The Andrew and Erna Viterbi Faculty of ELECTRICAL & COMPUTER ENGINEERING

TECHNION Israel Institute of Technology

VLSI המעבדה ל- Laboratory

Hardware Accelerator for DNA Local Sequence Alignment

Author:

Eliyahu Wiener, 203638341

Supervisor:

Zuher jashan

VLSI LAB

Winter 2025

Abstract

Local sequence alignment is a technique used to identify regions of similarity within biological sequences, such as DNA, RNA, or proteins. This process plays a vital role in genomics and bioinformatics, enabling the understanding of evolutionary relationships and functional characteristics. The Smith-Waterman algorithm is a well-known method for achieving local sequence alignment, offering optimal accuracy through dynamic programming. However, its computational complexity creates significant challenges when dealing with large genomic datasets. This project proposes designing, architecting, and partially implementing an ASIC hardware accelerator to perform local sequence alignment efficiently. By leveraging parallelism and concurrency, the proposed solution aims to mitigate computational bottlenecks, offering reduced runtime and enhanced performance through a dedicated hardware approach. The work combines advanced hardware design with algorithmic optimization to achieve high efficiency.

My methodology focuses on enhancing time efficiency to adapt the hardware accelerator for environments with limited resources. The objective of My project is to deliver a high-performance solution for local sequence alignment, meeting the increasing need for fast and precise sequence analysis.

Table of Contents

Introduction

Pairwise alignment is a method used to measure similarity between two biological sequences, such as DNA or proteins. It plays a crucial role in understanding genetic variations, evolutionary relationships, and functional annotations. DNA sequences are composed of nucleotides represented by the letters T (thymine), A (adenine), C (cytosine), and G (guanine). Identifying similarities in these sequences provides valuable insights into biological processes and disease mechanisms. Local sequence alignment specifically focuses on aligning regions of interest within larger sequences, making it particularly useful for analyzing complex datasets.

Global and Local Sequence Alignment

Two fundamental methods of sequence alignment are global and local alignment. Global alignment attempts to align sequences in their entirety, maximizing overall similarity. It is most effective when sequences are similar in length and closely related. Local alignment, on the other hand, identifies regions of high similarity within the sequences, making it suitable for analyzing specific functional domains or highly divergent sequences.

For example:

- **Global Alignment:** Comparing entire genomes of related species.
- **Local Alignment:** Identifying functional motifs or conserved regions within genes.

While global alignment provides a comprehensive comparison, local alignment is more flexible and specific, making it indispensable for bioinformatics research involving complex biological data.

Motivation

While aligning short or similar sequences can be done manually, longer, more diverse, or larger sets of sequences often surpass human capabilities. To address these challenges, computational algorithms are utilized. These range from slow but rigorously accurate approaches like dynamic programming to more efficient heuristic and probabilistic methods for large-scale database searches. However, these faster methods may not always guarantee optimal matches. Aligning just a few hundred DNA or protein sequences can take hours on high-performance computing systems. This growing demand for faster processing drives the motivation to develop hardware accelerators capable of handling the increasing volume of data.

Alternative solutions

Solution 1: Manual Alignment
Advantages: No reliance on computational resources, suitable for very short sequences.
Disadvantages: Time-consuming, prone to errors, impractical for long sequences.

Solution 2: Software-Based Dynamic Programming Algorithms
Advantages: Guarantees optimal alignment with high accuracy.
Disadvantages: Inefficient for lengthy or highly variable sequences, requires costly computational resources.

Solution 3: Heuristic and Probabilistic Algorithms
Advantages: Quick and resource-efficient for large datasets.
Disadvantages: Cannot guarantee the best alignment.

Solution 4: Hardware Accelerators
Advantages: Combines speed, efficiency, and accuracy while requiring minimal computing resources.
Disadvantages: Requires specialized hardware for operation.

Given the explosion of biological data in recent years, efficient hardware-based solutions are increasingly critical. This project focuses on implementing a hardware accelerator to address the computational challenges of local sequence alignment.

The Chosen Solution

The project's goal is to develop a hardware accelerator tailored for local sequence alignment using the Smith-Waterman algorithm. Recognized for its accuracy, Smith-Waterman employs dynamic programming to achieve optimal alignment. The hardware accelerator is designed to operate independently, utilizing parallel processing capabilities to achieve high efficiency. This approach minimizes the complexity and overhead associated with software implementations. While the time complexity remains linear ($O(n)$), the space complexity is quadratic ($O(n^2)$), consistent with software implementations. The primary objectives of this project include architectural design, frontend development, verification, and partial backend implementation of the accelerator.

Smith-Waterman Algorithm

The Smith-Waterman algorithm is a dynamic programming approach that identifies optimal local alignments. The algorithm assigns each cell in the substitution matrix (pair of residues) a score based on the scoring system. In order to determine the scoring system we will need to define the following rules:

The match/mismatch score $s(i,j)$ is the similarity score of the two residues related to the cell (i,j). In case the two residues are identical, we will determine the match/mismatch score as $s(i,j)=1$. In any other case, $s(i,j)=-1$.

In addition, gap penalty P is the penalty of a gap in the alignment. The value of the gap penalty is $P=-2$.

Hence, the following definitions we required to define a scoring system:

- **Match:** +1
- **Mismatch:** −1
- **Gap Penalty:** −2

(note, it's possible to define other values for the scoring system for match/mismatch or for gap penalty).

Algorithm Stages:

1. **Initialization:**
   - The first row and column of the scoring matrix are initialized to zero.
2. **Matrix Scoring:**
   - Scores are computed based on matches, mismatches, and gap penalties, selecting the maximum score at each cell, according to the following formula:

$$H_{i,j} = \begin{cases} H_{i-1,j-1} + s(i,j) \\ H_{i-1,j} + P \\ H_{i,j-1} + P \\ 0 \end{cases}$$

