wait KMER_BUFFER_SIZE cycles after writing to the
            ↪ first buffer
        repeat(KMER_BUFFER_SIZE) @(posedge clk);
        // Assert chg_idx on the negedge of the clock
        @(negedge clk);
        chg_idx = 1;
        // Deassert chg_idx on the next negedge
        @(negedge clk);
        chg_idx = 0;
        // Test reading from both buffers
        for (int i = 0; i < BUFFER_COUNT; i++) begin
            frag_idx = 0;
            @(negedge clk);
            $display("Test␣%0d,␣Buffer␣%0d,␣frag_idx␣=␣%0d,␣
                ↪ out_rdata␣=␣%b", test, i, frag_idx,
                ↪ out_rdata);
```

```
            frag_idx = FM_BUFFER_SIZE - FRAG_LEN;
            @(negedge clk);
            $display("Test %0d, Buffer %0d, frag_idx = %0d,
                ↪ out_rdata = %b", test, i, frag_idx,
                ↪ out_rdata);
            // Test negative index
            frag_idx = -2;
            @(negedge clk);
            $display("Test %0d, Buffer %0d, frag_idx = %0d,
                ↪ out_rdata = %b", test, i, $signed(frag_idx),
                ↪ out_rdata);
        end

        $display("Finished test %0d", test);
    end

    // End simulation
    #20 $finish;
end

endmodule
```

Hasher

proj_hasher.sv

```
`timescale 1ns / 1ps

module proj_hasher
  #(
    parameter KMER_LEN = proj_pkg::KMER_LEN,
    parameter DATA_BITS = proj_pkg::BASE_LEN,
    parameter HASHER_DATA_BITS =
        ↪ proj_pkg::HASHER_SORTER_SIGNATURE
  )(
    input  wire                          clk,    // Clock signal
    input  wire                          rst_n,  // Reset signal
        ↪ (active low)
    input wire [HASHER_DATA_BITS-1:0]  seed,
    input wire [HASHER_DATA_BITS-1:0]  kmer,
    output logic [HASHER_DATA_BITS-1:0] signature
  );

  // Intermediate signals
  logic [HASHER_DATA_BITS:0]        k, key;

  localparam [HASHER_DATA_BITS-1:0]   c1 = 32'hcc9e2d51;
  localparam [HASHER_DATA_BITS-1:0]   c2 = 32'h1b873593;
  localparam [HASHER_DATA_BITS-1:0]   m = 32'd5;
  localparam [HASHER_DATA_BITS-1:0]   n = 32'he6546b64;

  // Hash computation
  always_comb begin
```

```
26        k = kmer;
27        k = k * c1;
28        k = {k[HASHER_DATA_BITS-16:0],
              ↪ k[HASHER_DATA_BITS-1:HASHER_DATA_BITS-15]}; //
              ↪ ROL15
29            k = k * c2;
30            key = seed;
31        key = key ^ k;
32        key = {key[HASHER_DATA_BITS-14:0],
              ↪ key[HASHER_DATA_BITS-1:HASHER_DATA_BITS-13]}; //
              ↪ ROL13
33        signature = key * m + n;
34    end
35
36  endmodule
```

proj_hasher_tb.sv

```
1  `timescale 1ns / 1ps
2  import proj_pkg::*;  // Include the package
3  module proj_hasher_tb;
4
5      parameter KMER_LEN = proj_pkg::KMER_LEN;
6      parameter BASE_BITS = proj_pkg::BASE_LEN;
7      parameter HASHER_DATA_BITS =
           ↪ proj_pkg::HASHER_SORTER_SIGNATURE;
8
9    reg [HASHER_DATA_BITS-1:0]  seed_tb;
10   reg [HASHER_DATA_BITS-1:0]  kmer_tb;
11   wire [HASHER_DATA_BITS-1:0] signature_tb;
12
13   proj_hasher #(.HASHER_DATA_BITS(HASHER_DATA_BITS))
14     dut (
15     .seed(seed_tb),
16     .kmer(kmer_tb),
17     .signature(signature_tb)
18   );
19
20   initial begin
21     // Initialize Inputs
22     seed_tb = 0;
23     kmer_tb = 0;
24     #10;
25
26     // Apply Test Vectors
27     seed_tb = 32'hac718add;
28     kmer_tb = 32'hab1020c5;
29
30     #10;
31
32     // Display the result
33     $display("Seed: %h, Kmer: %h, Signature: %h", seed_tb,
           ↪ kmer_tb, signature_tb);
```

```
34
35
36      // Test reading from both buffers
37      for (int i = 0; i < HASHER_DATA_BITS; i++) begin
38      kmer_tb =  $random ;
39
40      #10;
41
42      // Display the result
43      $display("Seed: %h, Kmer: %h, Signature: %h", seed_tb,
            ↪ kmer_tb, signature_tb);
44      end
45      #10;
46      $finish;
47    end
48
49  endmodule
```

Sorter

proj_sorter.sv

