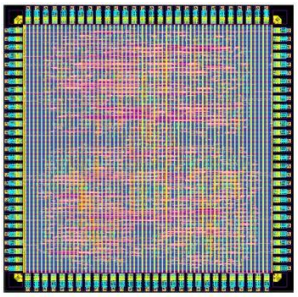
Technion – Israel Institute of Technology

The Andrew and Erna Viterbi Faculty of Electrical & Computer Engineering

VLSI Laboratory

Semesters Winter 2023-2024, Spring 2024

Hardware Accelerator for Genetic Local Sequence Alignment



Authors:

Niv Bar-Tov, 206133829

Dor Lerman, 209400761

Liran Lavi, 207501354

Supervised by: Goel Samuel , Pavel Gushpan

This project was done in collaboration with Apple Israel.

Abstract

The local sequence alignment problem, where two sequences are compared to identify similarities between them, is a fundamental tool in bioinformatics, genomics, proteomics, and other related fields. The Smith-Waterman algorithm is a dynamic programming algorithm which solves this problem. However, its computational complexity poses a significant challenge when dealing with large genomic databases. This project focuses on the design and implementation stages of planning a dedicated hardware accelerator for the Smith-Waterman algorithm, aimed at significantly improving its execution speed from a quadratic complexity to linear complexity.

The proposed hardware accelerator leverages parallel processing and custom hardware architecture to accelerate sequence alignment tasks. We discuss the architectural plan, hardware design, logical simulations, physical implementation, and alternative solutions.

Our approach emphasizes optimizing time efficiency to make the hardware accelerator suitable for resource-constrained environments. Our project's target is to provide a high-performance solution for local sequence alignment, addressing the growing demand for rapid and accurate sequences analysis.

Table of Contents

List of Figures

List of Tables

SECTION A:
Introduction

1 Introduction

Numerous fields presently necessitate sequence database searches that rely on pairwise alignment. This involves comparing a given query sequence with a database of sequences to pinpoint the sequence with the highest degree of similarity. In the realm of bioinformatics, a sequence alignment is a way of arranging sequences of DNA, RNA, or proteins to identify regions of similarity that may indicate on a relationship between the sequences. Such similarities can yield valuable insights into the function of the query protein or a gene. Furthermore, this method is also beneficial for alignment of non-biological sequences.

In the case of DNA sequences, the residues are nucleotides that each one of them can be one of the four nucleobases: adenine, guanine, cytosine, and thymine. These nucleotides will be called throughout this project as A, G, C, and T, accordingly.

1.1 Global and Local Alignment

Computational approaches to sequence alignment generally fall into two categories: global alignment and local alignment. Performing a global alignment constitutes a type of global optimization that ensures the alignment extends across the entire length of the sequences. On the other hand, local alignment pinpoint regions of similarity within sequences that are often widely divergent overall. While local alignments are frequently favored, they can pose greater computational complexity due to the added task of identifying these similarity regions. Table 1 summarizes the differences between these two approaches.

Global Alignment	Local Alignment
Align the entire sequence	Align regions having highest similarities
Suitable for closely related sequences	Suitable for more divergent sequences
Main well-known algorithm: Needleman-Wunsch algorithm	Main well-known algorithm: Smith-Waterman algorithm

Table 1: Differences Between Global and Local Alignment

1.2 Motivation and Alternative Solutions

While short or similar sequences can be manually aligned, the alignment of longer, highly diverse, or numerous sequences often exceeds human capabilities. To tackle such challenges, computational algorithms come into play. These include slow but formally correct methods like dynamic programming. Aligning just a few hundred DNA or protein sequences can demand several CPU hours on high-performance computers. Alternatively, there exist efficient heuristic algorithms and probabilistic methods tailored for large-scale database searches, although they do not guarantee to find the optimal matches.[1][2] Hence, there is a high motivation to implement hardware accelerators to keep up with the increasing amount of data.

Solution	Advantages	Disadvantages
Alignment by hand	- No need of computing resources - Efficient for very short sequences	- Slow method - High potential of mistakes - Impossible to do on long sequences
Software implementation of dynamic programming algorithms	- Accuracy: guarantee to find the optimal alignment	- Slow method for very long or highly variable sequences - Requires expensive computing resources
Heuristic algorithms and probabilistic methods	- Fast and efficient method - Consumes few computing resources	- No guarantee to find the optimal alignment
Hardware accelerator implementation	- Fast and efficient method - Consumes few computing resources - Accuracy: guarantee to find the optimal alignment	- Need of dedicated hardware to perform the alignment

Table 2: Comparison of Alternative Solutions

1.3 The Chosen Solution

We decided to implement in this project a hardware accelerator that will solve the local sequence alignment problem using Smith-Waterman algorithm, which is one of the well-known algorithms for solving this problem by dynamic programming. The accelerator will be separate from other expensive computing resources and will solve the problem more efficiently because the computation will be done in parallel and directly on the hardware, instead of complex programming implementations. The hardware implementation is done in linear time complexity $O(n)$, while the best software accurate implementation is done in quadratic time complexity $O(n^2)$. The space complexity is quadratic $O(n^2)$, like the software implementations.

The goal of this project is to perform the architectural plan, frontend design, verification stages, and backend design of the hardware accelerator that implements Smith-Waterman algorithm for local sequence alignment.s

2 Smith-Waterman Algorithm

Smith-Waterman algorithm is a dynamic programming algorithm used for local sequence alignment.[3] It compares segments of all possible lengths and optimizes the similarity measure, using a substitution matrix. Its time and space complexity of the software implementation are quadratic.

2.1 Scoring System

The algorithm assigns each cell in the substitution matrix (pair of residues) a score based on the scoring system. The following definitions are required to define a scoring system:

- 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, the match/mismatch score is $s(i,j) = 1$. In any other case, $s(i,j) = -1$.
- **Gap penalty** P is the penalty of a gap in the alignment. The value of the gap penalty is $P = -2$.

It is possible to define another scoring system with different values for the match/mismatch score and the gap penalty. We defined the scoring system described below for the implementation of the accelerator.

Score	Value
Match	1
Mismatch	-1
Gap Penalty	-2

Table 3: Chosen Scoring System

2.2 Main Algorithm Stages

Let $Q = q_1 q_2 \dots q_N$ be the query sequence to be aligned with $D = d_1 d_2 \dots d_M$, the database sequence. The Smith-Waterman Algorithm consists of 3 main stages:

1. Initialization: Construct a substitution matrix H and initialize its first row and column with zeros. The dimensions of the matrix are $(M+1) * (N+1)$.
2. Scoring: Fill the substitution matrix and calculate the score for each cell based on the scoring system using the following equation:

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

3. Traceback: Determine the optimal alignment by trace back from the highest-scoring cell in the matrix to a cell with a score of zero. We will generate the best local alignment in reverse direction by tracing back recursively based on the source of each cell. There are 3 different options for each traceback step:
   - Source is the diagonal adjacent cell – both aligned sequences contain the residue related to the cell.

- Source is the left adjacent cell – the aligned query sequence contains the query residue related to the cell and the database sequence contains a gap (–).
- Source is the top adjacent cell – the aligned database sequence contains the database residue related to the cell and the query sequence contains a gap (–).

A cell can receive a score from more than one adjacent cell, each will form a different path if this cell is traced back. In case of more than one cell with the highest score, the optimal alignment is not unique (all these alignments have the same similarity measure).

Using this algorithm stages ensures the optimal local alignment of the two sequences.

2.3 Example

Let TACGCTTG be the query sequence and CTACCTAG be the database sequence. The initialized substitution matrix appears in figure 1.

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

Figure 1: Initialization of the Substitution Matrix

According to the scoring system, cell $(1,1)$ has a mismatch and $s(1,1) = -1$. Thus, the cell's score is 0. Figures 2 and 3 show the calculation process of the substitution matrix.

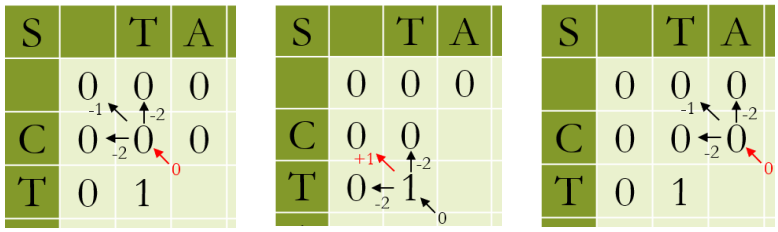


Figure 2: Calculation of 3 Cells in the Substitution Matrix

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

Figure 3: Calculation of the Substitution Matrix

After filling the entire matrix, apply the traceback process and generate the optimal alignment. There are 3 different optimal routes, shown in figure 4.

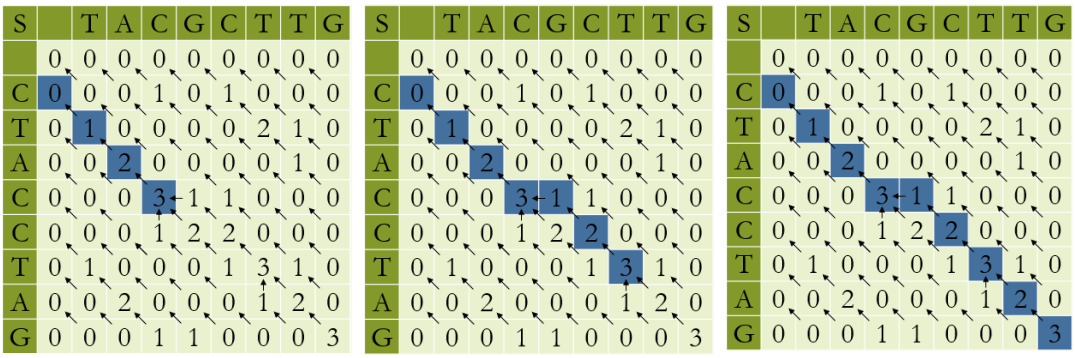


Figure 4: Traceback Possible Routes

Each one of these routes generates a different optimal alignment:

TAC	TACGCT	TACGCTTG
TAC	TAC–CT	TAC–CTAG
(a)	(b)	(c)

3 Goals, Requirements and Specifications

The project has several requirements and specifications we defined at the beginning of the work:

- The design will be implemented based on Smith-Waterman Algorithm.
- Each query and database sequence will contain 32 residues exactly (no support for alignment of sequences with different lengths).
- The system will not support alignment of more than two sequences in parallel.
- The scoring system will be predetermined and unchangeable (it will be part of the design).
- The system will output one possible pair of aligned sequences in case of more than one optimal route. It will output the similarity score as well.
- The accelerator will be designed in a multi-cycle architecture: each pair of sequences will be aligned in more than one cycle, without pipeline.
- The accelerator will be implemented using SystemVerilog hardware description language.
- The accelerator will be fabricated in TSMC with a technology process of 65 nm.
- The chip uses a single clock. The target clock frequency is 100 MHz.
- Aim to achieve maximum area efficiency, with area target of 1.0x1.0 sq. mm.

There are many ways to optimize the chip or make it more general and configurable according to the Smith-Waterman algorithm. For example, it could be possible to design the accelerator in a pipelined architecture to increase the system's throughput. It is also possible to support variable scoring system or alignment of sequences with unequal lengths. However, we had to simplify the project and to meet the time frame of the project work. Timing, power, and area constraints were also considered.

SECTION B:
Frontend Design

4 Top Level Architecture

4.1 General Terms

Throughout this project, we will use the following terms and definitions:

Term	Description
Sequence	An input sequence consists of 32 letters. A letter can be A,T,C or G.
Aligned Sequence	An output aligned sequence consists of 32 letters or a gap.
Query Sequence	The top Sequence in the matrix.
Database Sequence	The left Sequence in the matrix.
Processing Element (PE)	The basic computation unit of the system, performs computation of a single cell in the substitution matrix.
Processing Unit (PU)	Consists of 4 PEs, which are arranged in a 2x2 matrix.
Data Packet	Information regarding a single cell in the substitution matrix. The information includes the cell's adjacent source and a zero-score bit.

Table 4: General Terms and Definitions

4.2 Top Level Design Interface

The accelerator receives two sequences as an input, each one of them consists of 32 letters, 4 letters from each sequence are received per cycle. The accelerator will compute the best possible alignment of the sequences and will output the aligned sequences, one letter from each sequence per cycle (in a reverse order), as well as their similarity score.

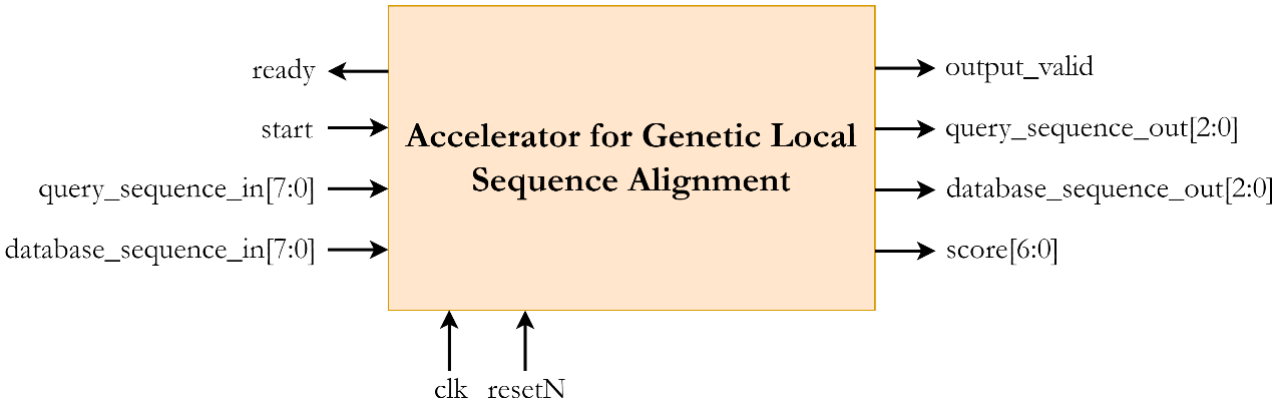


Figure 5: I/O Diagram

Signal Name	In/Out	Size (bits)	Description
clk	input	1	System's clock.
rst_n		1	Reset signal, active on low phase.
query_sequence_in		4x2	Sub-sequence of 4 letters of the query sequence.
database_sequence_in		4x2	Sub-sequence of 4 letters of the database sequence.
start		1	Single pulse signal which indicates the start of sequences loading. It must be active one cycle before loading the sequences.
ready	output	1	System is ready to get a new pair of sequences as an input.
query_seq_out		3	One letter of the aligned query sequence.
database_seq_out		3	One letter of the aligned database sequence.
score		7	The score of the optimal sequence alignment.
output_valid		1	Indicates the output is valid and ready for read.

Table 5: Top Level I/O Signals

4.3 Block Diagram

The top-level unit consists of the following sub-units:

1. **Sequences Buffer** – stores the inserted sequences.
2. **Matrix Calculation** – responsible for filling the substitution matrix by calculating the score of each cell.
3. **Matrix Memory** – stores the information regarding each cell in the substitution matrix. The Matrix Memory is based on registers.
4. **Max Registers** – responsible for computing the maximum score and index (position) in the substitution matrix. It is updated after every cycle of computation, if necessary.
5. **Traceback** – responsible for recursively tracing back from the maximum cell to a cell with a score of zero. It will generate the optimal local alignment in a reverse order by trace back recursively according to the source of each cell.
6. **Controller** – FSM that provides control signals to all the sub-units.

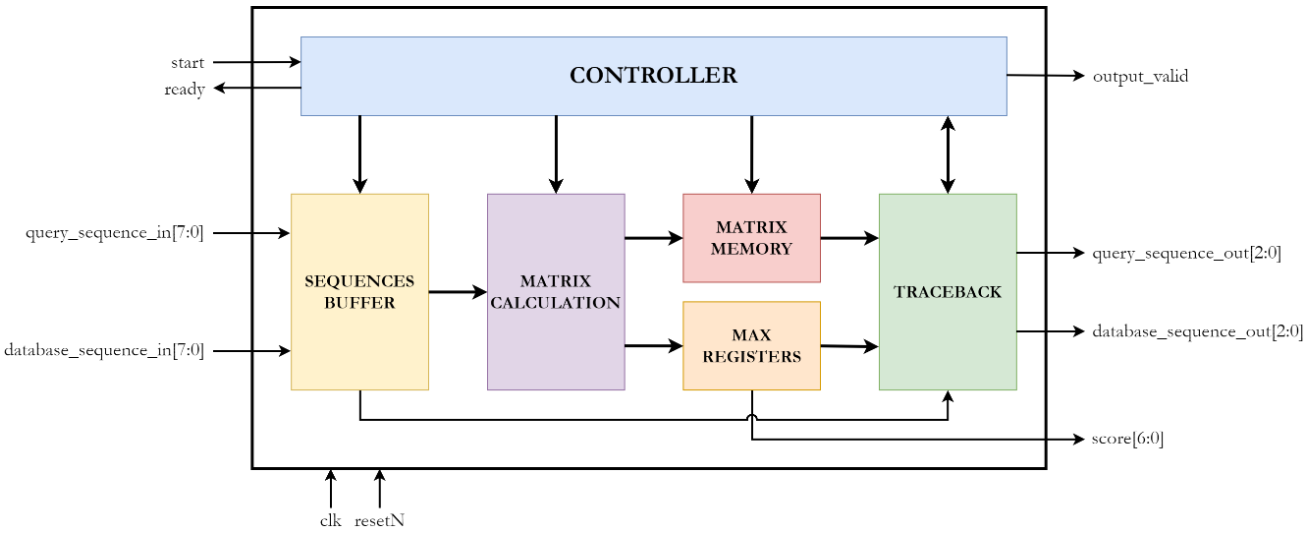


Figure 6: Top Level Architecture Scheme

4.4 High Level Architecture Stages

The Alignment process of the accelerator consists of 4 main stages:

1. **Idle** – Signifies that the chip is awaiting a start signal, indicating its readiness to perform a sequence alignment task. To proceed to the next stage, a "handshake" of 'start' and 'ready' signals is required.
2. **Load Sequences** – Load the two input sequences into the Sequences Buffer Unit in parallel – 4 residues from each sequence (query and database) on each cycle. This stage takes 8 cycles to store the 32 letters sequences.
3. **Score Matrix Computation** – The substitution matrix is filled using the Matrix Calculation Unit, with updates happening concurrently in the Matrix Memory and Max Registers Units. This process begins one cycle after the start of the previous stage and continues for a total of 32 cycles.
4. **Traceback** – Start at the highest-scoring cell in the matrix and follow the path of the highest-scoring cells until a cell with a zero score is reached. This is done by the Traceback Unit. The traceback stage takes between 1 to 31 cycles depending on the route to a zero-score cell.

Figure 7 displays a timeline with stages and their respective active units. The Controller Unit controls and schedules the computation stages.