3. **Traceback (CIGAR Building):**

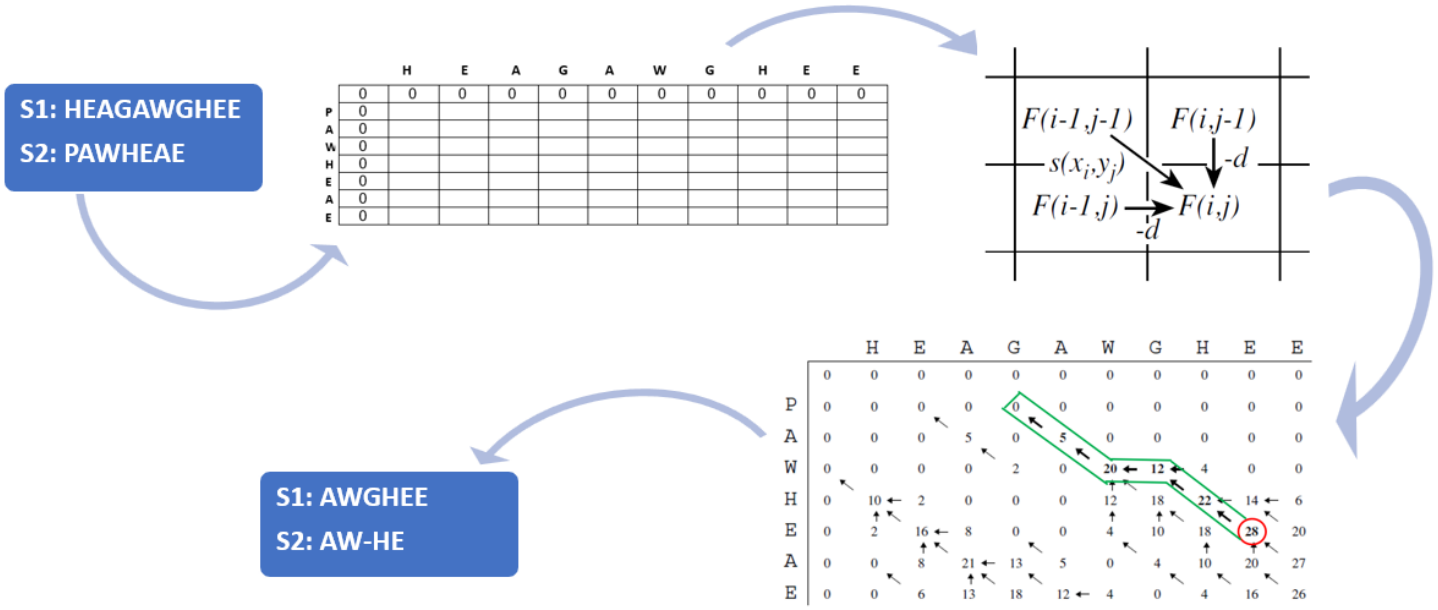
The optimal alignment is identified by tracing back from the cell with the highest score in the matrix to a cell with a score of zero. The best local alignment is constructed in reverse by recursively following the source of each cell. Each traceback step offers three possibilities:

- The source is the diagonally adjacent cell, where both sequences include the residue associated with the cell.
- The source is the cell to the left, where the query sequence includes the residue associated with the cell, while the database sequence has a gap (–).

- The source is the cell above, where the database sequence includes the residue associated with the cell, and the query sequence has a gap (–).

A cell can derive its score from multiple adjacent cells, creating different potential paths during the traceback process. If multiple cells share the highest score, the optimal alignment may not be unique, as all these alignments will have the same similarity score. Following these steps ensures the best possible local alignment between the two sequences.

Smith Waterman - General Example



Detaild Example – Step by Step:

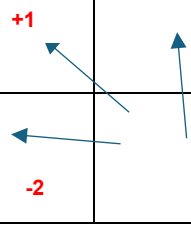
Consider aligning the sequences "GCATAGGT" and "AGCAAGCT":

1. **Initialization:**

		G	C	A	T	A	G	G	T
	0	0	0	0	0	0	0	0	0
A	0								
G	0								
C	0								
A	0								
A	0								
G	0								
C	0								
T	0								

2. **Matrix Scoring:**
   - Compute scores for each cell based on the scoring system: in every cell we will calculate the maximum score according to the formula above and according to the letter of the cell's raw and column. For example:

		G	C
	0	0	0
A	0	0	0
G	0	1	0



Fill like this the entire table (matrix)

		G	C	A	T	A	G	G	T
	0	0	0	0	0	0	0	0	0
A	0	0	0	1	0	1	0	0	0
G	0	1	0	0	0	0	2	1	0
C	0	0	2	0	0	0	0	1	0
A	0	0	0	3	1	1	0	0	0
A	0	0	0	1	2	2	0	0	0
G	0	1	0	0	0	1	3	1	0
C	0	0	2	0	0	0	1	2	0
T	0	0	0	1	1	0	0	0	3

- The arrow will be to the cell which brought us to the maximum score:

		G	C	A	T	A	G	G	T
	0	0	0	0	0	0	0	0	0
A	0	0	0	1	0	1	0	0	0
G	0	1	0	0	0	0	2	1	0
C	0	0	2	0	0	0	0	1	0
A	0	0	0	3	1	1	0	0	0
A	0	0	0	1	2	2	0	0	0
G	0	1	0	0	0	1	3	1	0
C	0	0	2	0	0	0	1	2	0
T	0	0	0	1	1	0	0	0	3

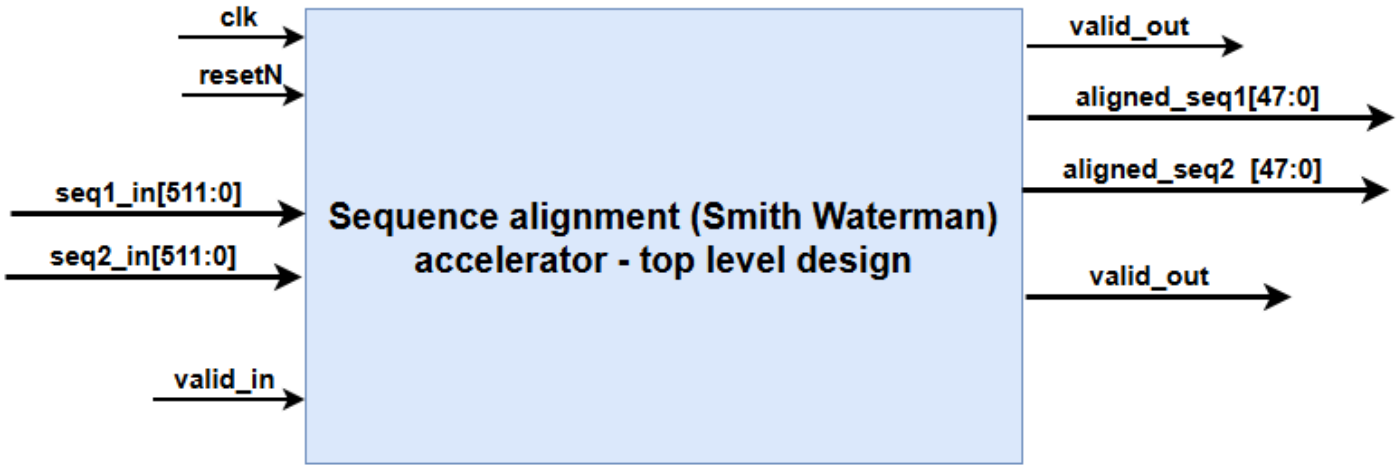
3. **Traceback:**
   - Trace back from the highest score to reconstruct the alignment: we got 3 different optimal alignments:

Alignment 1	Alignment 2	Alignment 3
GCA	GCATAG	GCATAGGT
GCA	GCA-AG	GCA-AGCT

Top level architecture

This document describes a high-performance hardware accelerator implementing the Smith-Waterman algorithm for local sequence alignment. The design focuses on maximizing throughput through parallel processing and efficient resource utilization. The accelerator processes two DNA sequences of 256 characters each, employing a scoring scheme where matches score +1, mismatches -1, and gaps -2.