```
1  import proj_pkg::*;  // Include the project package
2
3  module proj_sorter #(
4      // Module parameters, using values from the project package
5      parameter INDICES_COUNT =
           ↪ proj_pkg::SORTER_EXTENDER_INDICES_COUNT,
6      parameter INDICE_LEN = proj_pkg::INDICE_LEN,
7      parameter SIGNATURE_LEN =
           ↪ proj_pkg::HASHER_SORTER_SIGNATURE,
8      parameter POSITION_LEN = proj_pkg::SORTER_POSITION_LEN
9  )(
10     // Module inputs and outputs
11     input wire [SIGNATURE_LEN-1:0] in_signature,
12     input wire [INDICE_LEN-1:0] in_index,
13     output wire [INDICES_COUNT-1:0][INDICE_LEN-1:0]
           ↪ out_smallest_idx,
14     input wire rst_n,
15     input wire end_sorting,
16     output wire sort_valid,
17     input wire clk
18 );
19     // Internal signals
20     signature_index_pack [INDICES_COUNT-1:0] smallest_idx_next;
21     signature_index_pack [INDICES_COUNT-1:0] smallest_idx_curr;
22     signature_index_pack new_pack;
23     logic [INDICES_COUNT-1:0] count_signatures_smaller_than;
24     logic [POSITION_LEN:0] position_smaller_than;
25     logic [POSITION_LEN:0] new_position_long;
26     logic [POSITION_LEN-1:0] new_position;
27
28     // Assign input values to internal signals
```

```
29      assign new_pack.signature = in_signature;
30      assign new_pack.index = in_index;
31
32
33      always_comb begin
34          count_signatures_smaller_than = '0;
35          position_smaller_than = '0;
36          new_position = '0;
37          new_position_long = '0;
38          for (int i = 0; i < INDICES_COUNT; i= i+1) begin
39              count_signatures_smaller_than[i] =
                    ↪ (new_pack.signature <
                    ↪ smallest_idx_curr[i].signature) ? 1'b1 :
                    ↪ 1'b0;
40              position_smaller_than +=
                    ↪ count_signatures_smaller_than[i];
41              if (position_smaller_than == '0) begin
42                  smallest_idx_next[i].signature =
                        ↪ smallest_idx_curr[i].signature;
43                  smallest_idx_next[i].index =
                        ↪ smallest_idx_curr[i].index;
44              end else begin
45                  if (i < new_position) begin
46                      smallest_idx_next[i].signature =
                            ↪ smallest_idx_curr[i].signature;
47                      smallest_idx_next[i].index =
                            ↪ smallest_idx_curr[i].index;
48                  end else if (i > new_position) begin
49                      smallest_idx_next[i].signature =
                            ↪ smallest_idx_curr[i-1].signature;
50                      smallest_idx_next[i].index =
                            ↪ smallest_idx_curr[i-1].index;
51                  end
52              end
53          end
54          new_position_long = INDICES_COUNT -
                ↪ position_smaller_than;
55          new_position = new_position_long[POSITION_LEN-1:0];
56      end
57
58
59      // Assign sorted indices to output
60      generate
61          for (genvar j = 0; j < INDICES_COUNT; j++) begin
62              assign out_smallest_idx[j] = end_sorting ?
                    ↪ smallest_idx_curr[j].index : 'x;
63          end
64      endgenerate
65
66      assign sort_valid = end_sorting ? 1'b1 : 1'b0;
67
68      // Sequential logic for updating current smallest indices
69      always_ff @(posedge clk) begin
70          for (int i = 0; i < INDICES_COUNT; i++) begin
```

```
            if (~rst_n) begin
                // Reset values
                smallest_idx_curr[i].signature <= '1;
                smallest_idx_curr[i].index <= '0;
            end else begin
                // Update with new values
                if ((i == new_position) &
                    ↪ (position_smaller_than != '0)) begin
                    smallest_idx_curr[i].signature <=
                        ↪ new_pack.signature;
                    smallest_idx_curr[i].index <=
                        ↪ new_pack.index;
                end else begin
                    smallest_idx_curr[i].signature <=
                        ↪ smallest_idx_next[i].signature;
                    smallest_idx_curr[i].index <=
                        ↪ smallest_idx_next[i].index;
                end
            end
        end
    end
endmodule
```

proj_sorter_tb.sv

```
`timescale 1ns / 1ps
import proj_pkg::*;

module proj_sorter_tb;
    // Parameters
    localparam INDICES_COUNT =
        ↪ proj_pkg::SORTER_EXTENDER_INDICES_COUNT;
    localparam INDICE_LEN = proj_pkg::INDICE_LEN;
    localparam SIGNATURE_LEN =
        ↪ proj_pkg::HASHER_SORTER_SIGNATURE;

    // Inputs
    logic [SIGNATURE_LEN-1:0] in_signature;
    logic [INDICE_LEN-1:0] in_index;
    logic in_rst_n;
    logic in_clk;
    logic end_sorting;