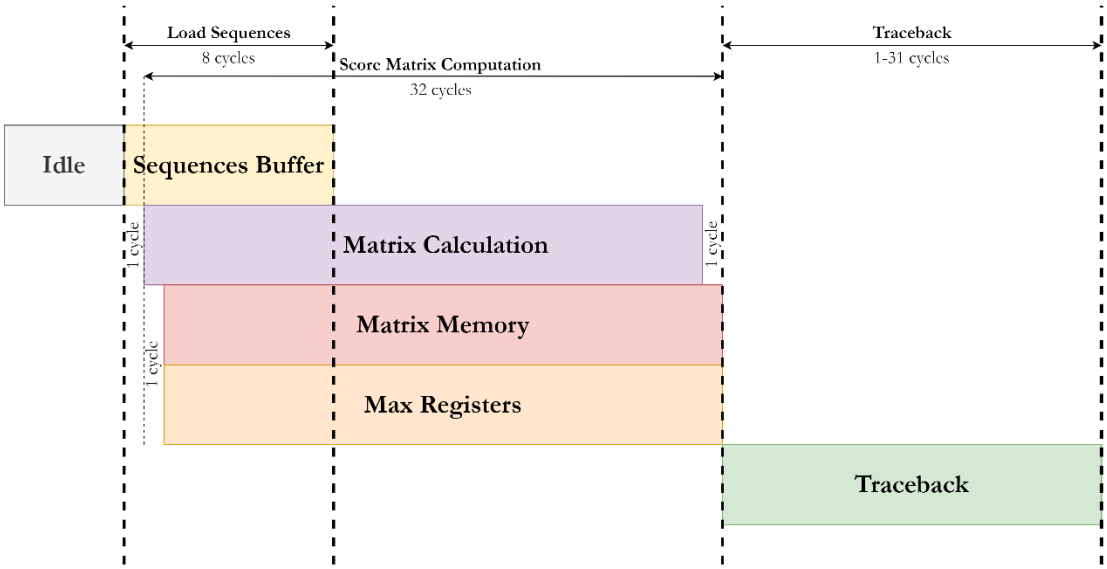


Figure 7: High-Level Architecture Stages

4.5 Letters Representation

The input sequences are represented differently compared to the aligned output sequences, because of the need to represent a gap in the aligned sequences.

Letter	Value
A	00
G	01
T	10
C	11

Table 6: Input Sequences Represantation

Letter	Value
A	000
G	001
T	010
C	011
–	100
Don't care	101
Don't care	110
Start/End Signal	111

Table 7: Output Sequences Represantation

4.6 Data Packet

Each cell in the substitution matrix contains the following necessary information:

1. **Zero Score Bit** – Indicates if the cell's score equals zero.
2. **Source Bits** – 2 bits indicate the adjacent cell from where the maximum score was determined (left, top or diagonally located).

Data packets are kept in the Matrix Memory Unit, and they are used by the Traceback Unit. To reduce area, only the source and the zero score bit of each cell are stored, instead of keeping the whole score.

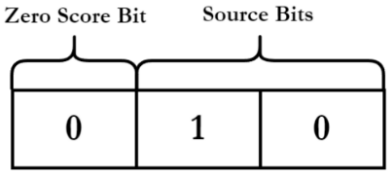


Figure 8: Data Packet Structure

5 Sequences Buffer

The Sequences Buffer Unit stores the query and database sequences. It receives them in 8 consecutive cycles, and stores 4 residues from each sequence on each cycle. The unit is controlled by the 'count' signal, which enables writing for the appropriate registers. Figure 9 shows the schematic of the Sequences Buffer Unit.

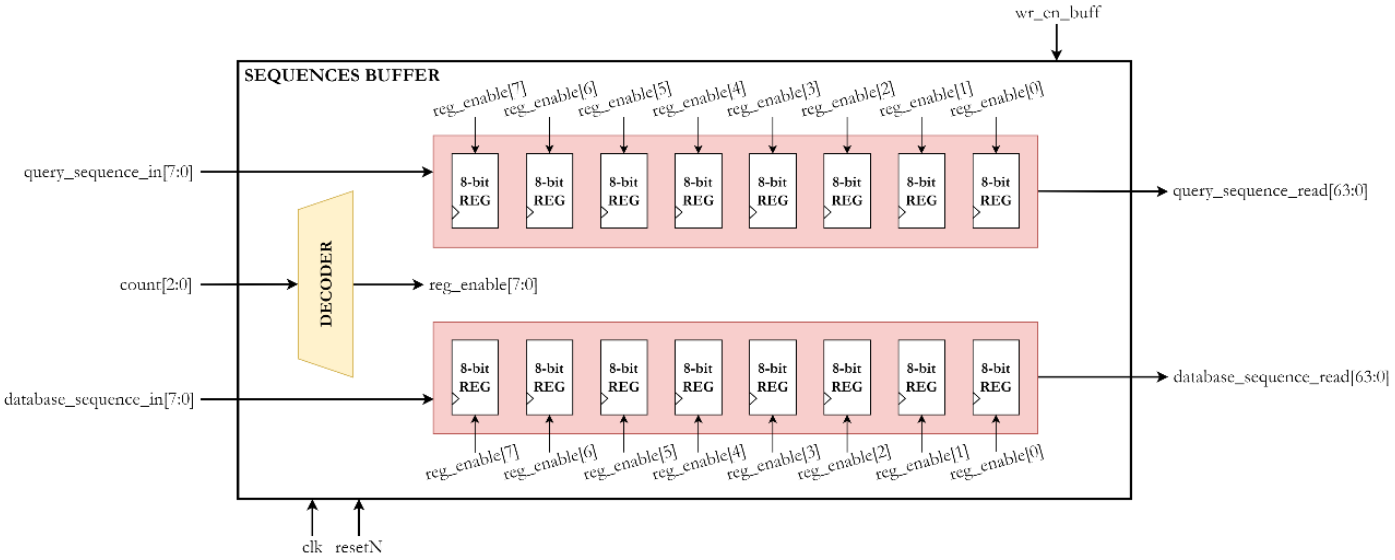


Figure 9: Sequences Buffer Scheme

5.1 Interface and Signals

The Sequences Buffer Unit receives the sequences letters as design inputs and control signals from the controller. The outputs of the unit are the stored sequences that are used by Matrix Calculation Unit and Traceback Unit.

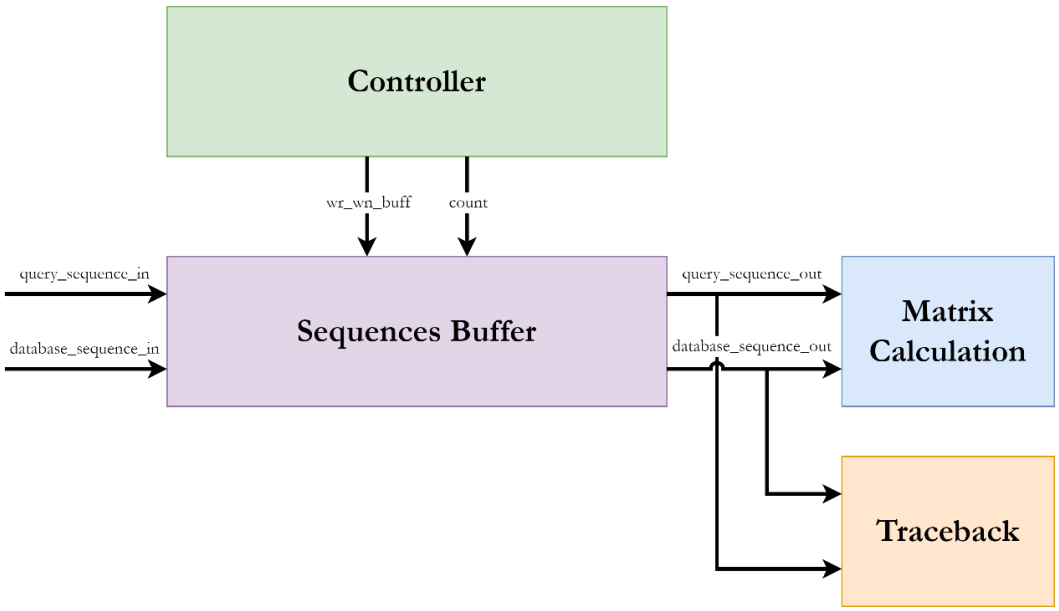


Figure 10: Sequences Buffer Interface

Signal Name	In/Out	Size (bits)	Description
clk	input	1	System's clock.
rst_n		1	Reset signal, active on low phase.
wr_en_buff		1	Enables loading of sequences.
query_sequence_in		8	4 residues of the query sequence.
database_sequence_in		8	4 residues of the database sequence.
count		3	Controller's signal that counts from 0 to 7.
query_sequence_out	output	2x32	32 letters of the query sequence.
database_sequence_out		2x32	32 letters of the database sequence.

Table 8: Sequences Buffer I/O Signals

6 Matrix Calculation

6.1 Processing Element

The Processing Element serves as the fundamental computational unit in the design. A single Processing Element (PE) represents a cell in the substitution matrix. The PE does the basic operation of computing the score of the current cell: it takes the top, left and diagonal scores from the adjacent cells and performs the calculation based on the scoring system described in section 3.1.

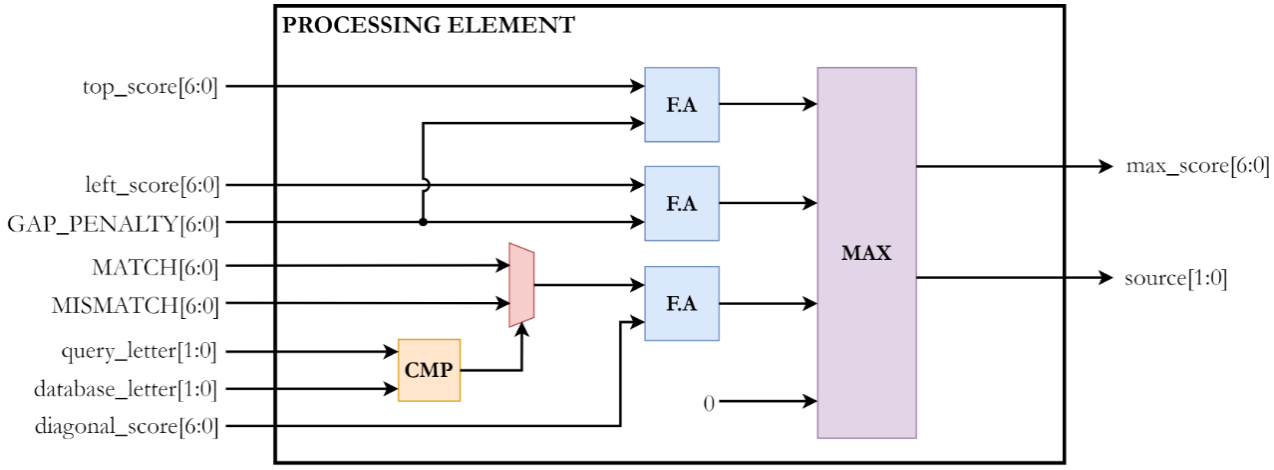


Figure 11: Processing Element Scheme

6.1.1 Processing Element Interface

Signal Name	In/Out	Size (bits)	Description
top_score	input	7	Score value of the top cell.
left_score		7	Score value of the left cell.
diagonal_score		7	Score value of the diagonal cell.
query_letter		2	The related letter from the query sequence that the PE uses to determine if it is a match or a mismatch is exist.
database_letter		2	The related letter from the database sequence that the PE uses to determine if it is a match or a mismatch is exist.
max_score	output	7	The maximum score out of all 3 directions: top, left and diagonal.
source		2	Direction from where the cell's score was updated.

Table 9: Processing Element I/O Signals

6.1.2 Max Unit

The Max Unit is a basic unit that outputs the maximum score among three values (or zero) and provides the source of this maximum score.
The possible values of the source are:

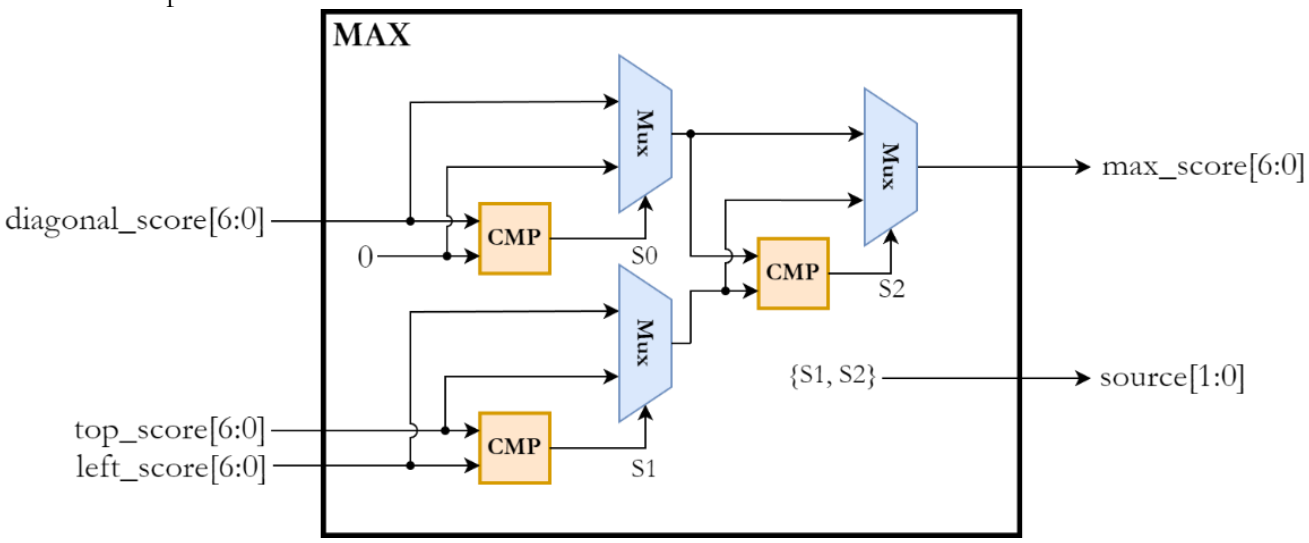


Figure 12: Max Unit Scheme

- 2'b00 – source is the diagonal adjacent cell
- 2'b01 – source is the left adjacent cell
- 2'b11 – source is the top adjacent cell

6.2 Processing Unit

The processing Unit (PU) is a structure of 4 PEs arranged in a 2x2 matrix.

PE 0 = $(00)_b$	**PE 1 = $(01)_b$**
PE 2 = $(10)_b$	**PE 3 = $(11)_b$**

Figure 13: Processing Unit Structure

The Processing Unit calculates the data packets of the 4 cells in a single cycle. The PU can send data to other PUs by providing the scores of the internal PEs calculated in the current cycle. Given its composition of four PEs, the PU derives its value from the top, left and diagonal directions. Additionally, for each PE, an indicator of a zero calculated score is

sent to the Matrix Memory Unit, together with the source of each cell. Packing the zero-score indicator with the cell's source forms a data packet as described in section 5.6.

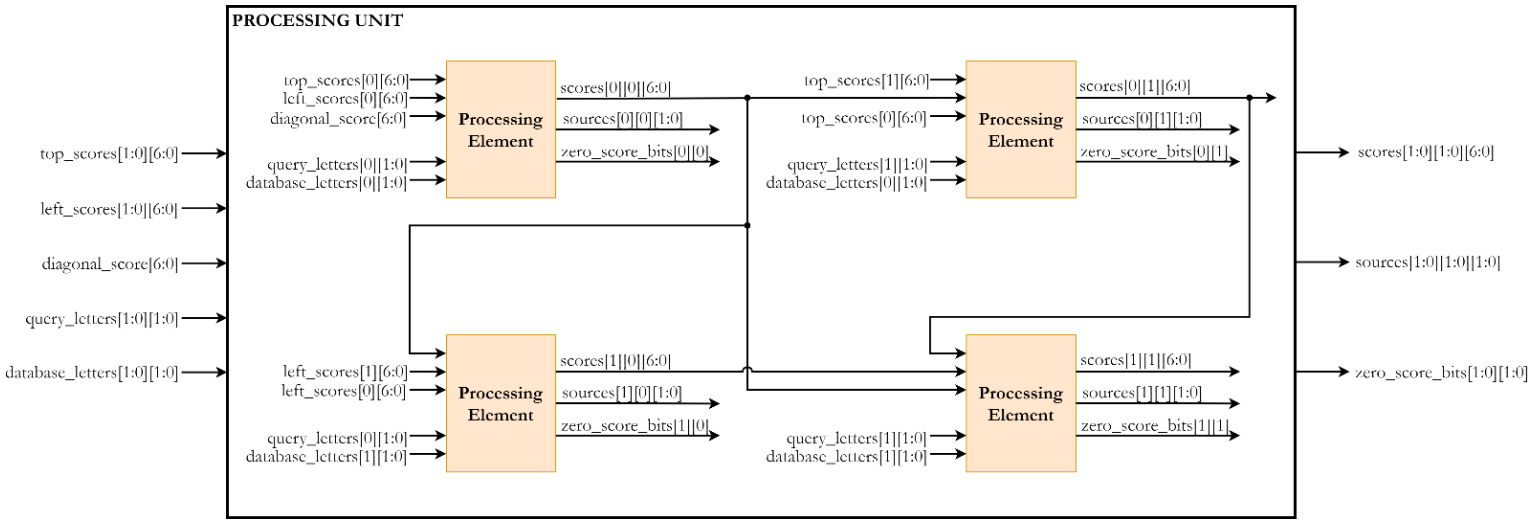


Figure 14: Processing Unit Scheme

6.2.1 Processing Unit Interface

Signal Name	In/Out	Size (bits)	Description
top_score	input	2x7	Score values coming from the top cells (2 scores in total).
left_score		2x7	Score values coming from the left cells (2 scores in total).
diagonal_score		7	Score value coming from the diagonal cell (a single score only).
query_letters		2x2	2 letters of the query sequence related to the current PU cells. Used to determine if a match or a mismatch is existing.
database_letters		2x2	2 letters of the database sequence related to the current PU cells. Used to determine if a match or a mismatch is existing.
scores	output	2x2x7	The scores of the 4 cells computed by the PEs.
sources		2x2x2	Direction from where the score of each cell was determined.
zero_score_bits		2x2	Indication if the score of a PE is equals zero.

Table 10: Processing Unit I/O Signals

6.3 Matrix Calculation

The Matrix Calculation Unit is responsible for performing the substitution matrix score computation. It consists of 16 identical PUs that perform the same task: compute scores and assemble data packets. The amount of 16 PUs serves as a strict lower limit for the PUs to work in parallel along the main diagonal. Since this unit consists of a PUs array, the computation is done in the granularity of PUs. The 16 PUs are connected to each other and can work together in parallel to provide the required data on each cycle.