Top level design interface



Signal/Vector/Matrix name	Input/output	Size (bits)	Description
Clk	Input	1	System clock
resetN	Input	1	Active low reset
valid_in	Input	1	Input data valid
seq1_in	Input	256 chars × 2 bits	First sequence
seq2_in	Input	256 chars × 2 bits	Second sequence
valid_out	output	1	Output data valid
aligned_seq1	output	16 chars × 3 bits	First aligned subsequence
aligned_seq2	output	16 chars × 3 bits	Second aligned subsequence

Input/Output Chars Encoding

The interface includes 2 input sequences of nucleotide letters. Each letter will be represented by 2 bits which will be encoded in hardware as follows:

A: 2'b00

T: 2'b01

C: 2'b10

G: 2'b11

The interface includes also 2 output aligned sequences which will be consisted of nucleotide letters or a gap. Each letter or gap will be represented in the output by 3 bits which will be encoded as follows:

A: 3'b000

T: 3'b001

C: 3'b010

G: 3'b011

Gap: 3'b100

Sequence Alignment Accelerator – Micro Architecture

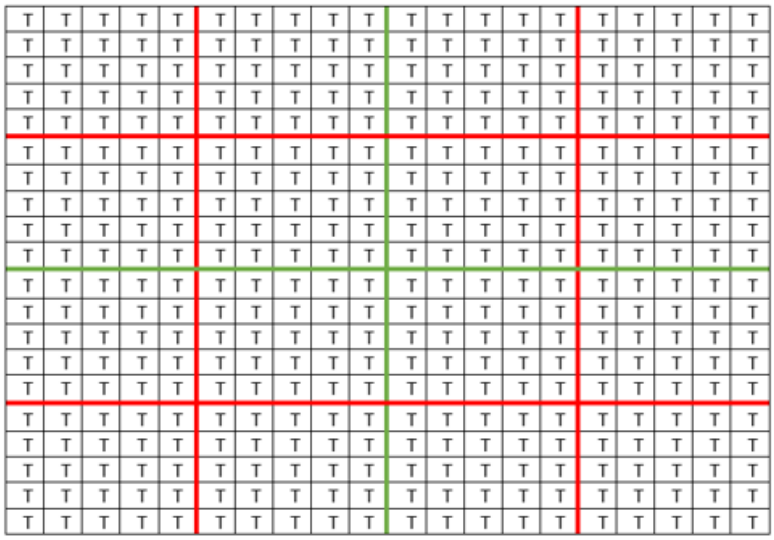
The architecture of the accelerator consists of five major interconnected components, each playing a vital role in the overall functionality of the system. These components are:

1. **Scheduler**: Manages sequence division, distribution, and orchestrates parallel processing across multiple solver units.
2. **Solver Units (16 instances)**: Perform the core computations of the Smith-Waterman algorithm on 16×16 tiles of the scoring matrix.
3. **Organizer Block**: Collects and stores results, manages tile-based processing, and coordinates the traceback process.
4. **RAM Block**: Stores intermediate results such as arrow matrices, utilizing an efficient position-based addressing scheme.
5. **CIGAR Builder**: Constructs the final sequence alignment by performing a traceback operation on the computed arrow matrices.

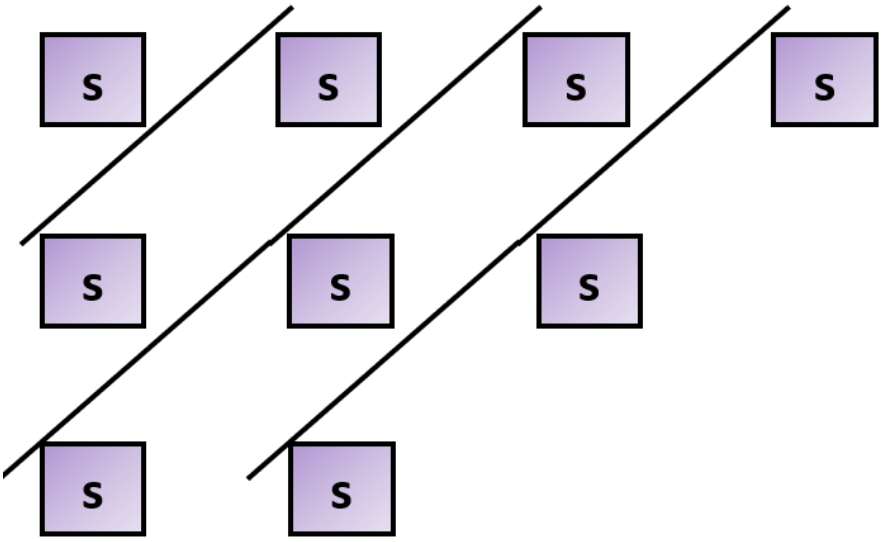
Parallel Processing Strategy

The design implements multiple levels of parallelism:

1. Tile-Level Parallelism:
    - 256×256 matrix divided into 16×16 tiles:



    - 16 Solver units operating simultaneously:



- Diagonal wavefront processing pattern

2. Within-Tile Parallelism:
    - Each Solver processes its tile using diagonal wavefront pattern
    - Internal parallel computation of scoring matrix cells
    - Concurrent arrow direction determination
3. System-Level Pipelining:
    - Scheduler: Preparing next tile data
    - Solvers: Computing current tiles
    - Organizer: Storing completed tile results
    - CIGAR Builder: Constructing alignment from completed tiles
4. Memory Access Optimization:
    - Efficient RAM storage scheme for arrow matrices
    - Parallel access to border scores
    - Streamlined data flow between components