    // Outputs
    logic [INDICES_COUNT-1:0][INDICE_LEN-1:0] out_smallest_idx;
    logic sort_valid;

    // Instantiate the Unit Under Test (UUT)
    proj_sorter #(
        .INDICES_COUNT(INDICES_COUNT),
        .INDICE_LEN(INDICE_LEN),
        .SIGNATURE_LEN(SIGNATURE_LEN)
    ) dut (
```

```
        .in_signature(in_signature),
        .in_index(in_index),
        .out_smallest_idx(out_smallest_idx),
        .rst_n(in_rst_n),
        .clk(in_clk),
        .end_sorting(end_sorting),
        .sort_valid(sort_valid)
    );

    // Clock generation
    always #5 in_clk = ~in_clk;

    // Test procedure
    initial begin
        // Initialize inputs
        in_signature = 0;
        in_index = 0;
        in_rst_n = 0;
        in_clk = 0;
        end_sorting = 0;

        // Reset
        #10 in_rst_n = 1;

        // Test case 1: Insert values in descending order
        for (int i = 10; i > 0; i--) begin
            in_signature = i * 32'h10101010;
            in_index = i;
            #10;
        end
        end_sorting = 1;
        in_signature = 32'hA0A0A0A0;
        in_index = 10;
        #10;
        end_sorting = 0;
        #10;

        // Test case 2: Insert values in ascending order
        for (int i = 1; i <= 10; i++) begin
            in_signature = i * 32'h01010101;
            in_index = i;
            #10;
        end
        end_sorting = 1;
        in_signature = 32'hA0A0A0A0;
        in_index = 10;
        #10;
        end_sorting = 0;
        #10;

        // Test case 3: Insert random values
        for (int i = 0; i < 20; i++) begin
            in_signature = $urandom();
            in_index = $urandom_range(0, 255);
```

```
            #10;
        end
        end_sorting = 1;
        #10;
        end_sorting = 0;
        #10;

        // Test case 4: Reset and insert new values
        in_rst_n = 0;
        #10;
        in_rst_n = 1;
        for (int i = 0; i < INDICES_COUNT * 2; i++) begin
            in_signature = $urandom();
            in_index = $urandom_range(0, 255);
            #10;
        end
        end_sorting = 1;
        #10;
        end_sorting = 0;
        #10;

        // End simulation
        #100;
        $finish;
    end

    // Monitor
    always @(posedge in_clk) begin
        $display("Time=%0t, in_signature=%h, in_index=%h, end_sorting=%b, sort_valid=%b",
                 $time, in_signature, in_index, end_sorting, sort_valid);
        if (sort_valid) begin
            for (int i = 0; i < INDICES_COUNT; i++) begin
                $display("  out_smallest_idx[%0d]=%h", i, out_smallest_idx[i]);
            end
            $display("--------------------");
        end
    end

    // Checker
    always @(posedge in_clk) begin
        if (in_rst_n) begin
            for (int i = 1; i < INDICES_COUNT; i++) begin
                assert(dut.smallest_idx_curr[i].signature >= dut.smallest_idx_curr[i-1].signature)
                else $error("Sorting error: smallest_idx_curr[%0d].signature=%h is smaller than smallest_idx_curr[%0d].signature=%h",
                            i, dut.smallest_idx_curr[i].signature, i-1,
```

```
                                    ↪ dut.smallest_idx_curr[i-1].signature);
126             end
127         end
128     end
129
130     // Check sort_valid
131     always @(posedge in_clk) begin
132         assert(sort_valid == end_sorting)
133         else $error("sort_valid mismatch: sort_valid=%b, 
                ↪ end_sorting=%b", sort_valid, end_sorting);
134     end
135 endmodule
```

Extender

proj_extender.sv

```
1  // Timescale directive for simulation
2  `timescale 1ns / 1ps
3  // Import the project package
4  import proj_pkg::*;
5  // Module declaration with parameters
6  module proj_extender #(
7      parameter FRAG_LEN_BITS =
           ↪ proj_pkg::FM_EXTENDER_FRAG_LEN_BITS,
8      parameter FRAG_SIZE = proj_pkg::FRAG_LEN,
9      parameter KMER_SIZE = proj_pkg::KMER_LEN,
10     parameter INDICES_COUNT =
           ↪ proj_pkg::SORTER_EXTENDER_INDICES_COUNT,
11     parameter INDICE_LEN = proj_pkg::INDICE_LEN,
12     parameter SIGNED_INDICE_LEN = proj_pkg::SIGNED_INDICE_LEN,
13     parameter FRAG_PART_ONE_HOT =
           ↪ proj_pkg::EXTENDER_OUT_PART_LEN_ONE_HOT,
14     parameter BASE_LEN = proj_pkg::BASE_LEN,
15     parameter ONE_HOT_LEN = proj_pkg::ONE_HOT_LEN,
16     parameter FRAG_PART = proj_pkg::EXTENDER_OUT_PART_LEN
17 )(
18     // Input ports
19     input logic [FRAG_LEN_BITS-1:0] in_fragment,
20     input logic [INDICES_COUNT-1:0][INDICE_LEN-1:0]
           ↪ in_kmer_indices,
21     input wire valid_indices,
22     input wire rst_n,
23     input wire clk,
24     // Output ports
25     output logic [SIGNED_INDICE_LEN-1:0] out_index,
26     output logic [FRAG_PART_ONE_HOT-1:0] out_gfm
27 );
28     // Local parameters
29     localparam FRAG_PARTS_COUNT = (FRAG_LEN_BITS >>
           ↪ $clog2(FRAG_PART));
30     localparam FRAG_PARTS_COUNT_BITS =
           ↪ $clog2(FRAG_PARTS_COUNT);
```