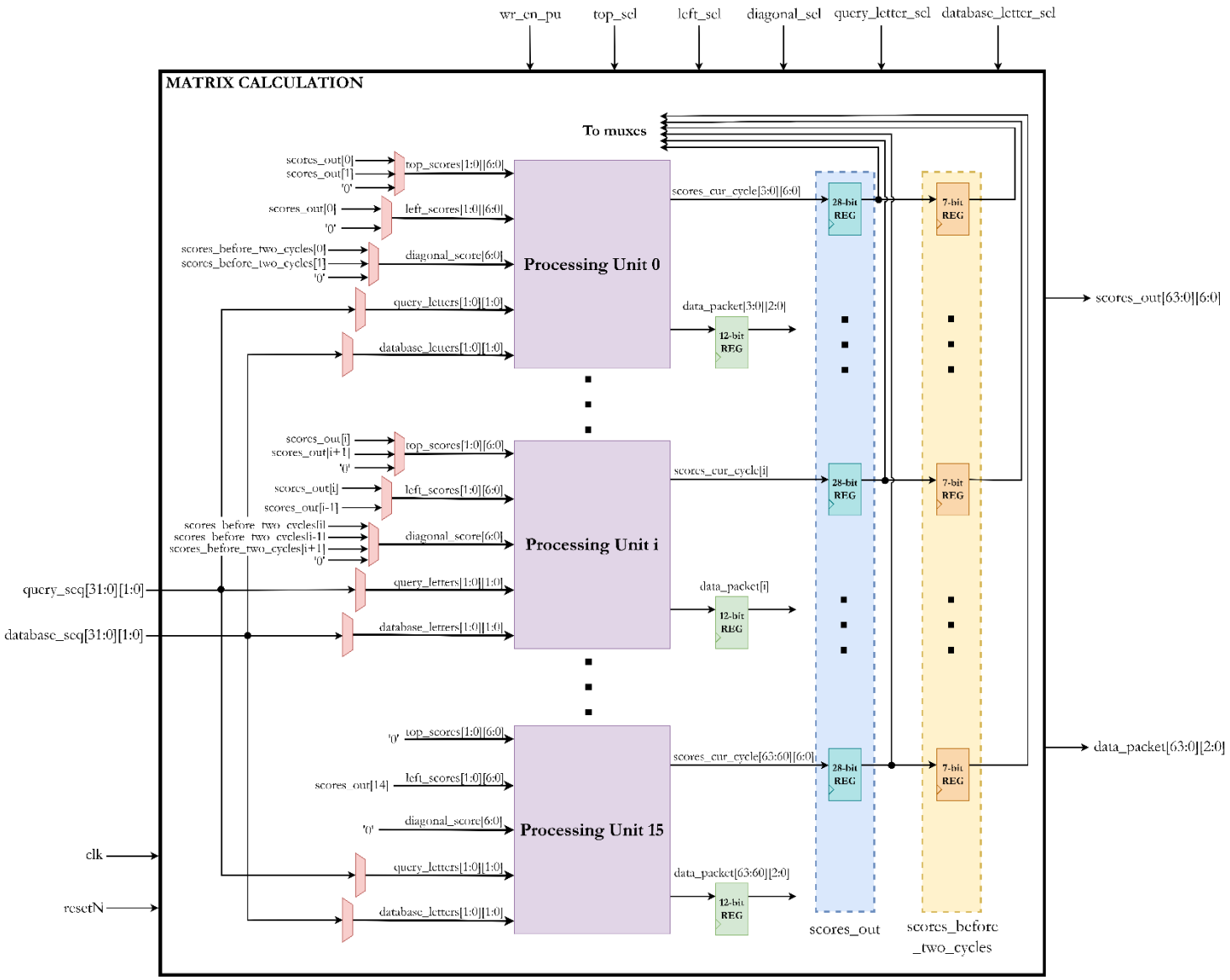


Figure 15: Matrix Calculation Scheme

6.3.1 Matrix Calculation Interface

Signal Name	In/Out	Size (bits)	Description
clk	input	1	System's clock.
rst_n		1	Reset signal, active on low phase.
query_seq		32x2	Query sequence, consists of 32 letters.
database_seq		32x2	Database sequence, consists of 32 letters.
top_sel		16x2	16 select signals for the 16 PUs, determines the inserted values from the top direction.
left_sel		16x2	16 select signals for the 16 PUs, determines the inserted values from the left direction.
diagonal_sel		16x2	16 select signals for the 16 PUs, determines the inserted value from the diagonal direction.
query_letter_sel		16x2x5	16 select signals for the 16 PUs, determines the 2 related query letters.
database_letter_sel		16x2x5	16 select signals for the 16 PUs, determines the 2 related database letters.
wr_en_pu		16	Enables the registers of the PUs output.
scores_out	output	16x2x2x7	The scores output from the 16 PUs.
data_packet		16x2x2x3	The data packets output from the 16 PUs.

Table 11: Matrix Calculation I/O Signals

6.3.2 Computation Flow

The computation starts from the top left corner in the matrix and ends in the bottom right corner. Figure 16 describes the direction flow of the computation (each color refers to a different calculation cycle). It is done in a diagonal manner: in the first cycle, the unit calculates the top left diagonal and uses the first PU only. Then, in the second cycle, it calculates the next diagonal while using the first two PUs. The calculation continues until it gets to the 16th cycle. In this cycle the unit computes the main diagonal and uses all the 16 PUs. In the 17th cycle the unit computes the next diagonal and uses 15 PUs, and so on, until it gets to the 31st cycle, in which the unit computes the bottom right diagonal using a single PU.

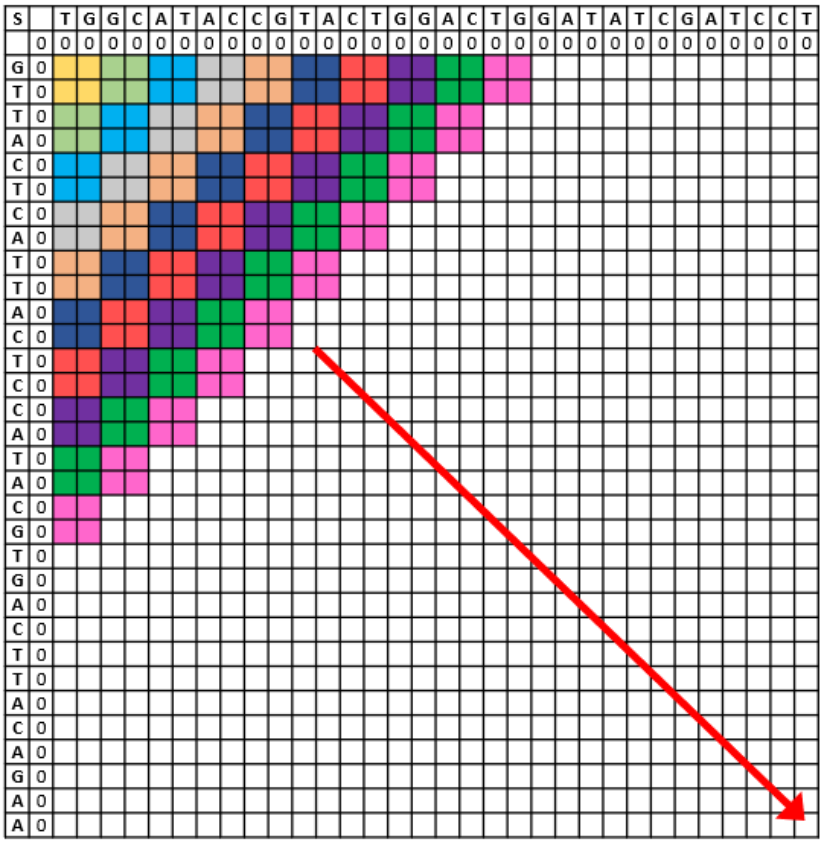


Figure 16: Matrix Computation Flow

6.3.3 Interaction Between PUs

Assume PU_i from the PUs array is performing a calculation. For this, it needs 5 scores of the adjacent cells, as described in figure 17.

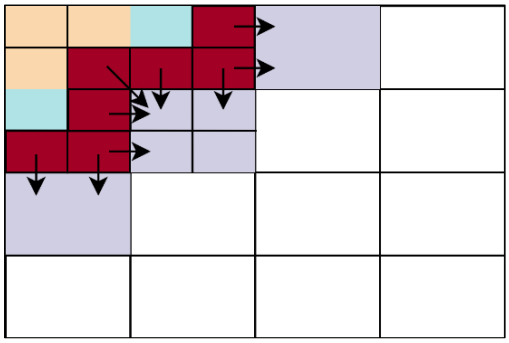


Figure 17: PU's Needed Scores

Figure 18 illustrates the interactions of a PU with index of i with other PUs in the PUs array, and the data selection performed by the multiplexers. It should be noted that in the case of the first or the last PU in the PUs array ($i = 0$ or $i = 15$), the multiplexers' entries are slightly different because of the initialized zeros in the first row and column of the substitution matrix.

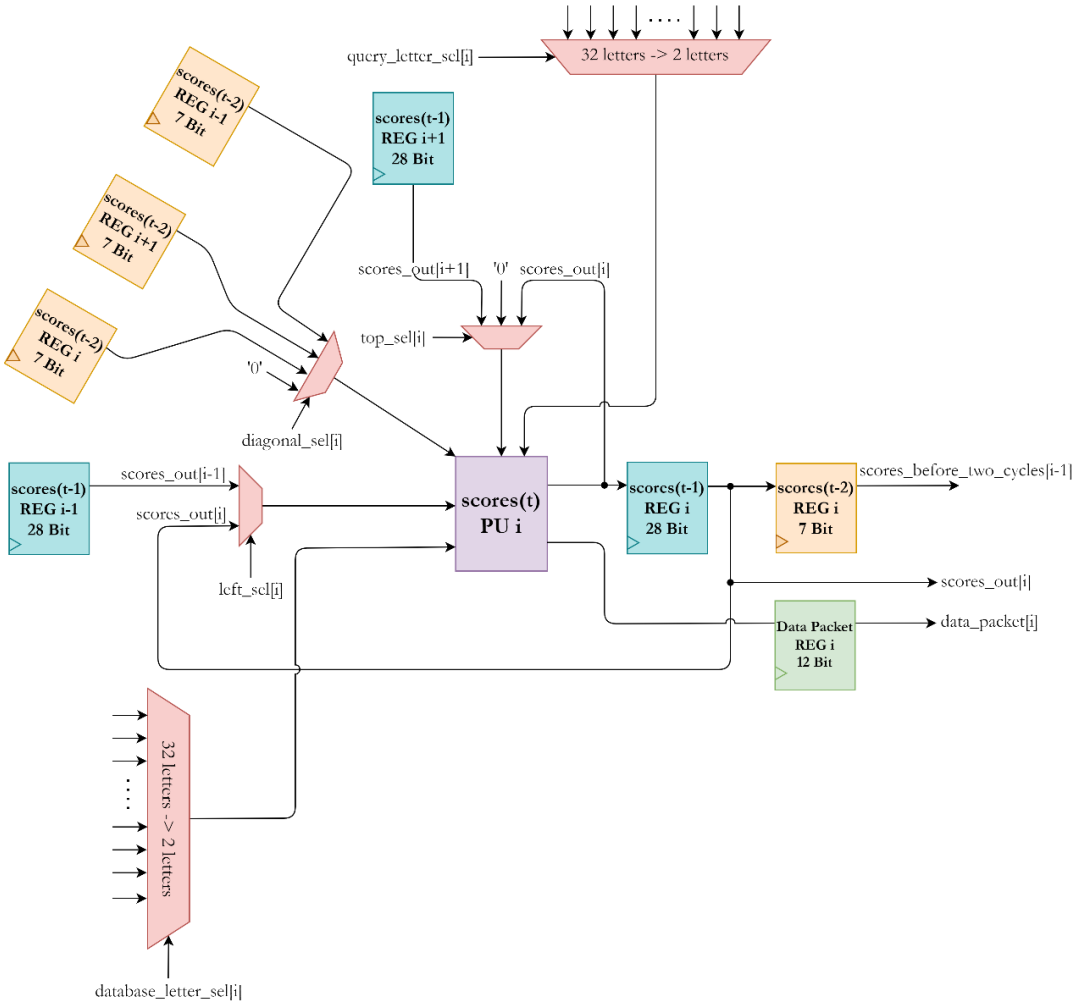


Figure 18: Interaction Between PUs

For the two adjacent top cells, it needs to choose between:

- Scores calculated by PU_i in the previous cycle.
- Scores calculated by PU_{i+1} in the previous cycle.
- Constant zero scores (in case of the first row in the matrix).

For the two adjacent left cells, it needs to choose between:

- Scores calculated by PU_i in the previous cycle.
- Scores calculated by PU_{i-1} in the previous cycle.

For the adjacent diagonal cell, it needs to choose between:

- Scores calculated by PU_i in the previous cycle.
- Scores calculated by PU_{i-1} in the previous cycle.
- Scores calculated by PU_{i+1} in the previous cycle.
- Constant zero scores (in case of the top left cell in the matrix).

These decisions are made by the 'top_sel', 'left_sel' and 'diagonal_sel' control signals. It is also necessary to know the query and database letters related to the current cells that are computed by the PU. This is done by the 'query_letter_sel' and 'database_letter_sel' control signals. Explanation on the algorithm that determines the selectors' values is described in section 11.2.2.

7 Matrix Memory

The Matrix Memory Unit stores the data packets of all the matrix cells that computed by the Matrix Calculation Unit. To reduce area, the unit stores the source and the zero score bit of each cell only, instead of keeping the whole score. The unit designed in a coned structure, for fast and efficient write to the registers.

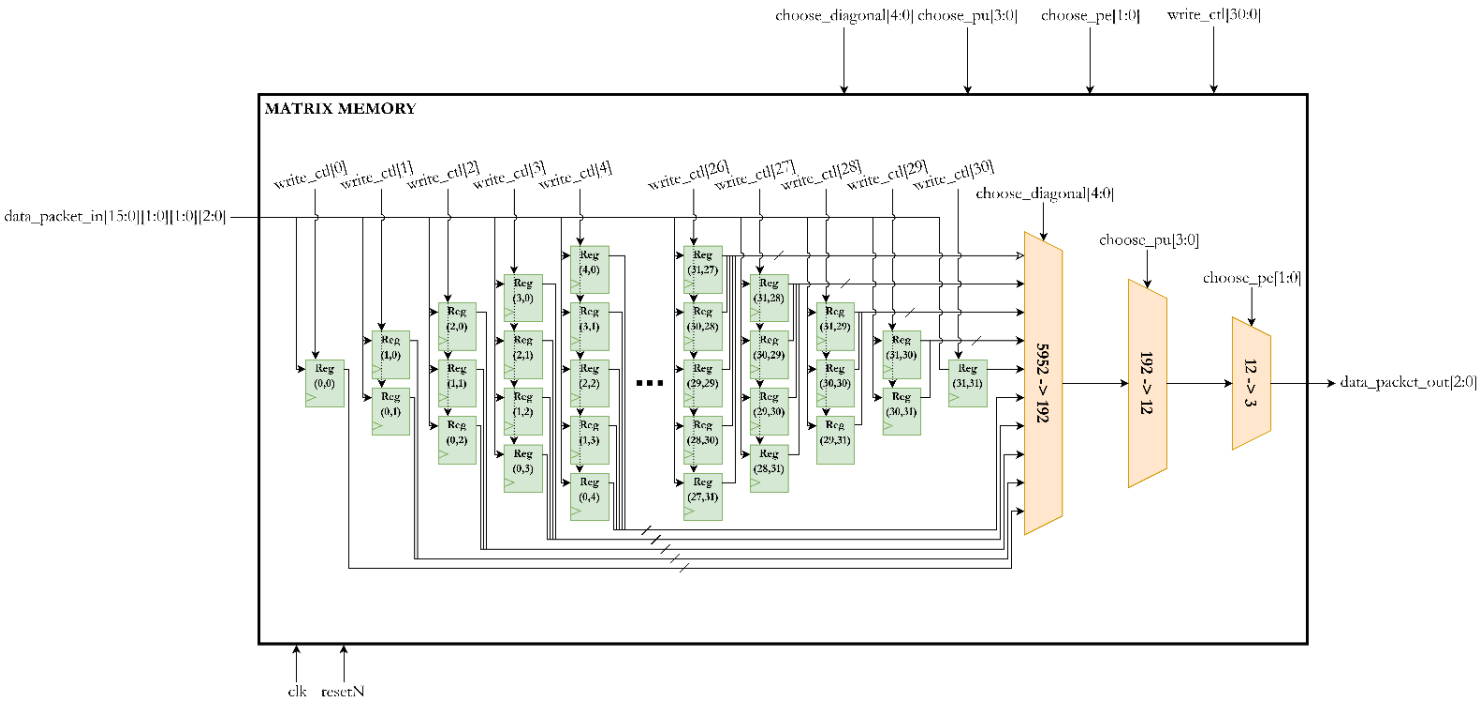


Figure 19: Matrix Memory Unit

7.1 Write Mechanism

After every cycle of matrix computation, all the data packets related to one diagonal in the matrix are sent from the Matrix Calculation Unit to the Matrix Memory Unit. Each data packet input is connected to the input of the appropriate register of each diagonal and written only to one enabled register, which determined by the 'write_ctl' signal provided by the controller. This is a one hot vector – the place in which the '1' is set represents the diagonal that will be enabled. When all the bits are '0', the write to the memory is disabled.

7.2 Read Mechanism

The read mechanism consists of 3 muxes. The first mux chooses all the scores of a specific diagonal, the second mux chooses the scores of a PU in that diagonal, and the third mux choose the requested data packet inside that PU. These 3 muxes are controlled by the following control signals:

1. choose_diagonal – determines the diagonal with the required information out of the 31 diagonals in the matrix.
2. choose_pu – determines the PU with the required information out of the 16 PUs.

3. choose_pe – determines the PE with the required information out of the 4 PEs.

Those signals are necessary to send the required data packet to the Traceback Unit.

7.3 Interface and Signals

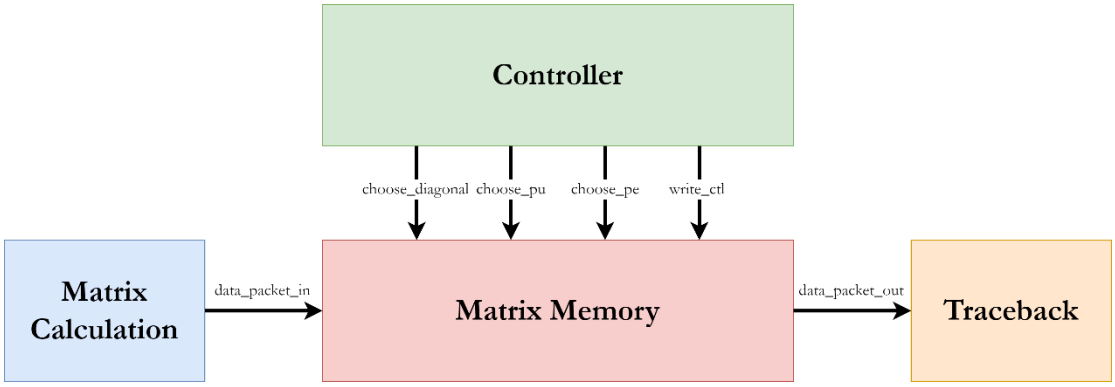


Figure 20: Matrix Memory Interface

The Matrix Memory Unit receives as an input all the data packets of the current diagonal from the Matrix Calculation Unit and outputs one data packet in a cycle for the Traceback Unit. It receives from the controller necessary control signals for reading the requested data packet for the Traceback Unit, as well as enable signals for writing the data packets from the Matrix Calculation Unit.

Signal Name	In/Out	Size (bits)	Description
clk	input	1	System's clock.
rst_n		1	Reset signal, active on low phase.
write_ctl		31	One hot vector that enables writing to the appropriate diagonal.
choose_diagonal		5	Determines the requested diagonal for read.
choose_pu		4	Determines the requested PU for read.
choose_pe		2	Determines the requested data packet for read.
data_packet_in		16x2x2x3	Data packets of a single diagonal (consists maximum of 64 data packets).
data_packet_out	output	3	Data packet for the Traceback Unit.