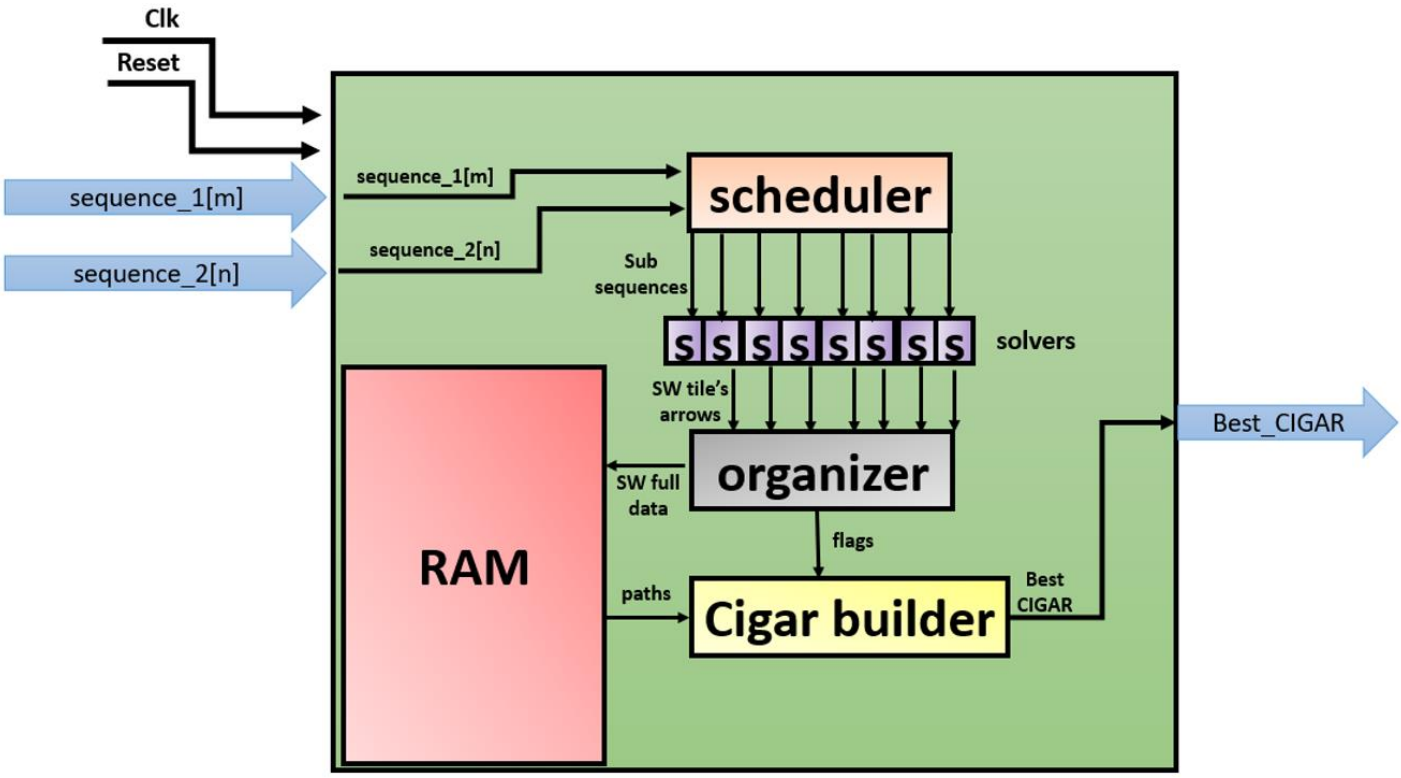
This multi-level parallel architecture achieves:

- High throughput processing of large sequences
- Efficient resource utilization
- Balanced workload distribution
- Optimized data dependencies management

Processing Flow

1. Initial Sequence Input:
    - The accelerator receives two complete 256-character sequences (512 bits each)
    - The Scheduler stores these sequences for systematic distribution
2. Tile Distribution:
    - The Scheduler divides the 256×256 scoring matrix into 16×16 tiles
    - Tiles are processed in diagonal wavefronts to maintain data dependencies
    - Starting from the upper-left corner tile, moving diagonally
3. Parallel Computation:
    - The 16 Solver units process different tiles concurrently
    - Each Solver computes:
        - Smith-Waterman scoring matrix for its 16×16 tile
        - Direction (arrow) matrix
        - Maximum score and its position within the tile
        - Border scores (last row and column) for neighboring tiles
4. Result Collection:
    - The Organizer:
        - Collects arrow matrices from Solvers
        - Stores them efficiently in RAM based on tile position
        - Tracks global maximum score and its position
        - Manages data transfer between Solvers and RAM
5. Alignment Construction:
    - Upon completion of all tiles:
        - CIGAR Builder receives the arrow matrix containing the global maximum
        - Performs traceback across tiles
        - Constructs aligned sequences in 16-character segments
        - Requests next tiles from Organizer as needed

Sub Blocks diagram – functionality and details



Scheduler (Sequence Management Unit)

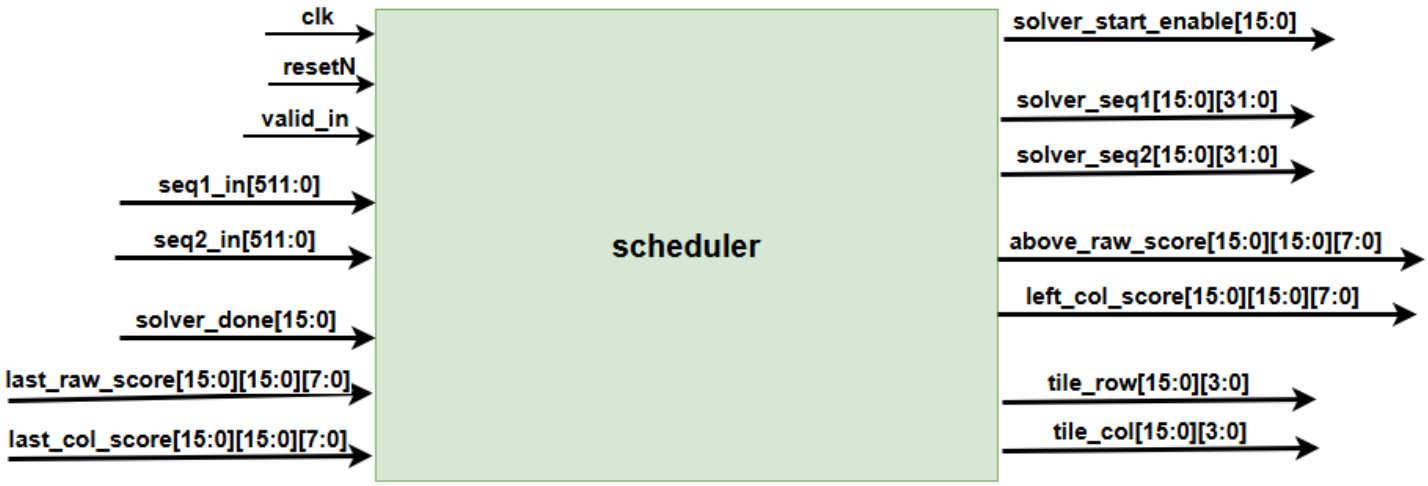
Functional Purpose

The Scheduler is responsible for managing the decomposition of the input sequences into tiles and distributing these tiles across 16 solver units. It operates in tandem with the Organizer and Solver units to maintain a seamless processing flow.