```
31      localparam INDICES_COUNT_BITS = $clog2(INDICES_COUNT);
32      // Internal signals
33      logic [FRAG_PARTS_COUNT_BITS-1:0] frag_parts_idx;
34      logic [FRAG_PARTS_COUNT_BITS-1:0] frag_parts_idx_next;
35      logic rst_frag_parts_idx;
36      logic [INDICE_LEN-1:0] curr_index;
37      logic [INDICES_COUNT_BITS-1:0] indices_idx;
38      logic [INDICES_COUNT_BITS-1:0] indices_idx_next;
39      logic [FRAG_PART-1:0] frag_part;
40      logic [INDICES_COUNT-1:0][INDICE_LEN-1:0]
           ↪ in_kmer_indices_r;
41
42      // Combinational logic
43      // Reset fragment parts index when it reaches the maximum
44      assign rst_frag_parts_idx = (frag_parts_idx ==
           ↪ (FRAG_PARTS_COUNT - 1)) ? 1'b1 : 1'b0;
45      // Calculate next fragment parts index
46      assign frag_parts_idx_next = rst_frag_parts_idx ? 1'b0 :
           ↪ frag_parts_idx + 1'b1;
47      // Select current index from input indices
48      assign curr_index = in_kmer_indices_r[indices_idx];
49      // Extract fragment part for output
50      assign frag_part = in_fragment[FRAG_PART*frag_parts_idx +:
           ↪ FRAG_PART];
51
52      generate
53          for (genvar i = 0; i < FRAG_PART >> $clog2(BASE_LEN);
               ↪ i++) begin : gen_gfm
54              always_comb begin
55                  case (frag_part[i*BASE_LEN +: BASE_LEN])
56                      2'b00: out_gfm[i*ONE_HOT_LEN +:
                           ↪ ONE_HOT_LEN] = 4'b0001;
57                      2'b01: out_gfm[i*ONE_HOT_LEN +:
                           ↪ ONE_HOT_LEN] = 4'b0010;
58                      2'b10: out_gfm[i*ONE_HOT_LEN +:
                           ↪ ONE_HOT_LEN] = 4'b0100;
59                      2'b11: out_gfm[i*ONE_HOT_LEN +:
                           ↪ ONE_HOT_LEN] = 4'b1000;
60                      default: out_gfm[i*ONE_HOT_LEN +:
                           ↪ ONE_HOT_LEN] = 4'b0000;
61                  endcase
62              end
63          end
64      endgenerate
65      // Calculate next indices index
66      assign indices_idx_next = rst_frag_parts_idx ? indices_idx
           ↪ + 1'b1 : indices_idx;
67      // Calculate output index
68      assign out_index = {1'b0, curr_index} -
           ↪ SIGNED_INDICE_LEN'(((FRAG_SIZE - KMER_SIZE) >> 1));
69      // Sequential logic for fragment parts index
70      always_ff @(posedge clk or negedge rst_n) begin :
           ↪ parts_index
71          if (~rst_n) begin
```

```
            frag_parts_idx <= '0;
        end else if (valid_indices) begin
            frag_parts_idx <= '0;
        end else begin
            frag_parts_idx <= frag_parts_idx_next;
        end
    end

    // Sequential logic for indices sample
    always_ff @(posedge clk or negedge rst_n) begin : index_sample
        if (valid_indices) begin
            in_kmer_indices_r <= in_kmer_indices;
        end
    end

    // Sequential logic for indices index
    always_ff @(posedge clk or negedge rst_n) begin : indices_index
        if (~rst_n) begin
            indices_idx <= '0;
        end else if (valid_indices) begin
            indices_idx <= '0;
        end else begin
            indices_idx <= indices_idx_next;
        end
    end
endmodule
```

proj_extender_tb.sv