Table 12: Matrix Memory I/O Signals

8 Max Registers

The Max Registers Unit is responsible for finding the maximum cell in the Matrix. It does so by comparing all scores of a diagonal computed by the Matrix Calculation Unit in the previous cycle. To simplify the calculation, the computation is divided into 3 main stages:

1. Finding the maximum cell in each PU separately. There are up to 16 PUs computed by the Matrix Calculation Unit in one cycle. Thus, there are 16 components that computes the maximum of each PU in parallel.
2. Finding the maximum cell in the current diagonal. It is computed by comparing the maximum cells computed in the previous stage and finding the maximum cell out of them.
3. Comparing the maximum cell of the current diagonal with the maximum cell already written into the registers. If the new score of the maximum cell is bigger than the score already written, the registers will be updated with the new maximum cell.

To perform this operation and update the maximum cell on each cycle, it is necessary that all these 3 stages are performed in a single cycle. It is also necessary to keep the row and column indexes of the maximum cell, in addition to the score itself.

8.1 Interface and Signals

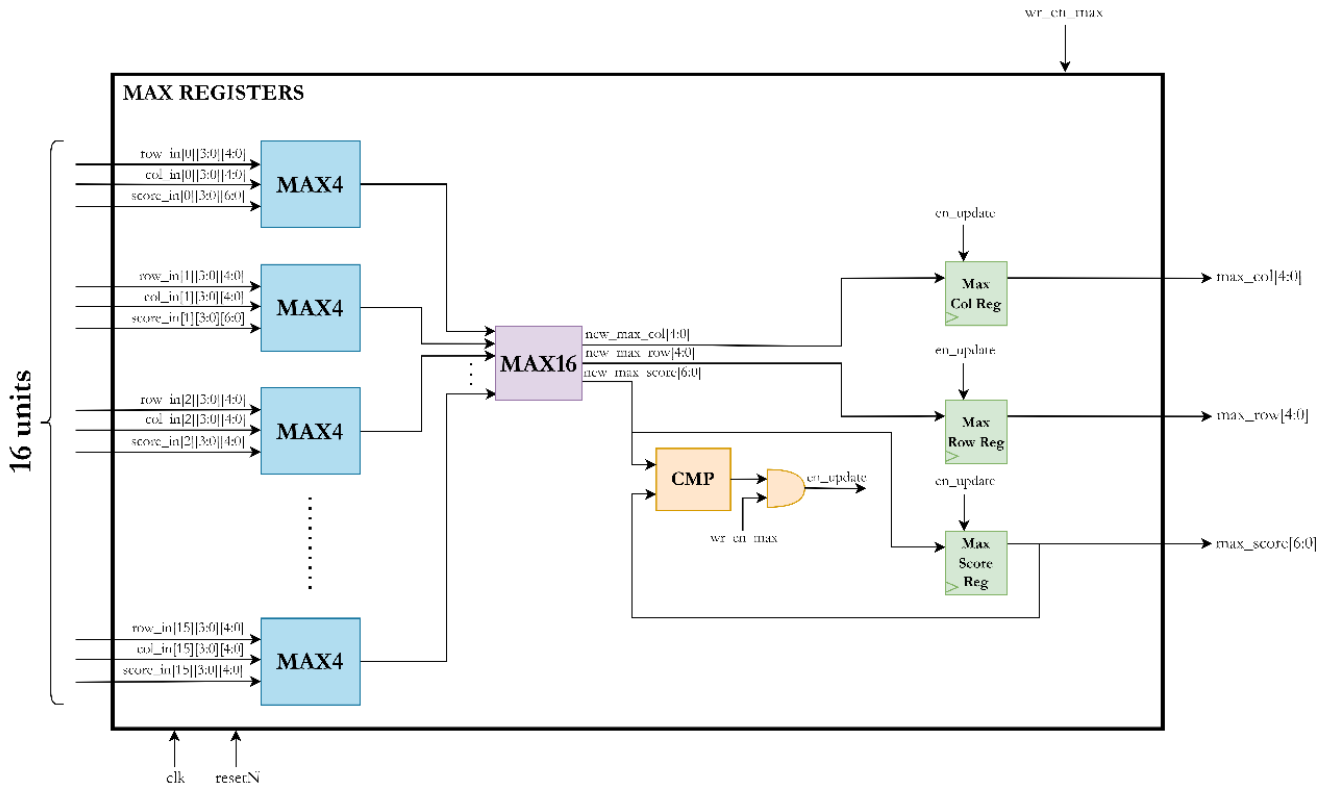


Figure 21: Max Registers Scheme

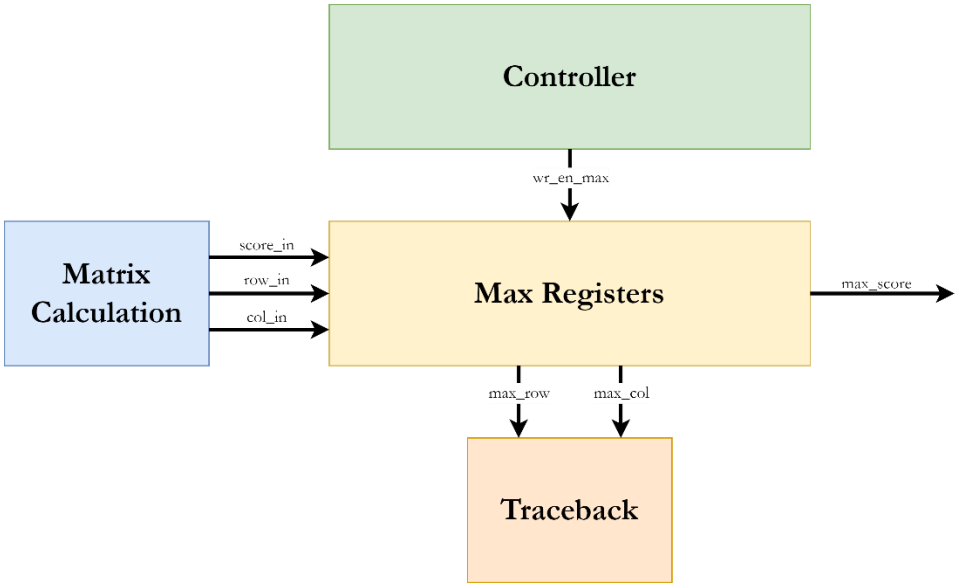


Figure 22: Max Registers Interface

The Max Registers Unit receives as an input the required information on each cell from the Matrix Calculation Unit. The indexes of the row and column of the maximum cell are sent to the Traceback Unit after the whole matrix calculation stage is finished. The maximum score is sent directly to the design output.

Signal Name	In/Out	Size (bits)	Description
clk	input	1	System's clock.
rst_n		1	Reset signal, active on low phase.
row_in		16x4x5	Row index of the cells computed in the previous cycle.
col_in		16x4x5	Column index of the cells computed in the previous cycle.
score_in		16x4x7	Scores of the cells computed in the previous cycle.
wr_en_max		1	Enable signal from the controller.
max_score	output	7	The score of the maximum cell.
max_row		5	The row index of the maximum cell.
max_col		5	The column index of the maximum cell.

Table 13: Max Registers I/O Signals

9 Traceback

The Traceback Unit is responsible for output the aligned sequences. The unit does one step of trace back every cycle and outputs one letter from each sequence. Thus, the output of the aligned sequences is sent out in a reverse order.

The traceback process starts when the unit receives from the controller the

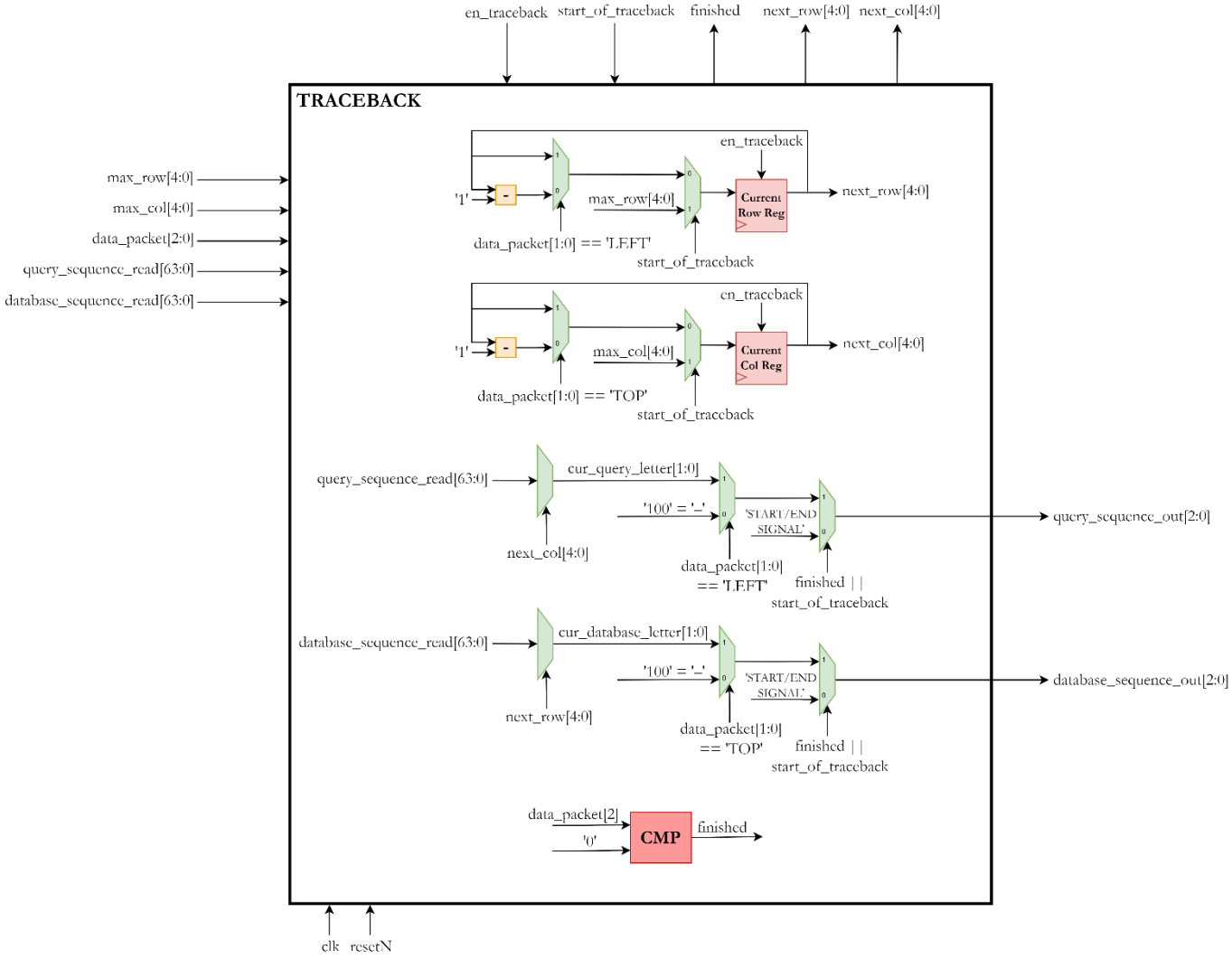


Figure 23: Traceback Scheme

'start_of_traceback' pulse, and the registers are enabled by 'en_traceback' signal. The unit starts in the cell with the highest score in the substitution matrix and sends to the controller the indexes of the row and column for the next traceback step. In every cycle, the unit calculates the next query and database letters of the aligned sequences, in parallel to the next row and next column calculation. This process continues until the current cell has a zero score bit that equals zero.

9.1 Interface and Signals

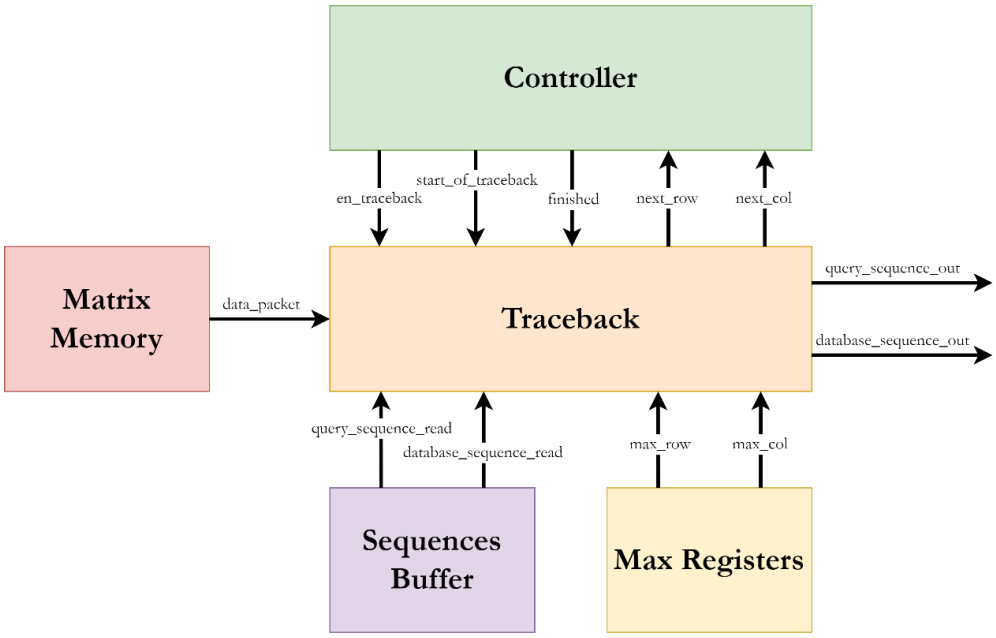


Figure 24: Traceback Interface

The Traceback Unit receives the information about the maximum cell in the matrix from the Max Registers Unit. It receives in each cycle the information of the current cell in traceback process from the Matrix Memory Unit and receives the query and database sequences from the Sequences Buffer Unit.

Signal Name	In/Out	Size (bits)	Description
clk	input	1	System's clock.
rst_n		1	Reset signal, active on low phase.
en_traceback		1	Enable signal for the unit's registers.
start_of_traceback		1	Indicates on start of the traceback stage.
finished		1	Indicate to the controller that the traceback stage has finished.
next_row		5	The row index of the next cell in the traceback process.
next_col		5	The column index of the next cell in the traceback process.
max_row		5	The row index of the maximum cell in the substitution matrix.
max_col		5	The column index of the maximum cell in the substitution matrix.
data_packet		3	The data packet of the current cell in the traceback process.
query_sequence_read		2	Current query letter.
database_sequence_read		2	Current database letter.

query_sequence_out	output	3	Current query letter of the aligned sequence.
database_sequence_out		3	Current database letter of the aligned sequence.

Table 14: Matrix Memory I/O Signals

10 Controller

The controller operates as a Finite State Machine, interacting with all the units within the chip. Its role is to send enable and control signals to all the sub-units and to manage communication between sub-units. It is responsible for managing the data flow across the entire system.

10.1 Controller as a FSM

The controller is a finite state machine implemented as a moore machine.

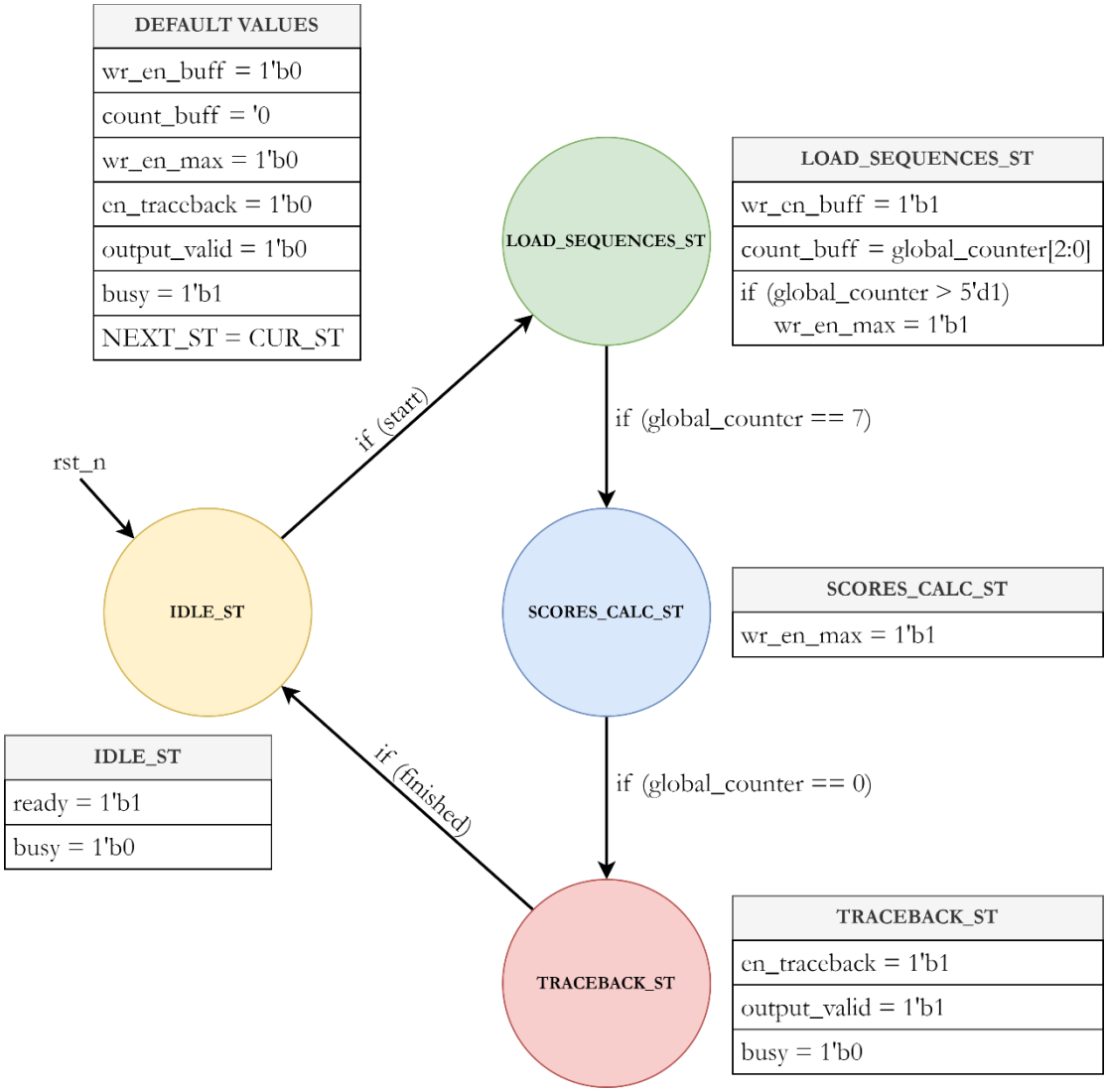


Figure 25: Controller's Finite State Machine

The controller is aided by the 'global_counter' internal signal. Global counter is a 5-bit counter which counts from 0 to 31. It is responsible for switching between states depending on its current value. Global counter is active only when the 'busy' signal is active and when the FSM is not in 'IDLE_ST'. When the 'finished' signal is received, the counter value is set to 0.

The Controller's FSM has 4 states:

- **IDLE_ST** – This state indicates that the chip is currently is not working on a sequence alignment task. Only when a "handshake" is made: the controller asserts the 'ready' signal and the user sends a 'start' signal for a single cycle, does the FSM move to the next state.
- **LOAD_SEQUENCES_ST –** In this state the global counter starts to count. Furthermore, the operation of the Sequences Buffer Unit is enabled. When global counter ≥ 1, the Matrix Calculation Unit is enabled and when global counter ≥ 2, the Max Registers Unit and the Matrix Memory Unit are enabled. This state takes 8 cycles exactly.
- **SCORES_CALC_ST –** This state takes 24 cycles exactly, during which the substitution matrix is continued to be filled.
- **TRACEBACK_ST –** In this state the Traceback Unit start to perform the traceback process until a 'finished' signal is sent to the controller. The global counter is inactive during this state.