Detailed Operation

1. **Input Handling**:
   - Accepts two 256-character sequences (512 bits each).
   - Stores sequences for systematic processing.
2. **Tile Division**:
   - Divides the 256×256 matrix into 16×16 tiles.
   - Assigns tiles to Solver units following a diagonal wavefront pattern, starting from the upper-left corner.
3. **Distribution**:
   - Ensures that each solver receives the required subsequences for processing.

Scheduler interface



Signal/Vector/Matrix name	Input/output	Size (bits)	Description
Clk	input	1	System clock
resetN	input	1	Active low reset
valid_in	input	1	Input data valid
seq1_in	input	256 chars × 2 bits	First sequence
seq2_in	input	256 chars × 2 bits	Second sequence
solver_done	input	16 solvers x 1 bit per solver	Indication from the solver that its done

last_row_score	input	16 solvers x 16 elements in row x 8 bit per element	Score of the last row of the tile from the solvers
last_col_score	input	16 solvers x 16 elements in col x 8 bit per element	Score of the last col of the tile from the solvers
solver_start_enable	output	16 solvers x 1 bit per solver	Permission for the solver to start
solver_seq1	output	16 chars × 2 bits per solver x 16 solvers	First subsequence for each solver
solver_seq2	output	16 chars × 2 bits per solver x 16 solvers	Second subsequence for each solver
above_row_score	output	16 solvers x 16 elements in row x 8 bit per element	Score of the row above the tile for each solver
left_col_score	output	16 solvers x 16 elements in col x 8 bit per element	Score of the left col above the tile for each solver
tile_row	output	16 solvers x 4 bits to represent tile row	Row index of the tile in the total matrix
tile_col	output	16 solvers x 4 bits to represent tile row	col index of the tile in the total matrix

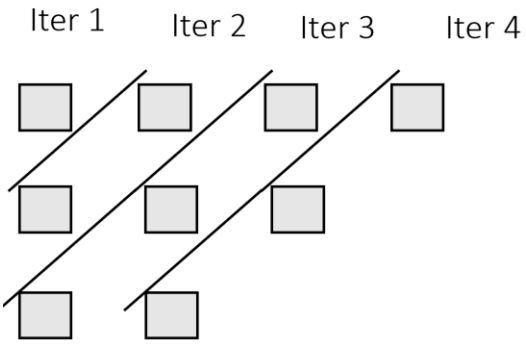
Solver Units (Parallel Processing Elements)

Functional Purpose

The Solver units perform the core computation of the Smith-Waterman algorithm on 16×16 tiles. Each solver operates independently on its assigned tile, computing both the scoring matrix and the arrow matrix.

Internal Workflow

1. **Initialization**:
    - Receives subsequences and initializes local scoring and arrow matrices.
2. **Diagonal Wavefront Processing**:
    - Processes cells in a diagonal pattern, starting from the upper-left corner.
    - Each diagonal within the tile is computed in parallel using specialized processing elements.



3. **Output**:
    - Generates a 512-bit arrow matrix representing traceback directions. Each arrow (or blank) will be represented by 2 bits as follows:

      A: 3'b000

      T: 3'b001

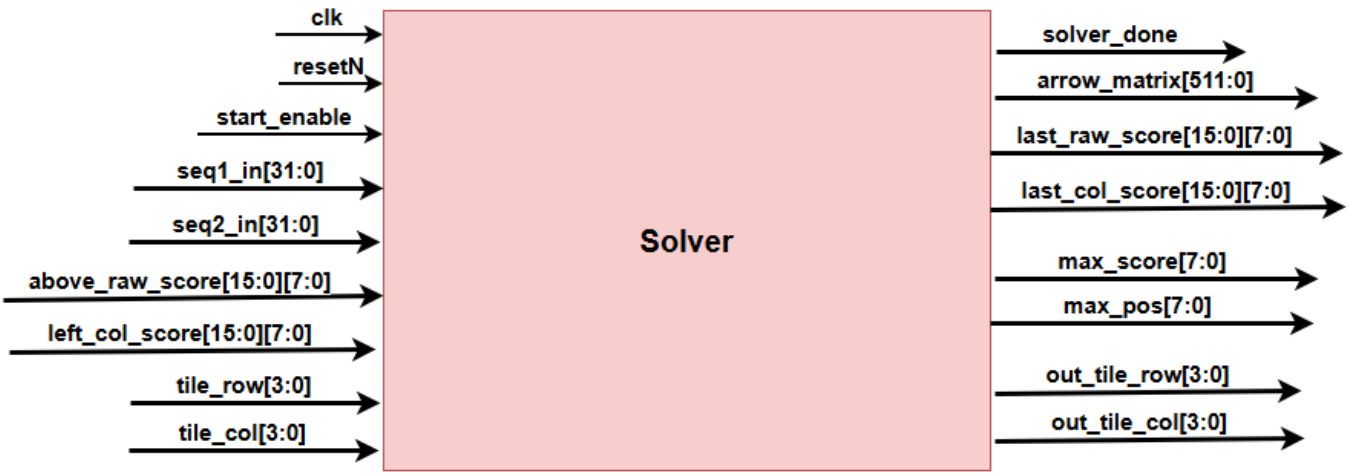
      C: 3'b010

      G: 3'b011

      Gap: 3'b100

    - Reports the maximum score, its position, and the border scores for neighboring tiles.