```
`timescale 1ns / 1ps
import proj_pkg::*;

module proj_extender_tb;
    // Define local parameters
    localparam FRAG_LEN_BITS = proj_pkg::FM_EXTENDER_FRAG_LEN_BITS;
    localparam FRAG_SIZE = proj_pkg::FRAG_LEN;
    localparam KMER_SIZE = proj_pkg::KMER_LEN;
    localparam INDICES_COUNT = proj_pkg::SORTER_EXTENDER_INDICES_COUNT;
    localparam INDICE_LEN = proj_pkg::INDICE_LEN;
    localparam SIGNED_INDICE_LEN = proj_pkg::SIGNED_INDICE_LEN;
    localparam FRAG_PART = proj_pkg::EXTENDER_OUT_PART_LEN;
    localparam FRAG_PART_ONE_HOT = proj_pkg::EXTENDER_OUT_PART_LEN_ONE_HOT;
    localparam BASE_LEN = proj_pkg::BASE_LEN;
    localparam ONE_HOT_LEN = proj_pkg::ONE_HOT_LEN;

    // TB parameters
    localparam MEM_WIDTH = 32;
    localparam MEM_DEPTH = 32;
```

```
    localparam FRAG_PARTS_COUNT = (FRAG_LEN_BITS >>
        ↪ $clog2(FRAG_PART));

    // Declare input and output signals
    logic [FRAG_LEN_BITS-1:0] in_fragment;
    logic [INDICES_COUNT-1:0][INDICE_LEN-1:0] in_kmer_indices;
    logic valid_indices;
    logic rst_n;
    logic clk;
    logic [SIGNED_INDICE_LEN-1:0] out_index;
    logic [FRAG_PART_ONE_HOT-1:0] out_gfm;

    // Declare external memory and padded fragment
    logic [MEM_WIDTH-1:0] ext_mem;
    logic [FRAG_LEN_BITS-1:0] padded_fragment;

    // Implement padding logic
    always_comb begin
        padded_fragment = '0; // Initialize with zeros
        for (int i = 0; i < FRAG_LEN_BITS; i++) begin
            // Check if index is within valid range
            if ((out_index + i >= 0) && (out_index + i <
                ↪ MEM_WIDTH)) begin
                padded_fragment[i] = ext_mem[out_index + i];
            end
            // Else, keep as 0 (padded)
        end
    end

    int step_count = 0;
    int current_kmer_index = 0;

    // Assign padded fragment to input fragment
    assign in_fragment = padded_fragment;

    // Instantiate the Unit Under Test (UUT)
    proj_extender #(
        .FRAG_LEN_BITS(FRAG_LEN_BITS),
        .FRAG_SIZE(FRAG_SIZE),
        .KMER_SIZE(KMER_SIZE),
        .INDICES_COUNT(INDICES_COUNT),
        .INDICE_LEN(INDICE_LEN),
        .SIGNED_INDICE_LEN(SIGNED_INDICE_LEN),
        .FRAG_PART(FRAG_PART),
        .FRAG_PART_ONE_HOT(FRAG_PART_ONE_HOT),
        .BASE_LEN(BASE_LEN),
        .ONE_HOT_LEN(ONE_HOT_LEN)
    ) dut (
        .in_fragment(in_fragment),
        .in_kmer_indices(in_kmer_indices),
        .valid_indices(valid_indices),
        .rst_n(rst_n),
        .clk(clk),
        .out_index(out_index),
```

```
72          .out_gfm(out_gfm)
73      );
74
75      // Clock generation
76      always begin
77          #5 clk = ~clk;
78      end
79
80      // Test procedure
81      initial begin
82          clk = 0;
83          rst_n = 0;
84          valid_indices = 0;
85          #10 rst_n = 1;
86
87          // Initialize external memory with random data
88          for (int i = 0; i < MEM_DEPTH; i++) begin
89              ext_mem = $random;
90          end
91
92          // Generate random indices
93          for (int i = 0; i < INDICES_COUNT; i++) begin
94              in_kmer_indices[i] = $urandom_range(0, 31);
95          end
96
97          // Run 10 test cases
98          for (int test = 0; test < 10; test++) begin
99              $display("Test case %0d:", test);
100
101             // Randomize external memory for each test
102             for (int i = 0; i < MEM_DEPTH; i++) begin
103                 ext_mem = $random;
104             end
105
106             // Generate new random indices for each test
107             for (int i = 0; i < INDICES_COUNT; i++) begin
108                 @(negedge clk);
109                 in_kmer_indices[i] = $urandom_range(0, 31);
110             end
111
112             // Assert valid_indices for one clock cycle
113             @(negedge clk);
114             valid_indices = 1;
115             @(negedge clk);
116             valid_indices = 0;
117
118             // Wait for the module to process all fragment
                    ↪ parts and indices
119             for (int i = 0; i < INDICES_COUNT; i++) begin
120                 for (int j = 0; j < FRAG_PARTS_COUNT; j++)
                        ↪ begin
121                     @(posedge clk);
122                     print_step_info(i);
123                 end
```