10.2 Controller Interface

The controller has an interface with all the sub-units:

1. Sequence Buffer
2. Matrix Calculation
3. Matrix Memory
4. Max Registers
5. Traceback

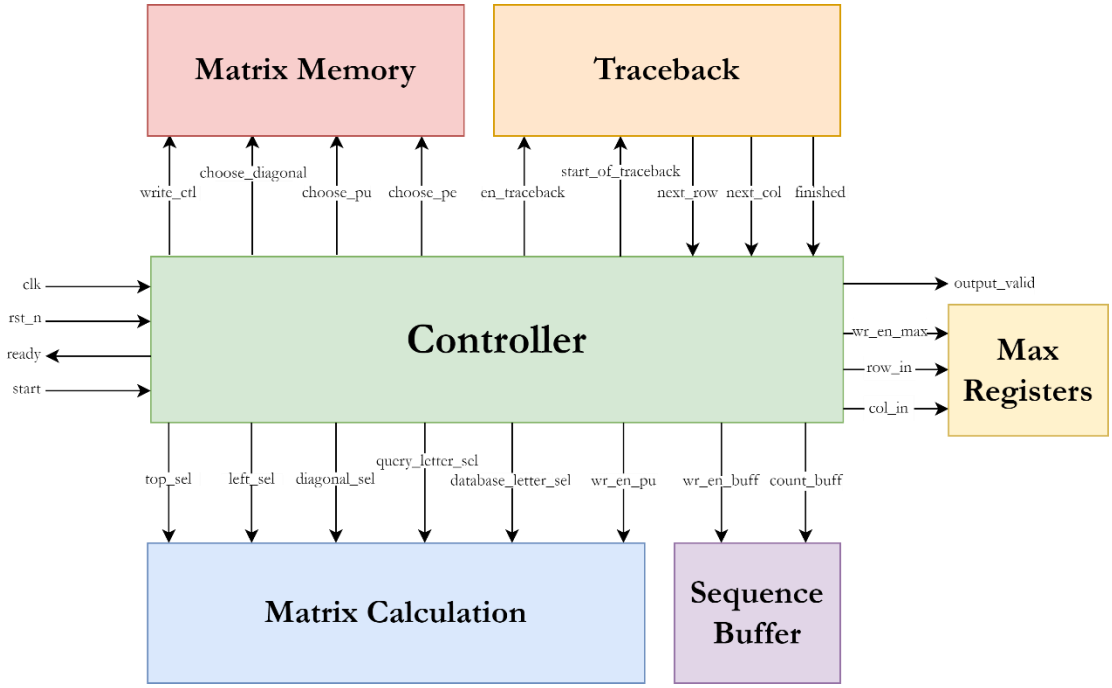


Figure 26: Controller Interface

Interface	Signal	In/Out	Size (bits)	Description
Clk & Reset	clk	input	1	System's clock.
	rst_n		1	Reset signal, active on low phase.
Sequence Buffer	wr_en_buff	output	1	Sequences Buffer enable.
	count_buff		3	Sequences Buffer counter.
Matrix Calculation	top_sel	output	16x2	Selector for the top values in the PUs array.
	left_sel		16x2	Selector for the left values in the PUs array.
	diagonal_sel		16x2	Selector for the diagonal value in the PUs array.
	query_letter_sel		16x2x5	Selector for the query letters in the PUs array.
	database_letter_sel		16x2x5	Selector for the database letters in the PUs array.
	wr_en_pu		16	Scores out enable.
Max Registers	wr_en_max	output	1	Max Registers enable.
	row_in		16x4x5	Row indexes of the current diagonal cells.
	col_in		16x4x5	Column indexes of the current diagonal cells.
Matrix Memory	write_ctl	output	31	One hot vector that enables write for the appropriate registers of the current diagonal.
	choose_diagonal		5	Requested diagonal for read.
	choose_pu		16	Requested PU for read.
	choose_pe		2	Requested PE for read.
Traceback	en_traceback	output	1	Traceback enable.
	start_of_traceback		1	Indicates to start the traceback process.
	next_row	input	5	The row index of the next cell in the traceback process.
	next_col		5	The column index of the next cell in the traceback process.
	finished		1	Indication for finishing traceback process.
Output	output_valid	output	1	Indicates output is valid.

Table 15: Controller I/O Signals

10.2.1 Sequences Buffer

The controller provides to the Sequences Buffer Unit an enable signal called 'wr_en_buff' to enable writing to the registers and a counter signal called 'count_buff' that enables writing into the specific registers of the Sequences Buffer.

10.2.2 Matrix Calculation

The Controller provides control signals for all the 16 Processing Units of the Matrix Calculation Unit. The controller ensures that the Matrix Calculation unit selects the right values for each Processing Unit during every computation cycle. Here is a pseudo code describes the algorithm of generating the control signals:

Query letters select:

```
for i in range (0, NUM_PU):
        if (global_counter <= 16): # Upper half
                query_letter_sel[i][0] = 2*j
                query_letter_sel[i][1] = 2*j + 1
        else: # Lower half
                query_letter_sel[i][0] = 2*(global_counter - 1 + i) - 30
                query_letter_sel[i][1] = 2*(global_counter - 1 + i) - 30 + 1
```

Database letters select:

```
for i in range (0, NUM_PU):
        if (global_counter <= 16): # Upper half
                database_letter_sel[i][0] = 2*(global_counter - 1 - i)
                database_letter_sel[i][1] = 2*(global_counter - 1 - i) + 1
        else: # Lower half
                database_letter_sel[i][0] = 31 - (2*i + 1)
                database_letter_sel[i][1] = 31 - 2*i
```

Top scores select:

```
for i in range (0, NUM_PU - 1):
        if (global_counter <= 16):
                if (i == 0):
                if (global_counter == 1):
                        top_sel[i] = 0
                else:
                                top_sel[i] = 1
                else if (i == global_counter - 1):
                top_sel[i] = 0
                else:
                top_sel[i] = 1
        else:
                top_sel[i] = 2
```

Left scores select:

```
for i in range (0, NUM_PU - 1):
        if (global_counter <= 16): # Upper half
                if (i == 0): # PU number 0
                left_sel[i] = 0
                else:
                left_sel[i] = 1
        else: # Lower half
                if (i == 0): # PU number 0
                left_sel[i] = 1
                else:
                left_sel[i] = 0
```

Diagonal scores select:

```
for i in range (0, NUM_PU - 1):
        if (global_counter <= 16): # Upper half
                if (i == 0): # PU number 0
```

```
                diagonal_sel[i] = 0
        else if (i == global_counter - 1): # PUs on the first row
                diagonal_sel[i] = 0
        else: # Lower half
                diagonal_sel[i] = 2
else:
        if (global_counter == 17): # Main diagonal
                diagonal_sel[i] = 1
        else:
                if (i == 0): # PU number 0
                        diagonal_sel[i] = 2
                else:
                        diagonal_sel[i] = 3
```

10.2.3 Max Registers

The controller provides the Max Registers Unit with an enable signal called 'wr_en_max' that allows it to operate. In addition, two vectors 'row_in' and 'col_in' represent the indexes of the cells in the current computation cycle of the PUs array.

10.2.4 Matrix Memory

The controller generates control signals for read and write operations in the Matrix Memory Unit. These signals specify the registers for reading or writing the data. Here is a pseudo code describes the algorithm of generating the control signals:

Choose diagonal:

```
choose_diagonal = (next_row >> 1) + (next_col >> 1)
```

Choose PU:

```
if (next_row + next_col ≤ 31):
        choose_pu = next_col >> 1
else:
        choose_pu = (31 - next_row) >> 1
```

Choose PE:

```
if (next_row[0] == 1) && (next_col[0] == 1):
        choose_pe = 2'b11; # bottom right PE
else if (next_row[0] == 0 && next_col[0] == 0):
        choose_pe = 2'b00; # top left PE
else if (next_row[0] == 0 && next_col[0] == 1):
        choose_pe = 2'b01; # top right PE
else if (next_row[0] == 1 && next_col[0] == 0):
        choose_pe = 2'b10; # bottom left PE
```

10.2.5 Traceback

The controller manages the Traceback Unit by transmitting control signals and additionally mediates the communication with the Matrix Memory Unit. The controller provides a single pulse called 'start_of_traceback' when it needs to start the traceback process. During the traceback stage, a signal called 'en_traceback' is set to '1', enabling the registers in the unit. When the traceback stage is done, a 'finished' signal is sent to the controller.

The Traceback Unit sends to the controller two signals: 'next_row' and 'next_col'. The controller translates them into select signals for reading the correct data from the Matrix Memory Unit.

11 Simulations and Verification

An important part of the design process is performing a verification of the chip. To avoid bugs, after the implementation of each unit is completed, we checked its behavior using logic simulations and verified that the signals seen in the simulation are corresponding to the expected behavior.

In this section will be presented a small subset of the simulations we made. The simulations made using VCS debug and verification management platform of Synopsys.

11.1 Sequences Buffer

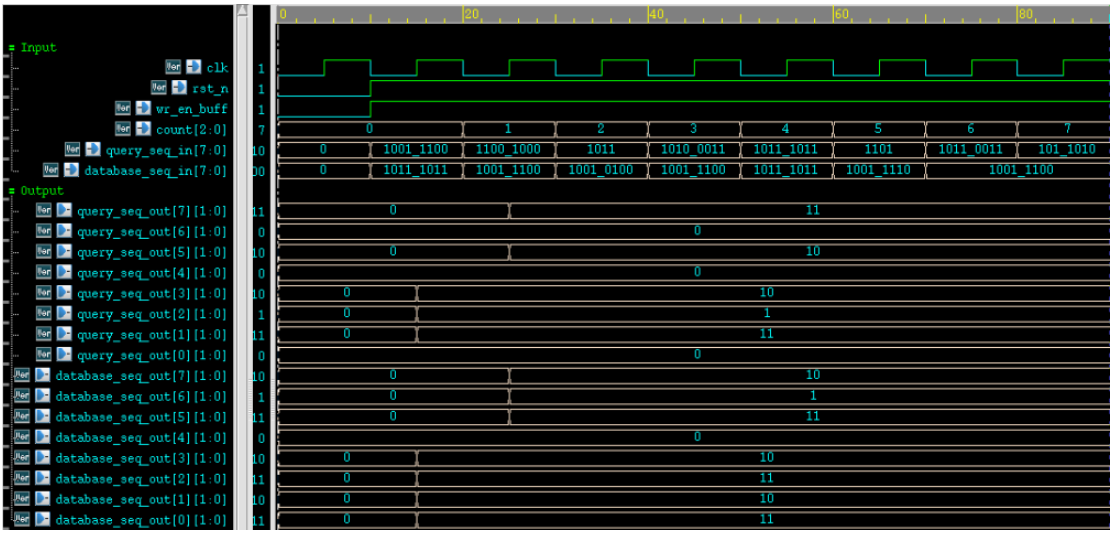


Figure 27: Sequences Buffer Simulation

The first 4 query letters received as an input in the first cycle are 10011100, which is TGCA. These letters are seen in the output after they are stored in the registers on the next clock rise. Similarly, the first 4 database letters received as an input on the first cycle are 10111011, which is TCTC. These letters are seen in the output after they are stored in the registers on the next clock rise. In the second cycle, the next 4 letters from each sequence are stored in the next registers, and so on.

11.2 Processing Element

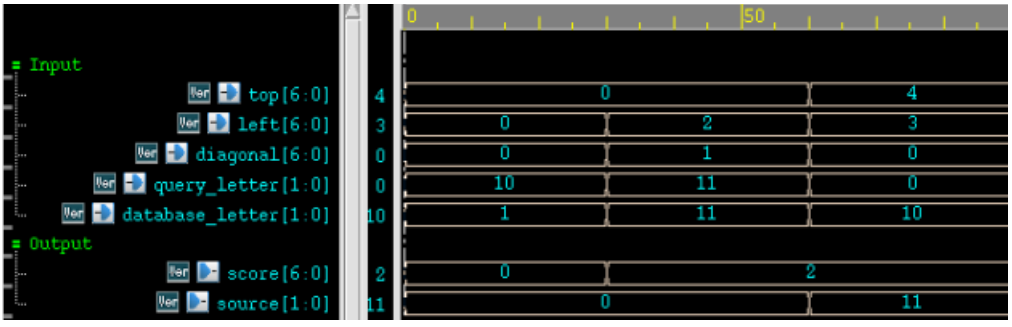


Figure 28: Processing Element Simulation

In the first cycle, all the 3 input scores are equal to 0 and there is a mismatch. Thus, the output score is 0 and the source is 00 (diagonal) by default. In the second cycle, the top score is 0, the left score is 2, the diagonal score is 1 and there is a match. Thus, the output score is 2 and the source is 00 (diagonal). In the third cycle, the top score is 4, the left score is 3, the diagonal score is 0 and there is a mismatch. Thus, the output score is 2 and the source is 11 (top).

11.3 Processing Unit

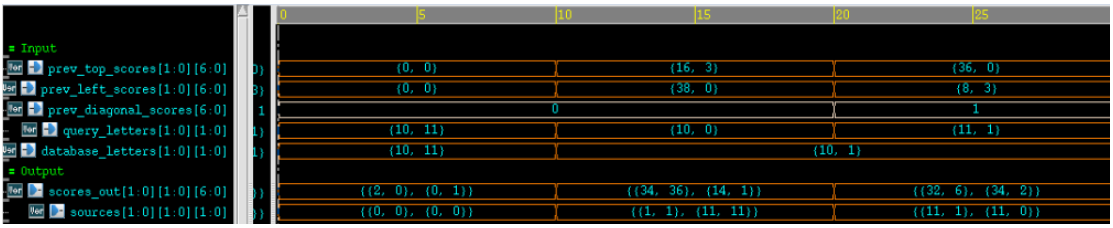


Figure 29: Processing Unit Simulation

Figure 30 shows the expected output related to the given simulation inputs.

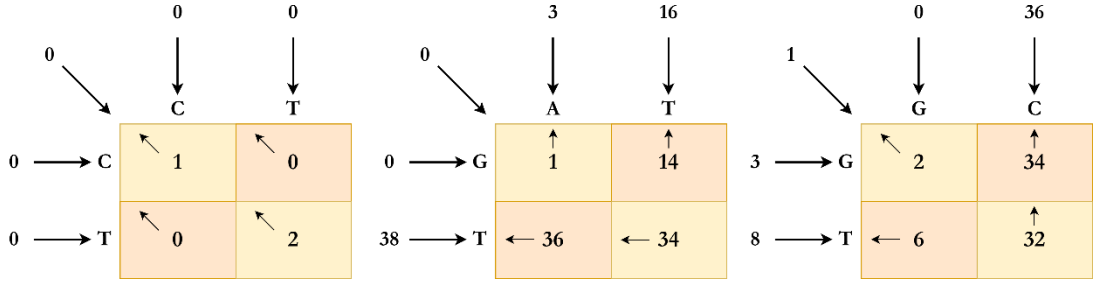


Figure 30: Processing Unit Expected Output

The results shown in the simulation correspond to the expected behavior of the Processing Element Unit.

11.4 Matrix Calculation

To simulate matrix calculation process, we entered the following pair of sequences:

Query sequence: ATCAGTACGCATCAGTACGCATCAGTACGCAT

Database sequence: CATCATCATCATCATCATCATCATCATCATCA

For simplicity, only the first 3 diagonals are shown in the simulation below. The selectors values were chosen in correspondence to the necessary values of the first 3 diagonals.

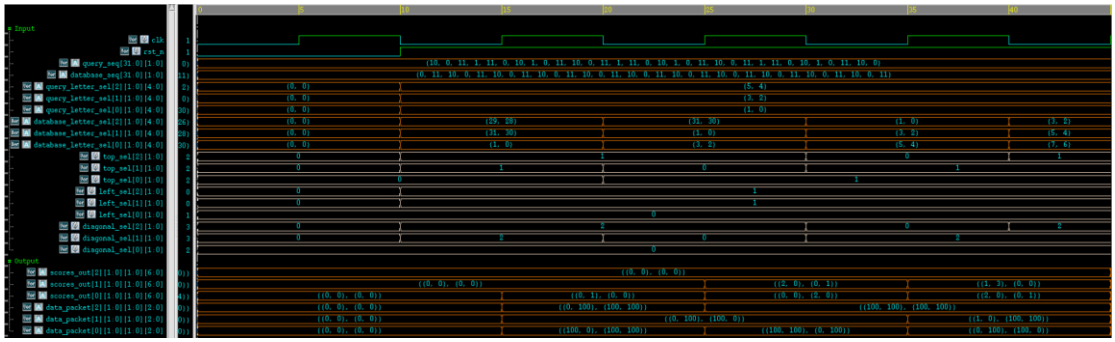


Figure 31: Matrix Calculation Simulation

Figure 32 shows the expected output related to the given simulation inputs.

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

Figure 32: Matrix Calculation Expected Output

The results shown in the simulation correspond to the expected behavior of the Matrix Calculation Unit.

11.5 Matrix Memory

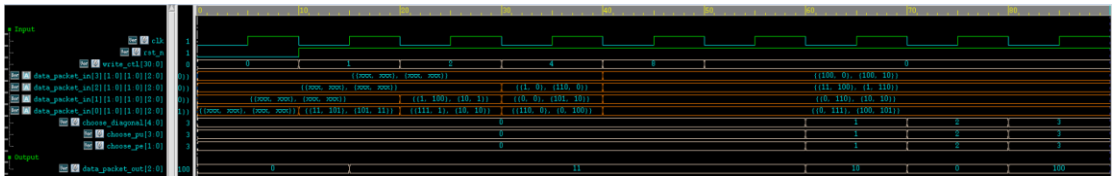


Figure 33: Matrix Memory Simulation

For simplicity, only the first 4 PUs are shown in each write cycle. In each cycle, the values of one diagonal are written to the Matrix Memory. Afterwards, selected values are chosen for read.

11.6 Max Registers

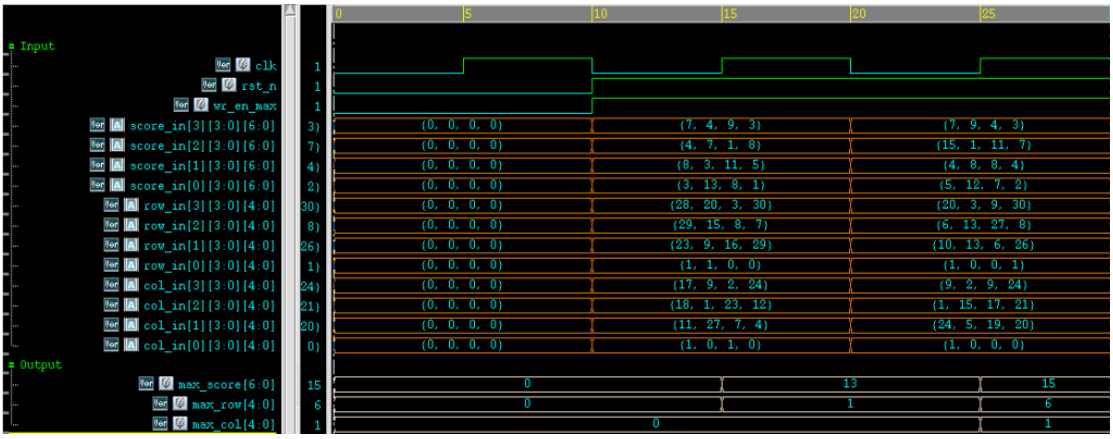


Figure 34: Max Registers Simulation

For simplicity, only the first 4 PUs are shown on each write cycle. In the first cycle the maximum is computed for the first time and the result is 13. In the next cycle there is a higher score, so the maximum is updated to 15. The indexes of the maximum cell are stored as well.