Solver interface



Signal/Vector/Matrix name	Input/output	Size (bits)	Description
Clk	input	1	System clock
resetN	input	1	Active low reset
start_enable	input	1	Permission for the solver to start
seq1_in	input	16 chars × 2 bits	First subsequence
seq2_in	input	16 chars × 2 bits	Second subsequence
above_row_score	input	16 elements in row x 8 bit per element	Score of the row above the tile
left_col_score	input	16 elements in col x 8 bit per element	Score of the left col above the tile
tile_row	input	4 bits to represent tile row	Row index of the tile in the total matrix from scheduler
tile_col	input	4 bits to represent tile col	col index of the tile in the total matrix from scheduler
solver_done	output	1 bit per solver	Indication from the solver that its done
last_row_score	output	16 elements in row x 8 bit per element	Score of the last row of the tile to scheduler
last_col_score	output	16 elements in col x 8 bit per element	Score of the last col of the tile to scheduler
max_score	output	8 bits	Highest score in tile

max_pos	output	8 bits	Location of the highest score in tile
out_tile_row	output	4 bits to represent tile row	Row index of the tile in the total matrix to organizer
out_tile_col	output	4 bits to represent tile row	col index of the tile in the total matrix to organizer

Organizer (Result Management Unit)

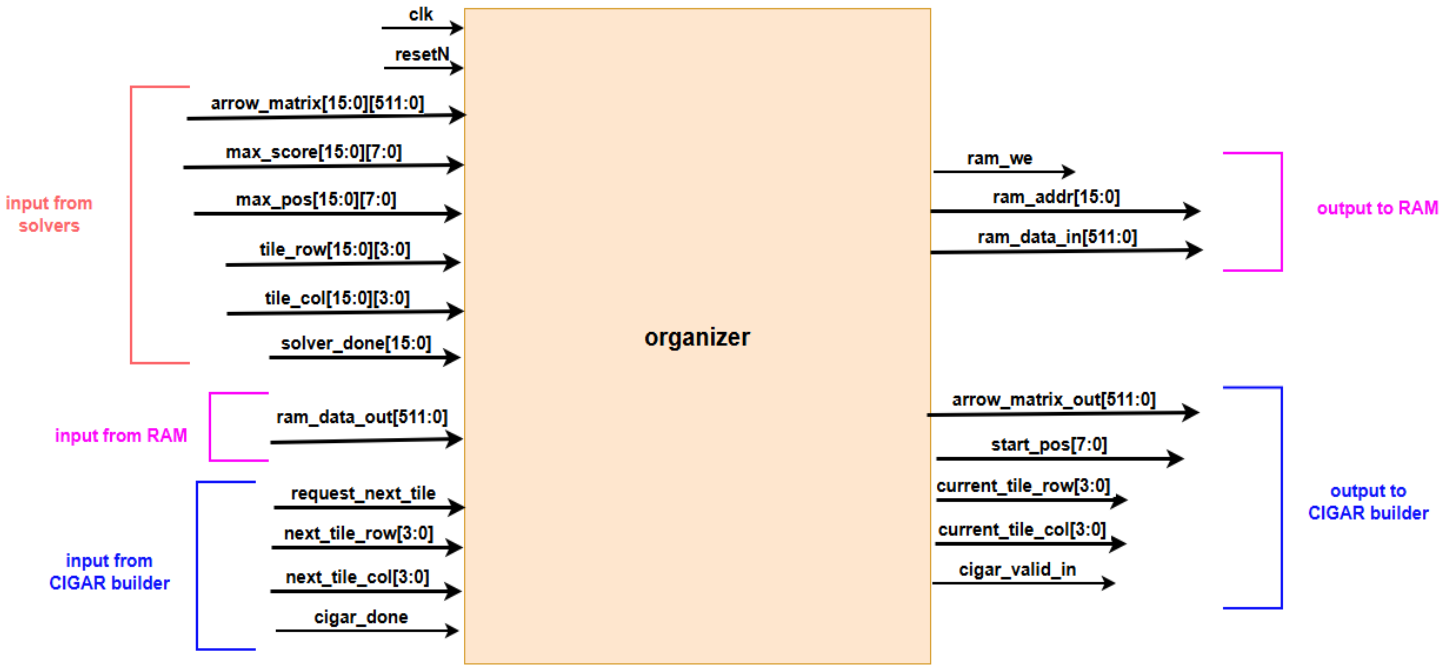
Functional Purpose

The Organizer oversees the collection and storage of results from Solver units, coordinates traceback operations, and manages intermediate data in RAM.

Traceback Coordination

1. **Initialization**:
   - Identifies the tile containing the global maximum score.
   - Retrieves the corresponding arrow matrix, start position, and tile position.
2. **Tile Handoff**:
   - Provides tile data to the CIGAR Builder.
   - Asserts control signals for traceback operations.
3. **RAM Management**:
   - Stores arrow matrices using position-based addressing.
   - Ensures quick and efficient data retrieval.

Organizer interface



Signal/Vector/Matrix name	Input/output	Size (bits)	Description
Clk	input	1	System clock
resetN	input	1	Active low reset
Interface with Solvers (×16)			
arrow_matrix	input	2 bits for arrow x 256 (sizeof tile = 16 x 16) x 16 solvers	Arrow matrices from solvers

max_score	input	8 bits score x 16 solvers	Maximum scores from solvers
max_pos	input	8 bits index x 16 solvers	Position of max score in each tile
tile_row	input	4 bits to represent tile row x 16 solvers	Tile row position in the large matrix
tile_col	input	4 bits to represent tile col x 16 solvers	Tile col position in the large matrix
solver_done	input	1 bit per solver x 16 solvers	Completion signals from solvers
RAM interface			
ram_we	output	1	Write enable
ram_addr	output	16 bits	Address for storing/reading tiles
ram_data_in	output	512 (size of arrow tile)	Arrow matrix to store
ram_data_out	input	512 (size of arrow tile)	Arrow matrix from RAM
CIGAR Builder interface			
arrow_matrix_out	output	2 bits for arrow x 256 (sizeof tile = 16 x 16)	Current tile's arrow matrix
start_pos	output	8 bits index	Initial position in first tile
current_tile_row	output	4 bits to represent tile row	Tile row position in the large matrix
current_tile_col	output	4 bits to represent tile col	Tile col position in the large matrix
cigar_valid_in	output	1	Arrow matrix is valid
request_next_tile	input	1	CIGAR builder needs next tile
next_tile_row	input	4 bits to represent tile row	Row of the next requested tile position
next_tile_col	input	4 bits to represent tile col	col of the next requested tile position
cigar_done	input	1	CIGAR builder finished alignment

RAM

Functional Purpose

The RAM block stores arrow matrices and facilitates rapid access to intermediate results during the traceback process. The addressing scheme is based on the tile's row and column position, eliminating the need for complex search operations.