```
            end

            $display("\n");  // Add a blank line between test cases
        end

        // End simulation
        $finish;
    end

    // Function to print step information
    function void print_step_info(int kmer_index);
        step_count++;
        $display("Step %0d:", step_count);
        $display("  Current kmer_indices[%0d]: 0x%h", kmer_index, in_kmer_indices[kmer_index]);
        $display("  out_index: 0x%h", out_index);
        $display("  in_fragment: 0x%h", in_fragment);
        $display("  out_gfm (one-hot) - part %0d of the in_fragment: 0b%b (binary), 0x%h (hexadecimal)",dut.frag_parts_idx + 1, out_gfm, out_gfm);
        $display("");  // Add a blank line for readability
    endfunction

    // Add this to stop the simulation after a certain number of steps or time
    initial begin
        #10000;  // Stop after 10,000 time units, adjust as needed
        $display("Simulation stopped due to timeout");
        $finish;
    end

endmodule
```

Index Counter

proj_index_counter.sv

```
import proj_pkg::*;  // Include the package

module proj_counter
#(
    parameter INDICE_LEN = proj_pkg::INDICE_LEN
)
(
    output wire [INDICE_LEN-1:0] index,
    output logic finished_count,
    input wire clk,
    input wire rst_n,
    input wire start
);
```

```
    // Internal signals
    logic [INDICE_LEN-1:0] out_index;
    logic end_of_count;
    logic [INDICE_LEN-1:0] idx_next;
    logic rst_index;
    logic count_enabled;
    logic start_prev;

    // Check if write address reached the end of the buffer
    assign end_of_count = (out_index == (FM_BUFFER_SIZE-1)) ?
        ↪ 1'b1 : 1'b0;
    assign finished_count = end_of_count;

    // Increment write address or keep current value
    assign idx_next = (count_enabled & ~end_of_count) ?
        ↪ out_index + 1'b1 : out_index;

    // Reset index when end of count is reached or on reset
    assign rst_index = end_of_count | (~rst_n);

    // Assign output index
    assign index = out_index;

    // Sequential logic for updating the index and detecting
        ↪ start signal
    always_ff @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            out_index <= '0;
            count_enabled <= 1'b0;
            start_prev <= 1'b0;
        end else begin
            start_prev <= start;
            if (rst_index) begin
                out_index <= '0;
                count_enabled <= 1'b0;
            end else if (start & ~start_prev) begin
                // Rising edge of start signal
                count_enabled <= 1'b1;
                out_index <= '0;
            end else if (count_enabled) begin
                out_index <= idx_next;
            end
        end
    end
endmodule
```

proj_index_counter_tb.sv_

```
`timescale 1ns / 1ps
import proj_pkg::*;

module proj_counter_tb();
    // Parameters
```

```
    localparam CLK_PERIOD = 10;  // 10 ns clock period
    localparam FM_BUFFER_SIZE = proj_pkg::FM_BUFFER_SIZE;

    // Signals
    logic clk;
    logic rst_n;
    logic [proj_pkg::FM_BUFFER_SIZE-1:0] index;
    logic finished_count;
    logic start;

    // Instantiate the Unit Under Test (UUT)
    proj_counter #(
        .FM_BUFFER_SIZE(FM_BUFFER_SIZE)
    )
    dut (
        .index(index),
        .clk(clk),
        .rst_n(rst_n),
        .finished_count(finished_count),
        .start(start)
    );

    // Clock generation
    always begin
        clk = 1'b0;
        #(CLK_PERIOD/2);
        clk = 1'b1;
        #(CLK_PERIOD/2);
    end

    // Test variables
    int error_count = 0;
    logic [proj_pkg::FM_BUFFER_SIZE-1:0] expected_index;

    // Test procedure
    initial begin
        $display("Starting␣proj_counter␣testbench");

        // Initialize signals
        rst_n = 1'b0;
        start = 1'b0;
        expected_index = '0;

        // Reset the module
        #(CLK_PERIOD);
        rst_n = 1'b1;


        // Check initial value
        if (index !== expected_index) begin
            $display("Error:␣Initial␣index␣mismatch.␣Expected␣
                ↪ %0d,␣got␣%0d", expected_index, index);
            error_count++;
        end
```