11.7 Traceback

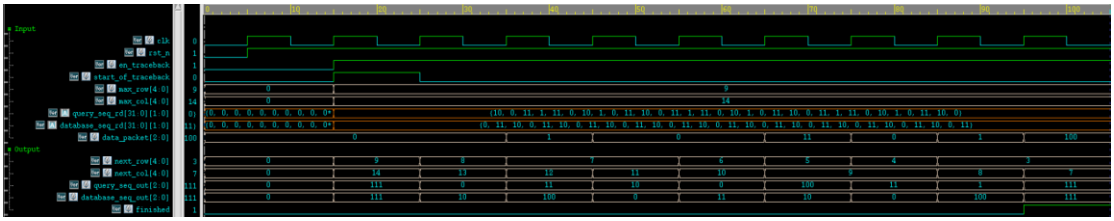


Figure 35: Traceback Simulation

In this simulation, the traceback stage takes 9 cycles. In the first and the last cycles, the letters output is 111 (start/end signal). After 9 cycles, the zero score bit equals zero. Thus, the 'finished' signal is sent and traceback stage is ended.

11.8 Full Simulations

We performed multiple simulations to check the logic's correctness in various scenarios. Below is shown one example of the simulations that has been performed.

ACTTCTAGTGATGTATACCGCCTTGACTTGCC

ATCAATCTACTACGTATTGTACCTGTAGTGCT

⇓

TCTAGTGATGTAT

TCTACT-ACGTAT

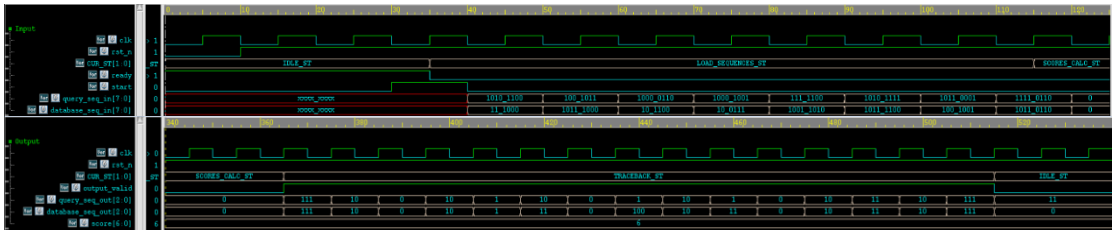


Figure 36: Full Simulation

The optimal alignment score is 6, as shown in the simulation.

SECTION C:
Verification

In fulfillment of the requirements for Project C (044170)

12 UVM (Universal Verification Methodology)

is a standardized methodology used for verifying complex hardware designs, particularly in the context of System Verilog-based testbenches. It provides a set of libraries, guidelines, and best practices for writing reusable, modular, and scalable verification environments.

Key Aspects of UVM:

1. **Object-Oriented**: UVM is based on object-oriented programming principles, leveraging classes, inheritance, and polymorphism. This structure enhances flexibility and extensibility, making verification environments easier to maintain, scale, and adapt to changes in the design.
2. **Reusability**: UVM emphasizes the reuse of verification components such as testbenches, sequences, and drivers. This saves significant time and effort when verifying multiple projects or different parts of a design, promoting efficiency and consistency.
3. **Modular Structure**: UVM adopts a modular approach, dividing the testbench into distinct layers, each with a specific responsibility. This simplifies complex verification environments by breaking them into smaller, manageable components, making debugging and updates more straightforward.
4. **Component Hierarchy**: UVM organizes components like agents, sequencers, drivers, monitors, and scoreboards in a structured hierarchy:
    - **Agents**: Generate and drive stimulus to the design under test (DUT).
    - **Sequencers**: Control the sequence of operations or transactions.
    - **Drivers**: Translate sequence items into signals that drive the DUT.
    - **Monitors**: Observe the DUT and collect coverage or check for expected behaviors.
5. **Scoreboards**: Compare expected results with actual results to detect discrepancies. This hierarchy facilitates communication and collaboration among components, ensuring efficient and accurate verification.
6. **Factory Pattern**: The factory mechanism allows objects to be created dynamically at runtime. This adds flexibility in configuring the testbench without modifying its code, making it adaptable to different scenarios and designs.
7. **Transaction-Level Verification**: UVM promotes verification at the transaction level, focusing on high-level operations rather than individual signals. This improves efficiency, simplifies testbench design, and provides better insight into system behavior.
8. **Randomization and Coverage**: Built-in random stimulus generation and coverage collection enable thorough testing by exploring corner cases and edge conditions. This ensures comprehensive verification and reduces the risk of undetected errors.
9. **Test Sequences**: UVM organizes tests as sequences that define the series of transactions applied to the DUT. Sequences can be randomized to explore unexpected

scenarios or predefined for specific functional tests, ensuring both flexibility and precision.

10. **Configurability**: UVM allows runtime configuration of components through configuration objects or the factory mechanism. This makes the testbench highly adaptable to different verification goals and design variations.
11. **Debugging and Automation**: UVM includes detailed logging, error reporting, and assertions, simplifying debugging and enabling automated verification workflows. This reduces manual effort and enhances productivity.

13 Testbench Architecture

The **Testbench (TB)** is developed using the **UVM framework** to verify the functionality and performance of the **Smith-Waterman accelerator** (the Design Under Test (DUT)). The architecture follows the principles of modularity, reusability, and configurability to create a robust verification environment. This architecture ensures that the DUT is tested under various conditions and scenarios to validate its correctness and performance.

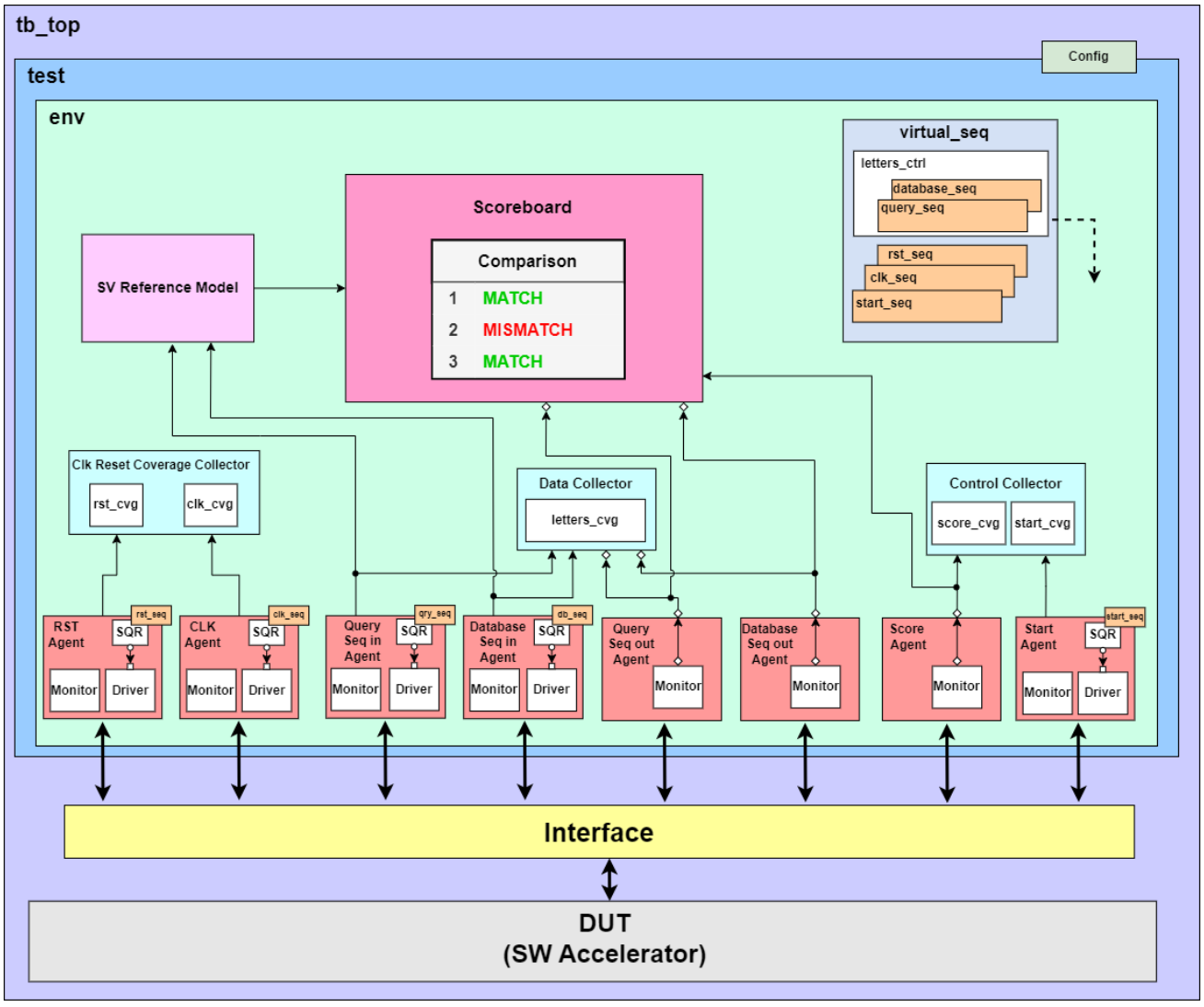


Figure 37: Testbench Architecture

Below is a detailed explanation of each component in the hierarchy:

13.1 Testbench Top

The **tb_top** module is the central structure of the testbench, connecting the DUT to all verification components. It instantiates various tests with unique configurations and integrates key components such as the environment, virtual sequences, scoreboard, coverage collectors, and agents. This modular design ensures smooth communication, synchronization, scalability, and comprehensive DUT verification.

13.2 Test

The Test object, inheriting from uvm_test, serves as the central controller in the UVM testbench. It orchestrates the verification process by defining scenarios, managing configurations, triggering sequences, and monitoring results, ensuring comprehensive and efficient validation of the Design Under Test (DUT). The base test, called base_test, is used by other tests that inherit its functionality and make necessary customizations. Here are the key responsibilities of the test:

1. **Environment Initialization -** Sets up the verification environment (sw_env) with the required components and configurations unique to each test.
2. **Configuration Management -** Utilizes a centralized configuration object (test_cfg) to define and apply randomized or directed values for each test run.
3. **Sequence Coordination -** Manages and triggers sequences across agents using the **Virtual Sequence (v_seq)**. In addition, it generates test stimuli, such as reset, clock, query, database, and start signals.
4. **Scenario Definition -** Defines random and directed scenarios to validate DUT functionality under varying conditions.
5. **Results Monitoring -** Connects analysis ports to the Scoreboard and Reference Model (ref_model) for DUT output comparison and collects coverage metrics to ensure comprehensive validation.
6. **Reference Model Execution-** Runs the reference algorithm in the ref_model during the test to calculate expected outputs for comparison.
7. **Finalization and Reporting-** Wraps up execution, evaluates the pass/fail condition, and generates a test report with coverage and results.

13.3 Environment (env)

The environment is the core of the testbench, containing and connecting all the main components, including UVM agents, coverage collectors, a reference model, virtual sequence and the scoreboard. Here is a detailed explanation about each component:

13.3.1 Agents

Agents are modular components responsible for interfacing with the DUT through drivers, monitors, and sequencers.

Each agent is responsible for a specific functionality:

1. **Reset (RST) Agent**:
    - This agent works with rst_seq packets. Each packet has a different reset duration.
    - It has a Sequencer (rst_seq) that generates reset sequences.
    - The driver sends reset packets to the DUT.
    - The monitor observes reset activity for verification and coverage purposes.
2. **Clock (CLK) Agent**:
    - This agent works with clk_seq packet. This packet can control the duty cycle, duration and choose a gated or ungated clock.
    - It has a Sequencer (rst_seq) that generates clk packets.
    - The driver sends clk packets to the DUT.
    - The monitor observes clock activity for functional and coverage analysis.
3. **Query Sequence Input Agent**:
    - This agent generates a qry_seq_in packet which inherits from the base_seq_in packet. The packet includes a parameterized number of sequence letters to insert into the design.
    - It has a Sequencer (qry_seq_in) that generates query sequences. Each packet is a portion (in our case 1/8) of the letters that form a full sequence.
    - The driver Drives query packets to the DUT. A full transaction is considered as 8 query packets.
    - The monitor observes query signals for coverage. It passes those inputs further to the reference model.
4. **Database Sequence Input Agent**:
    - This agent generates a db_seq_in packet which inherits from the base_seq_in packet. The packet includes a parameterized number of sequence letters to insert into the design.
    - It has a Sequencer (db_seq_in) that generates query sequences. Each packet is a portion (in our case 1/8) of the letters that form a full sequence.
    - The driver Drives query packets to the DUT. A full transaction is considered as 8 query packets.
    - The monitor observes query signals for coverage. It passes those inputs further to the reference model.
5. **Query Sequence Output Agent**:
    - It has a monitor that observes query output signals from the DUT for correctness. The output signals come as a full sequence packet.

6. **Score Agent**:
    - It has a monitor that observes the output score for coverage purposes.
7. **Start Agent**:
    - This agent works with start_seq packets. Each packet has a different start duration.
    - It has a Sequencer (start_seq) that generates start sequences.
    - The driver sends start packets to the DUT.
    - The monitor observes start activity for verification and coverage purposes.

13.3.2 Coverage Collectors

Coverage collectors collect coverage metrics to ensure that the testbench achieves adequate functional coverage. Here are the different coverages that are part of the functional coverage:

1. **Clock and Reset Coverage Collector** - Tracks coverage for reset signals (rst_cvg) and clock signals (clk_cvg).
2. **Data Collector** - Monitors data-related activities and collects coverage metrics (letters_cvg).
3. **Control Collector** - Tracks control signal activities such as start and scoring signals (score_cvg and start_cvg).

A detailed functional coverage plan can be viewed under section 16.

13.3.3 Reference Model

Implements a golden model that predicts the correct output based on input sequences. The reference model implements the Smith Waterman Algorithm, which is used for local sequence alignment. The input 2 sequences, query sequence and database sequence, 32 letters each. The output of the golden model is as the output of the chip – the aligned sequences and the maximum score achieved.

13.3.4 Scoreboard

The Scoreboard is responsible for comparing the actual output of the DUT against the expected output generated by the SV Reference Model. It compares the aligned sequences and the output score from both. Two sequences are considered a Match only if both the alignment and the output score are identical. Otherwise, they are considered a Mismatch. A test involves multiple comparisons, and the scoreboard determines if the test passes. It passes only if all sequences match in that test.

13.3.5 Virtual Sequence

The Virtual Sequence coordinates and manages the execution of sequences across multiple agents. Its main responsibility is to ensure that sequences are aligned with each other (e.g., query and database inputs are synchronized).

13.4 Interface

The Interface provides the connection between the testbench and the DUT. It handles signal communication (e.g., data inputs, clock, reset, control signals) between the UVM components and the DUT.

13.5 Design Under Test (DUT)

The DUT (Design Under Test) in this case is the Smith-Waterman accelerator, which was implemented as part of Section A of the project. This accelerator is designed to perform local sequence alignment, a critical operation in bioinformatics used for comparing biological sequences such as DNA, RNA, or protein sequences.

13.6 Reference model and Scoreboard interaction

The Reference Model and the Scoreboard work together to validate the Design Under Test (DUT) by comparing the DUT's outputs with the expected results that the reference model provides. Here's how the interaction occurs:

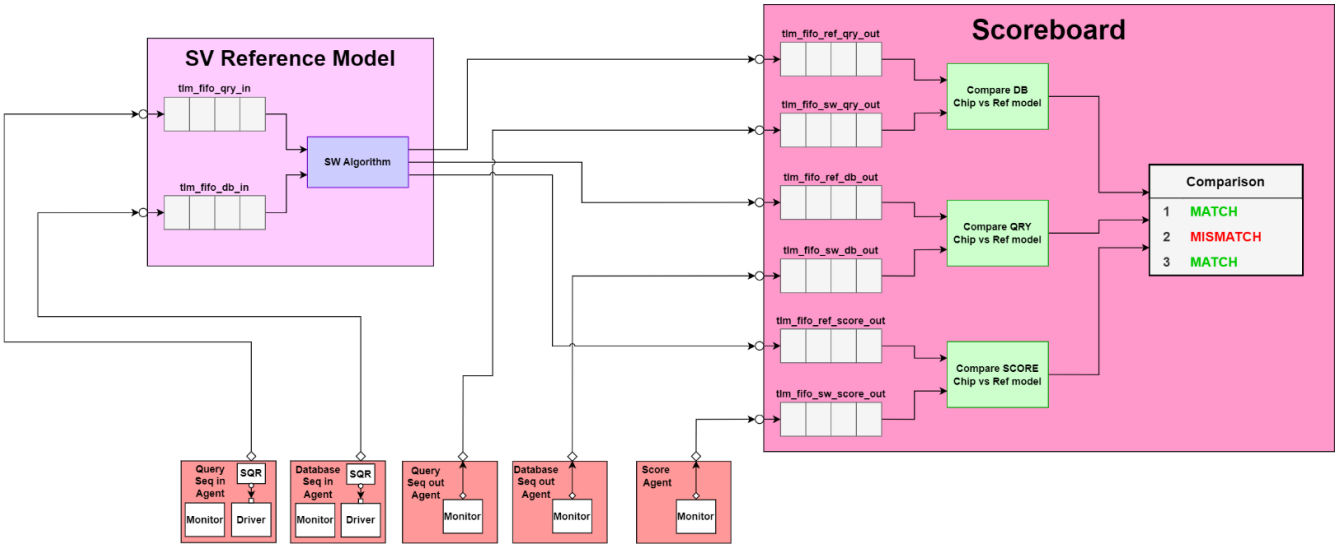


Figure 38: Ref Model and Scoreboard Interaction

As part of this verification flow, we leveraged the Transaction-Level Modeling (TLM) FIFOs provided by UVM to manage and preserve the order of transactions effectively. These FIFOs were instrumental in maintaining synchronization between two similar items being compared, ensuring that the verification environment could handle data coherently and consistently. By utilizing TLM FIFOs, we streamlined communication between components, allowing for precise alignment and comparison of transactions by keeping their order.

1. **Input Processing by the Reference Model**:

The Query (qry) and Database (db) inputs are fed into the Reference Model via transaction-level modeling FIFOs (tim_fifo_qry_in and tim_fifo_db_in). The Reference Model processes these inputs using the implemented SW Algorithm to produce expected outputs:

- ref_qry_out (expected Query output). It is sent to the tlm_fifo_ref_qry_out object in the scoreboard.
- ref_db_out (expected Database output) It is sent to the tlm_fifo_ref_qry_out object in the scoreboard.
- ref_score_out (expected Score output) It is sent to the tlm_fifo_ref_qry_out object in the scoreboard.