Architecture

1. Position-indexed storage structure
2. Optimized addressing scheme
3. High-speed access capabilities

Features

1. Storage Organization
    - Tile-based data structuring
    - Position-correlated addressing
    - Efficient access patterns
2. Access Management
    - Multiple port access support
    - Read/write arbitration
    - Access prioritization

RAM interface

CIGAR builder

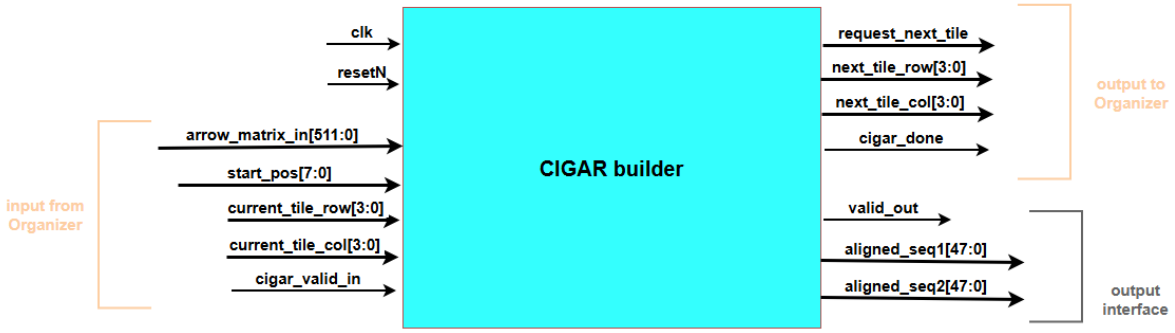
Functional Purpose

The CIGAR Builder constructs the final sequence alignment by performing a traceback operation on the stored arrow matrices.

Detailed Traceback Process

1. **Initialization**:
   - Receives the arrow matrix containing the global maximum score.
   - Retrieves start position and tile position.
2. **Within-Tile Traceback**:
   - Processes arrows sequentially to reconstruct alignment.
   - Handles diagonal, up, and left directions appropriately.
3. **Inter-Tile Transition**:
   - Calculates the next tile position when a boundary is reached.
   - Requests new tile data from the Organizer.
4. **Completion**:
   - Detects alignment completion when encountering a 00 arrow or matrix boundary.
   - Asserts a done signal to terminate processing.

CIGAR builder interface

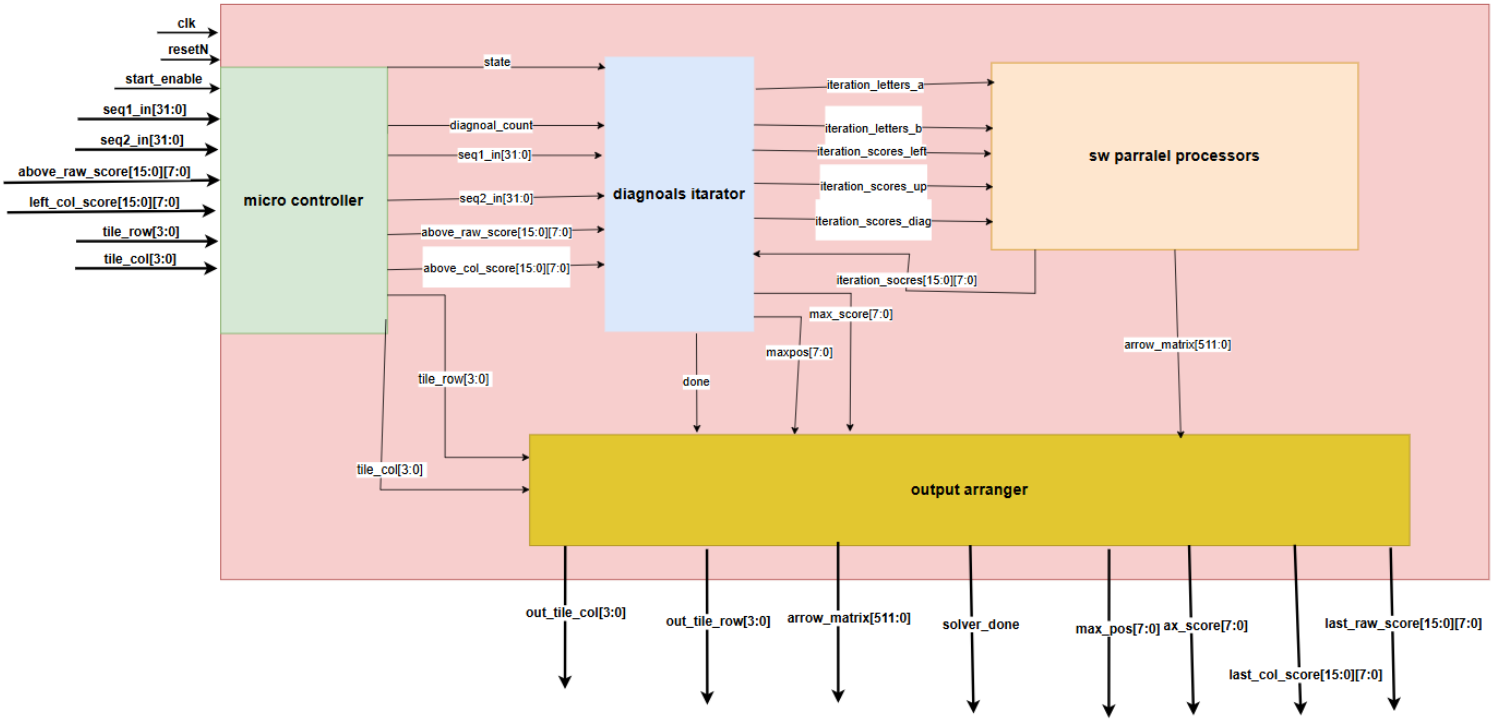


Signal/Vector/Matrix name	Input/output	Size (bits)	Description
Clk	input	1	System clock
resetN	input	1	Active low reset
Organizer interface			
arrow_matrix_in	input	2 bits for arrow x 256 (sizeof tile = 16 x 16)	Current tile's arrow matrix
start_pos	input	8 bits index	Initial position in first tile
current_tile_row	input	4 bits to represent tile row	Tile row position in the large matrix

current_tile_col	input	4 bits to represent tile col	Tile col position in the large matrix
cigar_valid_in	input	1	Arrow matrix is valid
request_next_tile	output	1	CIGAR builder needs next tile
next_tile_row	output	4 bits to represent tile row	Row of the next requested tile position
next_tile_col	output	4 bits to represent tile col	col of the next requested tile position
cigar_done	output	1	CIGAR builder finished alignment
Output interface			
valid_out	output	1	Output data valid
aligned_seq1	output	16 chars × 3 bits	Aligned sub seq 1
aligned_seq2	output	16 chars × 3 bits	Aligned sub seq 2

Components For Implementation

Solver



Internal components

1. Micro controller

- Responsibility for synchronization according to state machine and for pipelining the information

2. Diagnoals_itarator

- Responsibility for dividing the information to secondary diagnoals which will work sequentially. Also responsible to locate maximum score for tile

3. SW parallel processors

- Responsibility for operating the Smith Waterman formula for scoring and determining arrow. There are 16 processors that will work parallely – each one of them will work on a single cell of the relevant iteration. Not all of them will be busy in every iteration (depends on the size of the diagonal).