```
59          else begin
60              $display("Passed: Initial index match. Expected
                    ↪ %0d, got %0d", expected_index, index);
61          end
62
63          // Test normal operation
64          start = 1'b1;
65          #(CLK_PERIOD);
66          start = 1'b0;
67
68          // Check that index is 0 immediately after starting
                ↪ the count
69          if (index !== '0) begin
70              $display("Error: Index not 0 after starting count.
                    ↪ Expected 0, got %0d at time %0t", index,
                    ↪ $time);
71              error_count++;
72          end else begin
73              $display("Passed: Index is 0 after starting count
                    ↪ at time %0t", $time);
74          end
75
76          for (int i = 1; i < proj_pkg::FM_BUFFER_SIZE; i++)
                ↪ begin
77              @(posedge clk);
78              expected_index = i;
79              #1;
80              if (index !== expected_index) begin
81                  $display("Error: Index mismatch at cycle %0d.
                        ↪ Expected %0d, got %0d at time %0t", i,
                        ↪ expected_index, index, $time);
82                  error_count++;
83              end else begin
84                  $display("Passed: Index match at cycle %0d.
                        ↪ Expected %0d, got %0d at time %0t", i,
                        ↪ expected_index, index, $time);
85              end
86          end
87
88          // Check finished_count
89          #1;
90          if (finished_count !== 1'b1) begin
91              $display("Error: finished_count not asserted at
                    ↪ the end of count");
92              error_count++;
93          end else begin
94              $display("Passed: finished_count asserted at the
                    ↪ end of count");
95          end
96
97          repeat(10) begin
98              @(posedge clk);
99          end
100
```

```
101         // Test normal operation again
102         @(negedge clk);
103         start = 1'b1;
104         @(negedge clk);
105         start = 1'b0;
106 
107         // Check that index is 0 immediately after starting
                ↪ the count
108         if (index !== '0) begin
109             $display("Error: Index not 0 after starting count.
                    ↪ Expected 0, got %0d at time %0t", index,
                    ↪ $time);
110             error_count++;
111         end else begin
112             $display("Passed: Index is 0 after starting count
                    ↪ at time %0t", $time);
113         end
114 
115         for (int i = 1; i < proj_pkg::FM_BUFFER_SIZE; i++)
                ↪ begin
116             @(posedge clk);
117             expected_index = i;
118             #1;
119             if (index !== expected_index) begin
120                 $display("Error: Index mismatch at cycle %0d.
                        ↪ Expected %0d, got %0d at time %0t", i,
                        ↪ expected_index, index, $time);
121                 error_count++;
122             end else begin
123                 $display("Passed: Index match at cycle %0d.
                        ↪ Expected %0d, got %0d at time %0t", i,
                        ↪ expected_index, index, $time);
124             end
125         end
126 
127         // Check finished_count again
128         #1;
129         if (finished_count !== 1'b1) begin
130             $display("Error: finished_count not asserted at
                    ↪ the end of count");
131             error_count++;
132         end else begin
133             $display("Passed: finished_count asserted at the
                    ↪ end of count");
134         end
135 
136         // Final result
137         if (error_count == 0) begin
138             $display("\033[32m********** PROJ_COUNTER TEST
                    ↪ PASSED **********\033[0m");
139         end
140         else begin
141             $display("Test FAILED: %0d errors detected",
                    ↪ error_count);
```

```
142             $display("Failure reasons:");
143             $display("1. Initial index after reset might be
                    ↪ incorrect");
144             $display("2. Index might not increment correctly");
145             $display("3. Index might not wrap around at
                    ↪ FM_BUFFER_SIZE");
146             $display("4. Reset during operation might not work
                    ↪ as expected");
147             $display("5. finished_count signal might not
                    ↪ assert correctly");
148             $display("6. Counter might not respond correctly
                    ↪ to the start signal");
149         end
150         $finish;
151         end
152
153 endmodule
```

Kmer Buffer

proj_kmer_buffer.sv

```
1  `timescale 1ns / 1ps
2  import proj_pkg::*;  // Include the project package
3
4  module proj_kmer_buffer #(
5      parameter DATA_BITS = proj_pkg::KMER_BUFFER_BITS,     //
           ↪ Number of bits for each nucleotide
6      parameter KMER_LEN = proj_pkg::KMER_BUFFER_LEN,     //
           ↪ Length of the k-mer
7      parameter OUT_KMER = KMER_LEN * DATA_BITS  // Total bits
           ↪ in the output k-mer
8  )(
9      input wire clk,              // Clock input
10     input wire rst_n,            // Active-low reset
11     input wire [DATA_BITS-1:0] in_data,  // Input nucleotide
           ↪ data
12     output wire [KMER_LEN-1:0][DATA_BITS-1:0] out_kmer,  //
           ↪ Output k-mer
13     input  wire start_over,      // Signal to reset the buffer
14     output logic full            // Indicates when the buffer
           ↪ is full
15 );
16     // Internal signals
17     logic [KMER_LEN*DATA_BITS-1:0] kmer_buffer;     // Current
           ↪ k-mer buffer
18     logic [KMER_LEN*DATA_BITS-1:0] kmer_buffer_nxt; // Next
           ↪ state of k-mer buffer
19     logic [$clog2(KMER_LEN)-1:0] buffer_count;           //
           ↪ Current count of nucleotides in buffer
20     logic [$clog2(KMER_LEN)-1:0] buffer_count_nxt;       //
           ↪ Next state of buffer count
21
```

```
    // Increment buffer count
    assign buffer_count_nxt = buffer_count + 1'b1;

    // Determine if buffer is full
    assign buffer_full = (buffer_count == KMER_LEN - 1) ? 1'b1
        ↪ : 1'b0;
    assign full = buffer_full;