2. **Outputs from the DUT to the Scoreboard through Monitors**:

The DUT generates the following outputs, which are monitored and passed to the scoreboard:

- sw_qry_out (expected Query output). It is sent to the tlm_fifo_sw_qry_out object in the scoreboard.
- sw_db_out (expected Database output) It is sent to the tlm_fifo_sw_db_out object in the scoreboard.
- sw_score_out (expected Score output) It is sent to the tlm_fifo_sw_score_out object in the scoreboard.

3. **Comparison in the Scoreboard**:

The Scoreboard performs direct comparisons between DUT outputs (sw) and the Reference Model outputs (ref) for each packet:

- Query output comparison (qry)
- Database output comparison (db)
- Score output comparison (score)

The scoreboard determines the result of the three comparisons as either a Match or a Mismatch. A result is classified as a Match only if both the alignment and the output score from the comparison between the reference model and the DUT are identical. Any discrepancy results in a Mismatch.

14 Test Plan Overview

This test plan outlines a series of test classes that inherit from a base test class. The base test class generates a clock with a 50% duty cycle, provides a reset signal with fixed latency, and triggers the start signal after a fixed delay. Additionally, packets are inserted completely randomly (probability variable is set to 0%).

Each of the derived test classes introduces specific configurations or behaviors, such as controlling the clock, reset, or start signal, as well as additional functionalities like random sequence insertion and sequence similarity control. Here is a detailed overview of the tests:

14.1 Base_test

This is the base test class from which all other tests inherit. It extends the uvm_test. This test activates the virtual sequence, runs the algorithm of the reference model and performs the comparison function from the scoreboard described under Section 13. To proceed with the test, it raises an uvm objection and drops it when the test is done.

It generates:

- A clock signal with a 50% duty cycle.
- A reset signal with a fixed latency.
- A start signal that is triggered after a fixed delay.
- Randomly inserted packets.

Here is a test behavior example:

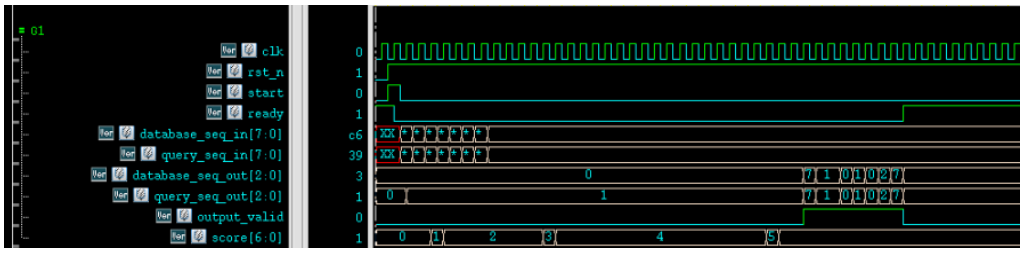


Figure 39: base_test behavior

We can observe the basic behavior – 50% duty cycle, reset with a fixed delay, start rises before sequences are inserted and packets are randomly generated.

14.2 Clk_test

This test extends the base_test and adds control over the clock behavior, allowing the definition of the duty cycle, clock period, and clock gating. It utilizes a clk_test_cfg object to set all configurations. While maintaining the same behavior as the base_test - including fixed reset and start signals with randomly inserted sequence pairs - it also provides precise control over the clock signal.

Here is a test behavior example:

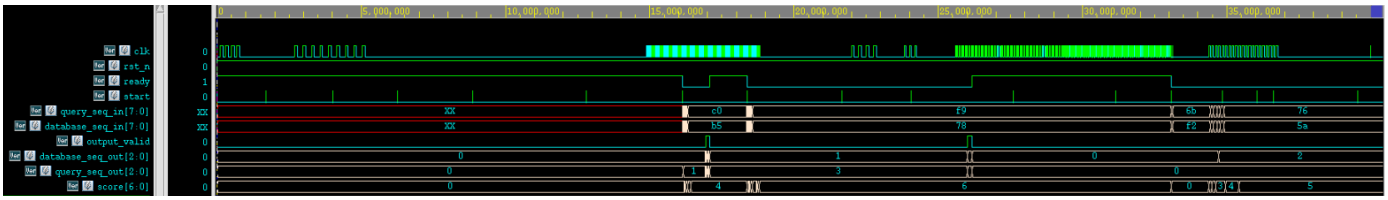


Figure 40: clk_test behavior

We can see that clock is gated and has different duty cycle and period.

14.3 Rst_test

This test extends the base_test and adds control over the reset behavior, allowing the definition of the reset duration. It utilizes a rst_test_cfg object to set all configurations. While maintaining the same behavior as the base_test - including fixed clock and start signals with randomly inserted sequence pairs - it also provides precise control over the reset signal.

Here is a test behavior example:

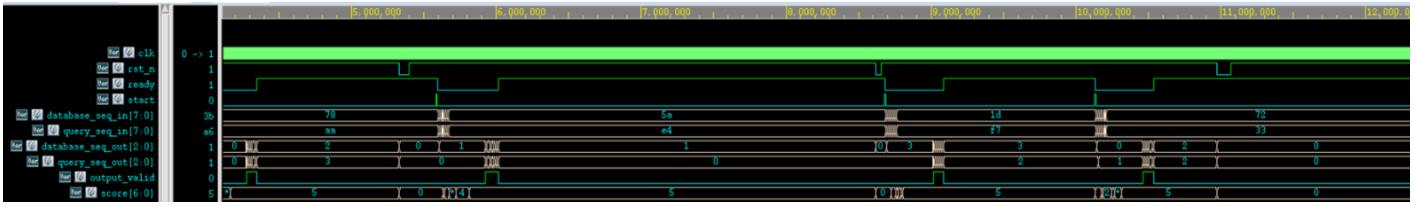


Figure 41: rst_test behavior

We can see that reset(rst_n signal) has different durations.

14.4 Clk_rst_test

This test extends the base_test and adds control over the reset and the clock as described above. It utilizes both rst_test_cfg and clk_test_cfg object to set all configurations. While maintaining the same behavior as the base_test - including start signals with randomly inserted sequence pairs - it also provides precise control over the reset and clock signals.

Here is a test behavior example:

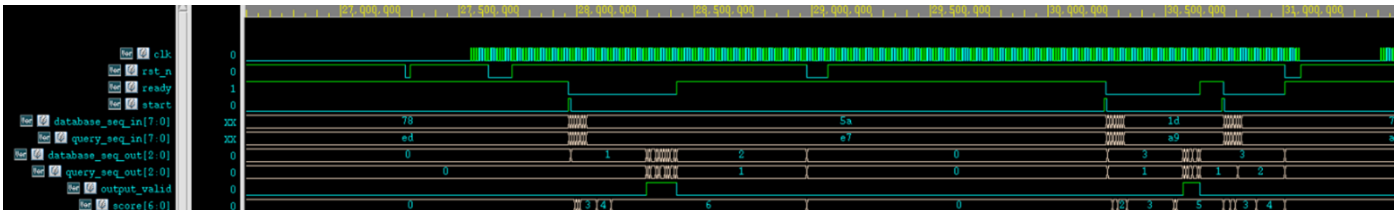


Figure 42: clk_rst_test behavior

We can see that reset(rst_n signal) has different durations and clock is gated.

14.5 Start_test

This test extends the base_test and adds control over the start behavior, enabling the definition of the start signal's duration and timing. It utilizes a start_test_cfg object to set all configurations. While maintaining the same behavior as the base_test - including fixed clock and reset signals with randomly inserted sequence pairs - it also provides precise control over the start signal.

Here is a test behavior example:

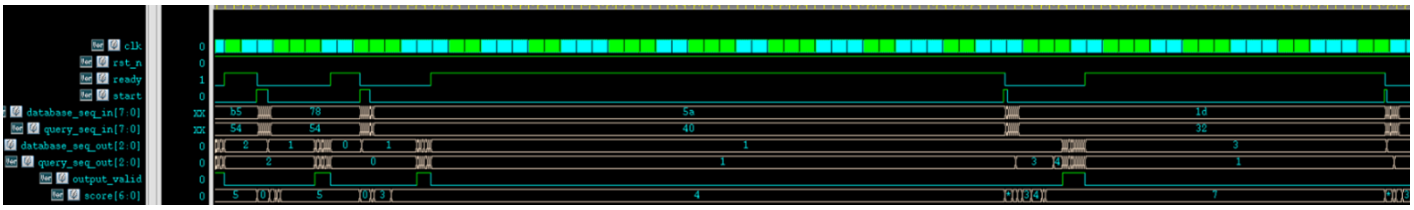


Figure 42: start_test behavior

We can see that start signal has different durations and it appears in random times during the simulation.

14.6 Letters_test

This test extends the base_test. It introduces full control over the content of the inserted sequences. It uses a lettters_test_cfg object to configure all parameters. This test is designed for additional sequence testing or as a basis for more complex behaviors, allowing precise control over inserted data.

How are Letters generated? Here are the steps:

1. Letters are generated per packet, with 4 letters produced at a time.
2. First, 4 letters are generated for the query sequence.
3. A 4-bit mask is then created using a probability variable, where each bit corresponds to a letter in both sequences.
4. The probability variable determines the likelihood of matching letters between the query and database sequences, ranging from 0 to 100. A value of 0 generates letters randomly, while 100 ensures identical letters in corresponding positions.

5. Finally, the database sequence is generated based on the mask and the query sequence.

While maintaining the same behavior as the base_test - including fixed clock, reset and start signals, it provides precise control over the inserted sequences – the query and the database sequences.

Here is an example:

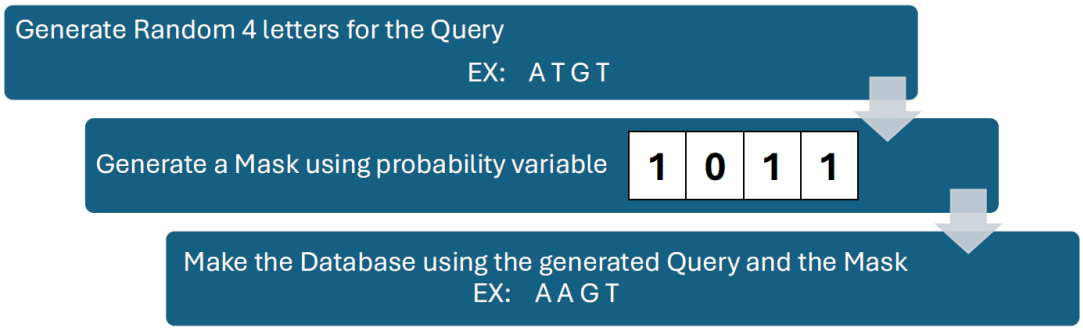


Figure 43: letters generation

In this example, we first generated the 4 letters: A, T, G, T for the query sequence. Next, a probability value (ranging from 0% to 100%) was chosen for the mask based on the test configuration. The resulting mask was 1011.

According to the query letters and the mask, only the letters in positions marked by 1 in the mask were copied to the database sequence. As a result, the database sequence became A, A, G, T. Here, the letters A, G, and T were copied directly from the query sequence, while the second A (from the left) was generated randomly.

Here is a test behavior example:

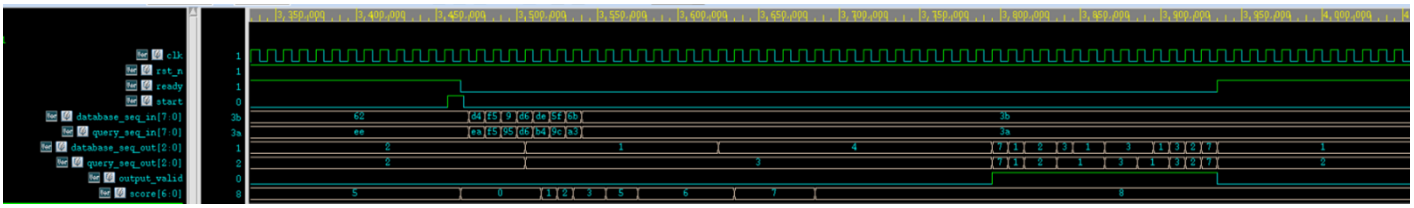


Figure 44: letters_test behavior

By looking at the database_seq_in and query_seq_in, We can see that some of the generated sequences based on the mask (probability variable with the value of 70%) are the same and some are completely different.

15 SVA and behavior Plan

System Verilog assertions play a crucial role in the verification phase of chip design by enabling engineers to specify expected behaviors and detect design flaws early in the process. Assertions are used to monitor and verify the functionality of a design under different conditions, ensuring that the chip behaves as intended. They can be applied to check properties such as signal consistency and state transitions within the design. By embedding assertions directly in the design or testbenches, engineers can automatically capture violations, simplifying the process of identifying bugs. This proactive approach not only helps in finding issues that may otherwise be overlooked during traditional simulation but also improves overall design quality and reduces verification time.

We implemented System Verilog Assertions (SVAs) on the interface. These SVAs ensure that the interface behavior aligns with the specifications outlined in this file. We developed the if_checker, which incorporates the following SVAs:

- score did not change while output valid is high.
- while output valid is high, the output letters are not Xs.
- ready start handshake – if a handshake occurred, the ready deasserts in the next cycle.
- After a handshake occurred, the output score resets to zero.
- letters are inserted maximum one cycle after the handshake occurs.

16 Coverage Plan

This plan outlines the different coverage parameters for the various test scenarios. Each parameter represents a specific aspect of the test conditions that will be monitored and evaluated.

1. **Letters Coverage**
   - **Range**: 0 to 32
   - **Description**: This measures the similarity between sequences based on the number of identical letters.
     - **0**: Represents completely different sequences.
     - **32**: Represents identical sequences.
2. **Start Signal Duration Coverage**
   - **Range**: 10 ns to 100 ns
   - **Description**: This monitors the duration for which the start signal is active.
3. **Score Coverage**
   - **Range**: 0 to 32
   - **Description**: This covers the score values, which may be used to assess the sequence alignment similarity. High score means that they are close to identical, while low score means that they are completely distinct.
4. **Reset Signal Duration Coverage**
   - **Range**: 10 ns to 100 ns
   - **Description**: This measures the duration of the reset signal during testing.
5. **Clock Duty Cycle Coverage**
   - **Range**: 30% to 70%
   - **Description**: This covers the duty cycle of the clock signal, indicating the percentage of time the clock is active during a period.
6. **Clock Period Coverage**
   - **Range**: 2 ns to 10 ns
   - **Description**: This monitors the period of the clock signal, specifying how often the clock pulses occur.
7. **Clock On/Off Coverage**
   - **Values**: 0 or 1
   - **Description**: This tracks whether the clock signal is on (1) or off (0) during the test.

17 Coverage Results

As part of the full coverage, we verified the following items:

1. **Group coverage** – Verifies all functional groups and scenarios in the coverage plan.
2. **Line coverage** – Ensures every line of code is executed during testing.
3. **Condition coverage** – Tests all Boolean expressions for true and false values.
4. **Finite state machine coverage** – Ensures all states and transitions in finite state machines are tested.
5. **Branch coverage** - Verifies all decision points are exercised for both taken and not-taken paths.

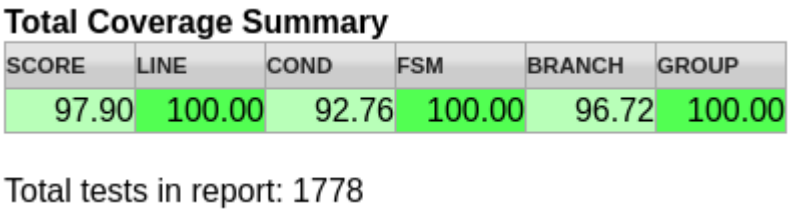


Figure 45: coverage results

We can see that we achieved the following results:

1. **LINE:** 100% of the code lines were executed during testing.
2. **COND:** 92.76% of the conditional logic has been tested.
3. **FSM:** 100% of the finite state machine's states and transitions have been covered.
4. **BRANCH:** 96.72% of all the possible branches of the code have been taken.
5. **GROUP:** 100% of grouped conditions or functionalities specified for testing have been covered.

We achieved full functional coverage, as well as complete coverage for the code and finite state machine states and transitions. However, full coverage was not achieved for condition and branch coverages. In the following chapters, we will analyze the results and draw conclusions for all coverage items.

18 Code Coverage

Here are the code coverage results for different sections within the design, showing the extent of their testing. Some sections have full coverage, while others remain uncovered. The uncovered sections are a result of the design implementation, and we will explain why they were not covered.

NAME	SCORE	LINE	COND	FSM	BRANCH
l_controller	97.50	100.00	93.33	100.00	96.67
l_flipflop	100.00	100.00	100.00		100.00
⊟ l_matrix_calculation	92.10	100.00	86.83		89.47
⊞ PU[0].pu_inst	90.85		90.79		90.91
⊞ PU[10].pu_inst	88.88		86.84		90.91
⊞ PU[11].pu_inst	88.88		86.84		90.91
⊞ PU[12].pu_inst	88.22		85.53		90.91
⊞ PU[13].pu_inst	87.56		84.21		90.91
⊞ PU[14].pu_inst	81.52		78.95		84.09
⊞ PU[15].pu_inst	69.77		69.08		70.45
⊞ PU[1].pu_inst	91.18		91.45		90.91
⊞ PU[2].pu_inst	90.52		90.13		90.91
⊞ PU[3].pu_inst	90.19		89.47		90.91
⊞ PU[4].pu_inst	90.19		89.47		90.91
⊞ PU[5].pu_inst	90.19		89.47		90.91
⊞ PU[6].pu_inst	90.19		89.47		90.91
⊞ PU[7].pu_inst	90.19		89.47		90.91
⊞ PU[8].pu_inst	89.53		88.16		90.91
⊞ PU[9].pu_inst	88.88		86.84		90.91
l_matrix_memory	100.00	100.00	100.00		100.00
⊞ l_max_registers	100.00	100.00	100.00		100.00
l_sequence_buffer	94.17	100.00	82.50		100.00
l_traceback	100.00	100.00	100.00		100.00

Figure 46: code coverage

18.1 Sequence Buffer

Let's review the conditions of the sequence buffers that were not satisfied:

```
LINE       53
EXPRESSION (reg_enable[1] && wr_en_buff)
            ------1------    -----2----
```

-1-	-2-	Status
0	1	Covered
1	0	Not Covered
1	1	Covered

Figure 47: sequence buffer unsatisfied condition

This example represents one of 8 scenarios that repeat. The case for all i from 0 to 8 (where 'reg_enable[i]' = 1 and 'we_en_buff' = 0) is not covered and will never be. This is duo to the fact that the **'we_en_buff'** signal is responsible for setting the **'reg_enable[i]'** to 1 for the matching bit. Thefore, neither of the bits in **'reg_enable'** will set to 1 when **'we_en_buff'** is not active.

18.2 Controller

Here we will review the conditions and branches of the controller that were not satisfied:

Branches:

-1-	-2-	-3-	-4-	-5-	-6-	Status
IDLE_ST	1	-	-	-	-	Covered
IDLE_ST	0	-	-	-	-	Covered
LOAD_SEQUENCES_ST	-	1	-	-	-	Covered
LOAD_SEQUENCES_ST	-	0	-	-	-	Covered
LOAD_SEQUENCES_ST	-	-	1	-	-	Covered
LOAD_SEQUENCES_ST	-	-	0	-	-	Covered
SCORES_CALC_ST	-	-	-	1	-	Covered
SCORES_CALC_ST	-	-	-	0	-	Covered
TRACEBACK_ST	-	-	-	-	1	Covered
TRACEBACK_ST	-	-	-	-	0	Covered
MISSING_DEFAULT	-	-	-	-	-	Not Covered

Figure 48: controller unsatisfied condition

The controller is missing the default value explicitly in the FSM case. However, it is set to be the previous state, therefore it is not considered a problem.