4. Output arranger

- Responsibility for arranging the relevant output signals and vectors – control output, score output and arrows output – some of the outputs are going to the scheduler and some of them to organizer.

Data Structures

- score_matrix: Stores current scores (8 bits per cell)
- temp_arrow_matrix: Stores direction arrows (2 bits per cell)
- seq1_chars/seq2_chars: Input sequence storage

Processing Algorithm

Initialization Phase

1. Load border scores from inputs
2. Reset counters and control signals
3. Prepare for diagonal processing

Diagonal Processing

1. Process cells along each diagonal in parallel
2. For each cell:
    - Calculate match/mismatch score
    - Compute scores for all three directions
    - Select maximum score and corresponding direction
    - Update matrices and track maximum score

Score Calculation

- Match: +1
- Mismatch: -1
- Gap: -2
- No negative scores allowed

4. Output Generation

- Pack arrow matrix into 512-bit output
- Store last row and column scores
- Record maximum score and position
- Signal completion

Implementation Details

1. Timing Considerations

- One clock cycle per diagonal
- Total processing time: 31 cycles plus initialization/finalization

2. Resource Usage

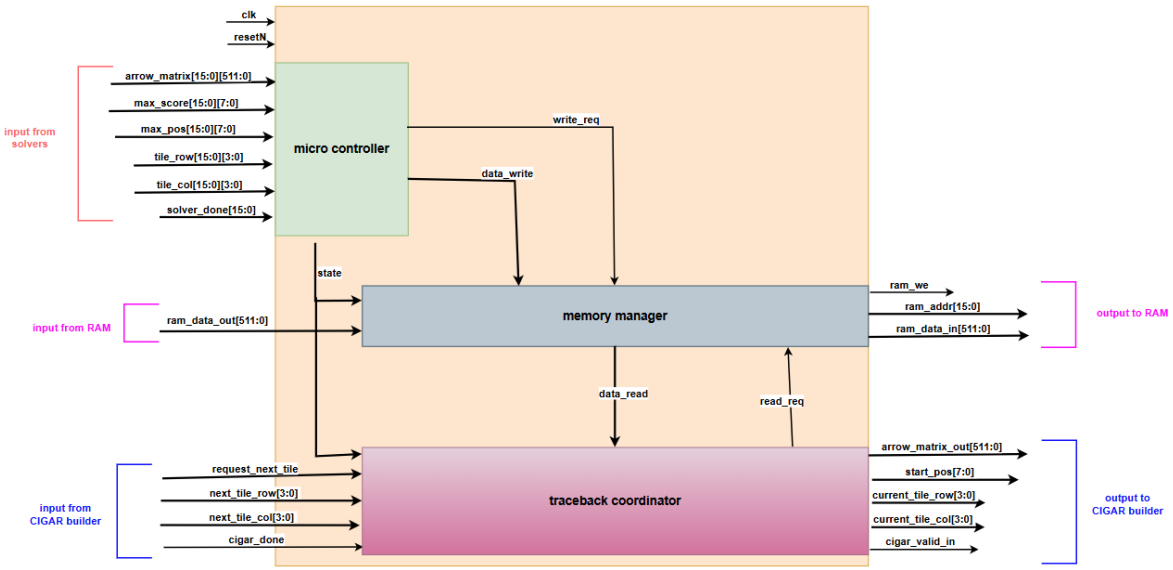
- Memory: 2 16×16 matrices
- Combinational logic for score calculation

- Sequential logic for state machine

3. Critical Paths

- Score calculation and comparison logic
- Diagonal cell processing logic

Organizer



Internal components

1. Micro controller

- Responsibility for synchronization according to state machine and for pipelining the information

2. Memory Manager

- Monitor solver completion signals
- Round-robin processing of completed solvers
- Track global maximum score and position
- Store arrow matrices in RAM

3. Traceback Coordinator

- Initial setup with maximum score tile
- Handle tile requests from CIGAR Builder
- RAM data retrieval and forwarding
- Position tracking and validation

Implementation Details

1. Control Signals

- Solver completion tracking
- RAM write enable management
- CIGAR Builder handshaking

2. Data Flow

- Solver → Organizer → RAM storage
- RAM → CIGAR Builder (during traceback)

3. Critical Features

- No loss of solver results
- Proper maximum score tracking
- Efficient RAM utilization
- Reliable traceback support

Bibliography

1. Smith, T. F., & Waterman, M. S. (1981). “Identification of common molecular subsequences.” Journal of Molecular Biology, 147(1), 195-197. doi:10.1016/0022-2836(81)90087-5
2. Altschul, S. F., et al. (1990). “Basic local alignment search tool”. Journal of Molecular Biology, 215(3), 403-410. doi:10.1016/S0022-2836(05)80360-2
3. Durbin, R., Eddy, S. R., Krogh, A., & Mitchison, G. (1998). Biological sequence analysis: Probabilistic models of proteins and nucleic acids. Cambridge University Press.
4. Mount, D. W. (2004). Bioinformatics: Sequence and genome analysis. Cold Spring Harbor Laboratory Press.
5. DeCypher, B. A. (2008). Accelerating bioinformatics applications using hardware-based solutions. IEEE Transactions on Bioinformatics, 5(3), 153-162.
6. Aluru, S. (2006). "Handbook of Computational Molecular Biology." Chapman & Hall/CRC Computer and Information Science Series.
7. Jiang, X., Smith, R. (2017). "FPGA-based Local Sequence Alignment Accelerators: A Systematic Review." ACM Computing Surveys.
8. Kumar, P., et al. (2019). "Hardware Acceleration of Sequence Alignment Algorithms - A Survey." IEEE Design & Test.
9. Wang, L., Jiang, T. (2019). "On the Complexity of Multiple Sequence Alignment." Journal of Computational Biology.