    // Generate next state of k-mer buffer
    generate
        for (genvar i = 2; i < KMER_LEN; i++) begin
          assign kmer_buffer_nxt[0] = in_data[0];
          assign kmer_buffer_nxt[1] = in_data[1];
          assign kmer_buffer_nxt[i] = kmer_buffer[i-1];  //
              ↪ Shift existing data
        end
    endgenerate

    // Connect internal buffer to output
    assign out_kmer = kmer_buffer;

    // Update k-mer buffer on clock edge
    always_ff @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
          kmer_buffer <= '0;  // Reset buffer on active-low
              ↪ reset
        end else if (start_over) begin
          kmer_buffer <= '0;  // Reset buffer on start_over
              ↪ signal
        end else begin
          kmer_buffer <= kmer_buffer_nxt;  // Update buffer
              ↪ with next state
        end
    end

    // Update buffer count on clock edge
    always_ff @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
          buffer_count <= '0;  // Reset count on active-low
              ↪ reset
        end else if (start_over) begin
          buffer_count <= '0;  // Reset count on start_over
              ↪ signal
        end else if (buffer_full) begin
          buffer_count <= buffer_count;  // Maintain count
              ↪ when buffer is full
        end else begin
          buffer_count <= buffer_count_nxt;  // Increment count
        end
    end
endmodule
```

proj_kmer_buffer_tb.sv

```
`timescale 1ns / 1ps
import proj_pkg::*;
module proj_kmer_buffer_tb();
    // Parameters
    localparam DATA_BITS = proj_pkg::KMER_BUFFER_BITS;
    localparam KMER_LEN = proj_pkg::KMER_BUFFER_LEN;
    localparam OUT_KMER = KMER_LEN * DATA_BITS;
    localparam CLK_PERIOD = 10; // 10ns clock period
    // Signals
    logic clk;
    logic rst_n;
    logic [DATA_BITS-1:0] in_data;
    logic [KMER_LEN-1:0][DATA_BITS-1:0] out_kmer;
    logic start_over;
    logic full;
    // Instantiate the DUT
    proj_kmer_buffer #(
        .DATA_BITS(DATA_BITS),
        .KMER_LEN(KMER_LEN),
        .OUT_KMER(OUT_KMER)
    ) dut (
        .clk(clk),
        .rst_n(rst_n),
        .in_data(in_data),
        .out_kmer(out_kmer),
        .start_over(start_over),
        .full(full)
    );
    int j;
    // Clock generation
    always #(CLK_PERIOD/2) clk = ~clk;
    // Test stimulus
    initial begin
        $display("Starting proj_kmer_buffer testbench");
        // Initialize signals
        clk = 0;
        rst_n = 0;
        in_data = 0;
        start_over = 0;
        $display("Initializing signals: clk=%b, rst_n=%b,
            ↪ in_data=%b, start_over=%b", clk, rst_n, in_data,
            ↪ start_over);
        // Reset
        #(CLK_PERIOD*2) rst_n = 1;
        $display("Reset complete: rst_n=%b", rst_n);
        // Test case 1: Fill the buffer
        $display("Starting Test Case 1: Filling the buffer");
        for (int i = 0; i < KMER_LEN; i++) begin
            in_data = $random;
            $display("Cycle %0d: in_data=%b", i, in_data);
            #CLK_PERIOD;
        end
        $display("Buffer filled. Waiting for %0d cycles",
```

```
            ↪ KMER_LEN);
        for (int i = 0; i < KMER_LEN; i++) begin
            in_data = $random;
            $display("Cycle␣%0d:␣out_kmer=%b", i+KMER_LEN,
                ↪ out_kmer);
            #CLK_PERIOD;
        end
        // Test case 2: Start over
        $display("Starting␣Test␣Case␣2:␣Start␣over");
        @(negedge clk)
        start_over = 1;
        $display("start_over␣asserted:␣start_over=%b",
            ↪ start_over);
        @(negedge clk)
        start_over = 0;
        $display("start_over␣de-asserted:␣start_over=%b",
            ↪ start_over);
        // Fill buffer again
        $display("Filling␣buffer␣again");
        for (int i = 0; i < KMER_LEN; i++) begin
            @(negedge clk)
            j = i+1;
            in_data = j[DATA_BITS-1:0];
            $display("Cycle␣%0d:␣in_data=%b", i, in_data);
        end
        // Test case 3: Continuous input after full
        $display("Starting␣Test␣Case␣3:␣Continuous␣input␣after␣
            ↪ full");
        for (int i = 0; i < 5; i++) begin
            @(negedge clk)
            in_data = $random;
            $display("Additional␣input␣%0d:␣in_data=%b", i,
                ↪ in_data);
            $display("out_kmer=%b", out_kmer);
        end
        $display("Testbench␣completed");
        $finish;
    end
endmodule
```