Here is a snippet of this part in the code:

```
/*=============================FSM LOGIC=============================*/
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        CUR_ST <= IDLE_ST;
    end
    else begin
        CUR_ST <= NEXT_ST;
    end
end

always_comb begin
    // Default values
    ready = 1'b0;
    wr_en_buff = 1'b0;
    count_buff = '0;
    wr_en_max = 1'b0;
    en_traceback = 1'b0;
    output_valid = 1'b0;
    busy = 1'b1;
    NEXT_ST = CUR_ST;

    case (CUR_ST)
        IDLE_ST:
        begin
            ready = 1'b1;
            busy = 1'b0;
            if (start == 1'b1) begin
                NEXT_ST = LOAD_SEQUENCES_ST;
            end
        end

        LOAD SEQUENCES ST:
```

We can see in the code that the default case is implemented: the next state becomes the current state and nothing changes, as expected.

18.3 Matrix Calculation

Here we will review the conditions and branches of the Matrix calculation that were not satisfied**:**

```
LINE       54
SUB-EXPRESSION BIT 5 of (  score )
                           --1--
```

-1-	Status
0	Covered
1	Not Covered

```
LINE       54
SUB-EXPRESSION BIT 4 of (  score )
                           --1--
```

-1-	Status
0	Covered
1	Not Covered

Figure 49: Matrix calculation unsatisfied conditions

In all PUs, based on the parameters and algorithm, we cannot exceed a certain value for score. Let's for example check PU number 1. This PU is responsible for the values across the first column:

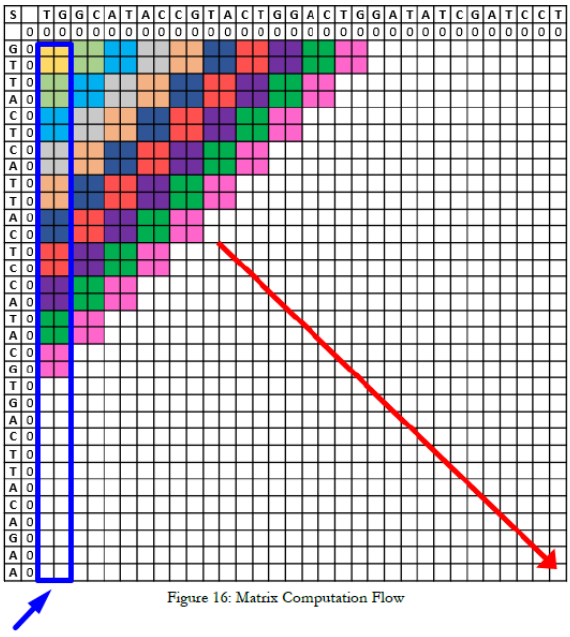


Figure 50: values calculated across first column

For score, to set bits number 4 and 5 to be 1, we need at least to see the values 16 to 25 on the first column which is not possible in the smith waterman algorithm using those parameters.

By examining the code coverage, we can see that PU number 15 has significantly lower coverage compared to the other PUs.

```
LINE       41
EXPRESSION ((top > design_variables::GAP_PENALTY) ? ((top - design_variables::GAP_PENALTY)) : '0)
            -----------------1-----------------
```

-1-	Status
0	Covered
1	Not Covered

Figure 51: PU number 15 unsatisfied condition

The reason for that is, by looking at PU number 15, and his area of responsibility, we can tell that the result coming from the top and diagonal cannot satisfy the condition based on the algorithm that updates the matrix where there is a mismatch. Only the result coming from the left side can satisfy the condition in some cases.

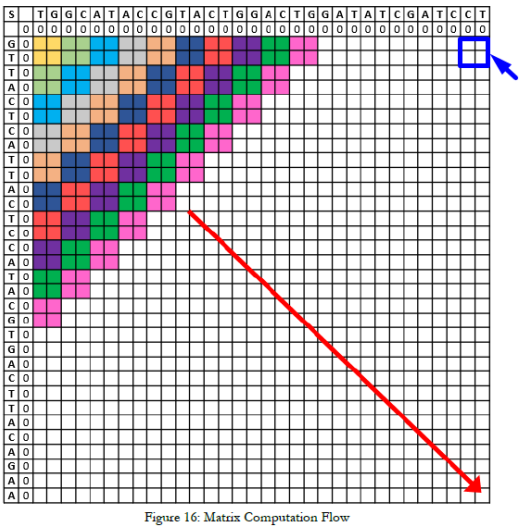


Figure 52: The area that PU is affected by

19 Functional Coverage

Total Groups Coverage Summary

SCORE	WEIGHT
100.00	1

Total groups in report: 5

NAME	SCORE	WEIGHT	GOAL	AT LEAST	PER INSTANCE	AUTO BIN MAX	PRINT MISSING	COMMENT
$unit::rst_cvg#(10,100)::rst_cg	100.00	1	100	1	0	64	64	
$unit::start_cvg#(10,100)::start_cg	100.00	1	100	1	0	64	64	
$unit::letters_cvg#(32)::letters_cg	100.00	1	100	1	0	64	64	
$unit::score_cvg#(32)::score_cg	100.00	1	100	1	0	64	64	
$unit::clk_cvg#(2,10,70,30)::clk_cg	100.00	1	100	1	0	64	64	

Figure 53: functional coverage

This result demonstrates that all the groups in our coverage plan, as mentioned below, have been covered. We assigned a weight of 1 to each group to ensure equal emphasis on their importance.

Here is a summary of the coverage information for each group:

19.1 Letters Coverage

Summary for Variable num_similar_letters

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	33	0	33	100.00

User Defined Bins for num_similar_letters

Bins

NAME	COUNT	AT LEAST
num_similar_letters_bins_0	666	1
num_similar_letters_bins_1	139	1
num_similar_letters_bins_2	371	1
num_similar_letters_bins_3	824	1
num_similar_letters_bins_4	2186	1
num_similar_letters_bins_5	1875	1
num_similar_letters_bins_6	2431	1
num_similar_letters_bins_7	2594	1
num_similar_letters_bins_8	3324	1
num_similar_letters_bins_9	2415	1
num_similar_letters_bins_10	1969	1
num_similar_letters_bins_11	1535	1
num_similar_letters_bins_12	1835	1
num_similar_letters_bins_13	871	1
num_similar_letters_bins_14	622	1
num_similar_letters_bins_15	491	1
num_similar_letters_bins_16	713	1
num_similar_letters_bins_17	293	1
num_similar_letters_bins_18	230	1
num_similar_letters_bins_19	196	1
num_similar_letters_bins_20	331	1
num_similar_letters_bins_21	190	1
num_similar_letters_bins_22	169	1
num_similar_letters_bins_23	189	1

Figure 54: letters coverage

We can see the distribution based on the probability variable. We set the probability in regular tests to be somewhere in the 0% to 40% range of similarity, in which all the interesting cases occur.

19.2 Clock Coverage

Summary for Variable clk_period_cp

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	9	0	9	100.00

User Defined Bins for clk_period_cp

Bins

NAME	COUNT	AT LEAST
clk_period_bins_2	314247	1
clk_period_bins_3	319878	1
clk_period_bins_4	393595	1
clk_period_bins_5	450937	1
clk_period_bins_6	428842	1
clk_period_bins_7	454028	1
clk_period_bins_8	428788	1
clk_period_bins_9	394313	1
clk_period_bins_10	11646789	1

Summary for Variable clk_on_cp

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	2	0	2	100.00

User Defined Bins for clk_on_cp

Bins

NAME	COUNT	AT LEAST
clk_on_bins_0	7418653	1
clk_on_bins_1	7418770	1

Summary for Variable duty_cycle_cp

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	41	0	41	100.00

User Defined Bins for duty_cycle_cp

Bins

NAME	COUNT	AT LEAST
duty_cycle_bins_30	4564	1
duty_cycle_bins_31	29782	1
duty_cycle_bins_32	17533	1
duty_cycle_bins_33	23222	1
duty_cycle_bins_34	21021	1
duty_cycle_bins_35	34873	1
duty_cycle_bins_36	23272	1
duty_cycle_bins_37	23918	1
duty_cycle_bins_38	40193	1
duty_cycle_bins_39	42217	1
duty_cycle_bins_40	58954	1
duty_cycle_bins_41	40095	1
duty_cycle_bins_42	74165	1
duty_cycle_bins_43	73530	1
duty_cycle_bins_44	67485	1
duty_cycle_bins_45	53578	1
duty_cycle_bins_46	81778	1
duty_cycle_bins_47	87353	1
duty_cycle_bins_48	91555	1
duty_cycle_bins_49	88543	1
duty_cycle_bins_50	11915136	1

Figure 55: clk coverage

We can see that in regular tests, the duty cycle and clock period were set to 50% and 10ns respectively. This is why their count is the highest. We tried to balance gating the clock – from keeping it on and turning it off.

19.3 Reset Coverage

Summary for Variable rst_duration_time_cp

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	91	0	91	100.00

User Defined Bins for rst_duration_time_cp

Bins

NAME	COUNT	AT LEAST
duration_time_ns_bins_10	4984	1
duration_time_ns_bins_11	13	1
duration_time_ns_bins_12	3	1
duration_time_ns_bins_13	2	1
duration_time_ns_bins_14	12	1
duration_time_ns_bins_15	15	1
duration_time_ns_bins_16	18	1
duration_time_ns_bins_17	12	1
duration_time_ns_bins_18	12	1
duration_time_ns_bins_19	22	1
duration_time_ns_bins_20	15	1
duration_time_ns_bins_21	19	1
duration_time_ns_bins_22	26	1
duration_time_ns_bins_23	25	1
duration_time_ns_bins_24	29	1
duration_time_ns_bins_25	20	1
duration_time_ns_bins_26	34	1
duration_time_ns_bins_27	27	1
duration_time_ns_bins_28	23	1
duration_time_ns_bins_29	32	1
duration_time_ns_bins_30	25	1
duration_time_ns_bins_31	29	1
duration_time_ns_bins_32	26	1
duration_time_ns_bins_33	19	1

Figure 56: rst coverage

We can see that the reset duration in regular tests was set to 10ns. This is why its count is the highest.

19.4 Start Coverage

Summary for Variable start_duration_time_cp

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	91	0	91	100.00

User Defined Bins for start_duration_time_cp

Bins

NAME	COUNT	AT LEAST
duration_time_ns_bins_10	30414	1
duration_time_ns_bins_11	4	1
duration_time_ns_bins_12	5	1
duration_time_ns_bins_13	4	1
duration_time_ns_bins_14	8	1
duration_time_ns_bins_15	5	1
duration_time_ns_bins_16	6	1
duration_time_ns_bins_17	6	1
duration_time_ns_bins_18	3	1
duration_time_ns_bins_19	7	1
duration_time_ns_bins_20	19	1
duration_time_ns_bins_21	7	1
duration_time_ns_bins_22	15	1
duration_time_ns_bins_23	8	1
duration_time_ns_bins_24	13	1
duration_time_ns_bins_25	11	1
duration_time_ns_bins_26	8	1
duration_time_ns_bins_27	18	1
duration_time_ns_bins_28	27	1
duration_time_ns_bins_29	11	1
duration_time_ns_bins_30	21	1
duration_time_ns_bins_31	15	1
duration_time_ns_bins_32	19	1
duration_time_ns_bins_33	25	1

Figure 57: start coverage

We can see that the start duration in regular tests was set to 10ns. This is why its count is the highest.

19.5 Score Coverage

Summary for Variable score_cp

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	33	0	33	100.00

User Defined Bins for score_cp

Bins

NAME	COUNT	AT LEAST
score_bins_0	145	1
score_bins_1	1110	1
score_bins_2	1047	1
score_bins_3	536	1
score_bins_4	6173	1
score_bins_5	8126	1
score_bins_6	4450	1
score_bins_7	1823	1
score_bins_8	742	1
score_bins_9	335	1
score_bins_10	204	1
score_bins_11	164	1
score_bins_12	141	1
score_bins_13	102	1
score_bins_14	126	1
score_bins_15	146	1
score_bins_16	432	1
score_bins_17	257	1
score_bins_18	118	1
score_bins_19	53	1
score_bins_20	107	1
score_bins_21	64	1
score_bins_22	126	1
score_bins_23	40	1

Figure 58: score coverage

We can see that all the possible scores occurred in our coverage, from 0 to 32. We can observe that most of the low scores have a higher counter than others. This is because most of the tests generated random inputs, which means that the sequences are different. Different sequences mean low score – they have low number of similarities.

20 Regression Flow

The goal of regression is to run multiple tests covering various test cases to identify issues or bugs in the design. It systematically validates different scenarios by randomly selecting, configuring, and running tests automatically. During regression, each test is built, simulated, and its coverage results are collected for analysis. The regression flow is described below:

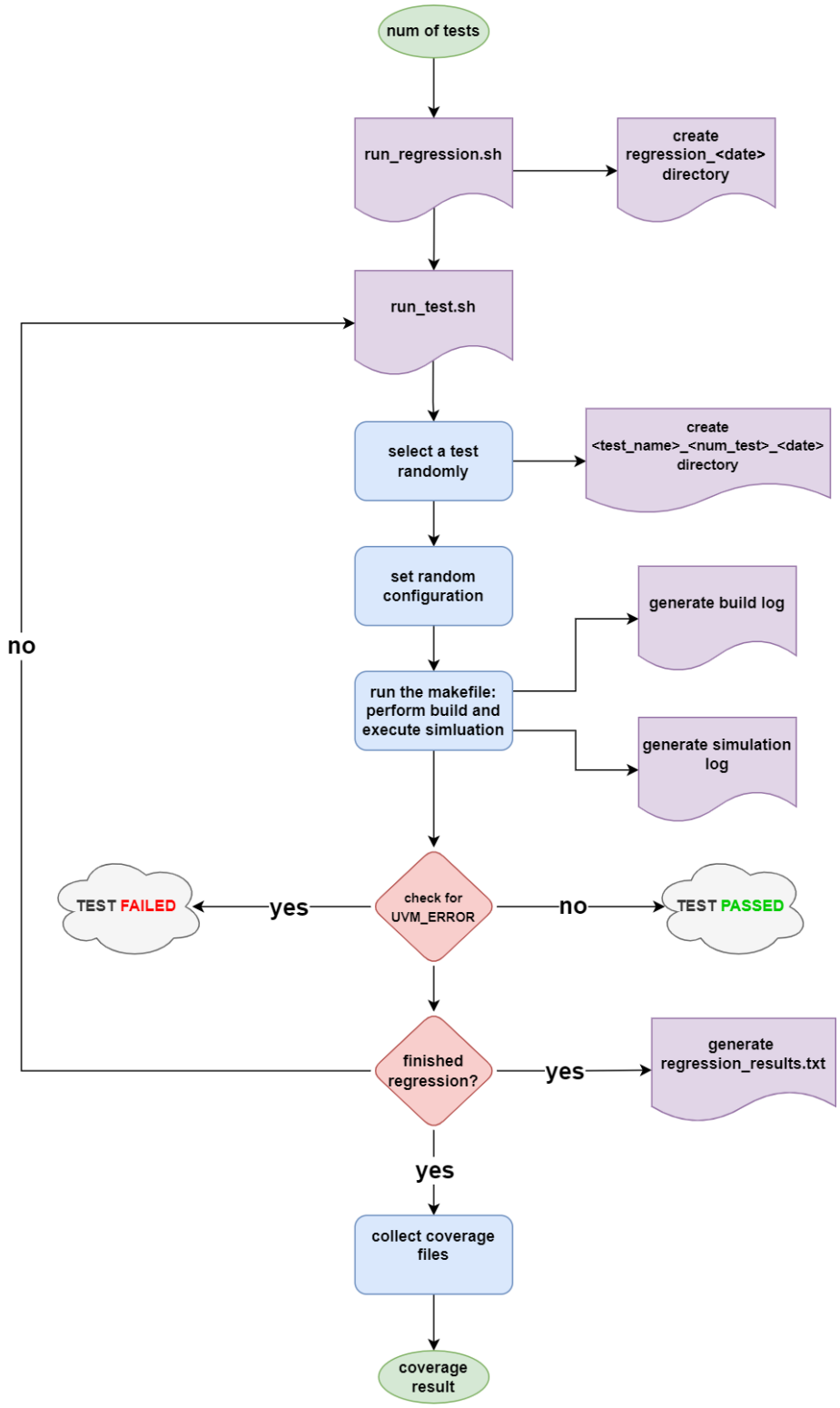


Figure 59: regression flow

Here's a breakdown of the flow:

1. **Initialization:** The process begins with determining the number of tests to run and initializing the run_regression.sh script.
2. **Regression Directory Creation**: A regression directory with the current date is created for organizing the run.
3. **Running run_test.sh Script**: For each test, the run_test.sh script is executed to manage the individual test processes.
4. **Test Selection**: Within run_test.sh, a test is selected randomly from the available tests. A directory for the selected test is created in the regression directory.
5. **Configuration and Build**: A random configuration is set for the test. The test's Makefile is executed to build and run the simulation. A random Seed is chosen based on the entropy of the system.
6. **Log Generation**: Build logs and simulation logs are generated during this step for tracking purposes.
7. **Error Checking**: After running the simulation, the script checks for the presence of UVM_ERROR in the logs. If UVM_ERROR is detected, the test fails, and the result is logged. If no error is found, the test passes, and the result is logged.
8. **Post-Test Processing:** Logs are saved, and the results for each test are recorded in regression_results.txt.
9. **Regression Completion:** Once all tests are complete, the script collects coverage files from all test runs.
10. **Coverage Results**: Finally, a coverage report is generated, summarizing the coverage metrics for the regression run.

20.1 Regression Usage

To execute the regression script in the current directory, use the following command:
run_regression.sh <number_of_tests>

For example:
run_regression.sh 1800 &

To view the coverage report, open the following file in your browser:
firefox ./regression_<date>/coverage_report/dashboard.html

21 Simulation and Tests Results

Each night we run a nightly run with around 1800 tests. The tests were randomly selected with a random configuration matching the test and a random SEED number. The seed number and test log helped us debug the run. If we wanted to review and debug, we opened the log, retrieved the configuration parameters and ran the test with a FSDB output.

Below is a summary snippet of the run:

Test Name	Iteration	SEED	Result
clk_rst_test	1	259474551	FAILED
base_test	2	286153340	PASSED
rst_test	3	9577960	PASSED
letters_test	4	103369352	FAILED
start_test	5	338881212	FAILED
start_test	6	272798280	FAILED
rst_test	7	350774435	FAILED
clk_rst_test	8	600940269	FAILED
letters_test	9	83941806	FAILED
start_test	10	562085132	PASSED
base_test	11	660363445	FAILED
letters_test	12	194088076	FAILED
clk_test	13	75426766	PASSED
letters_test	14	819356032	FAILED
start_test	15	151367553	FAILED
clk_rst_test	16	358787070	PASSED
letters_test	17	8058344	FAILED
start_test	18	68464560	PASSED
clk_test	19	36456620	FAILED

Figure 60: regression_results.txt

While running the regression, we found a few bugs:

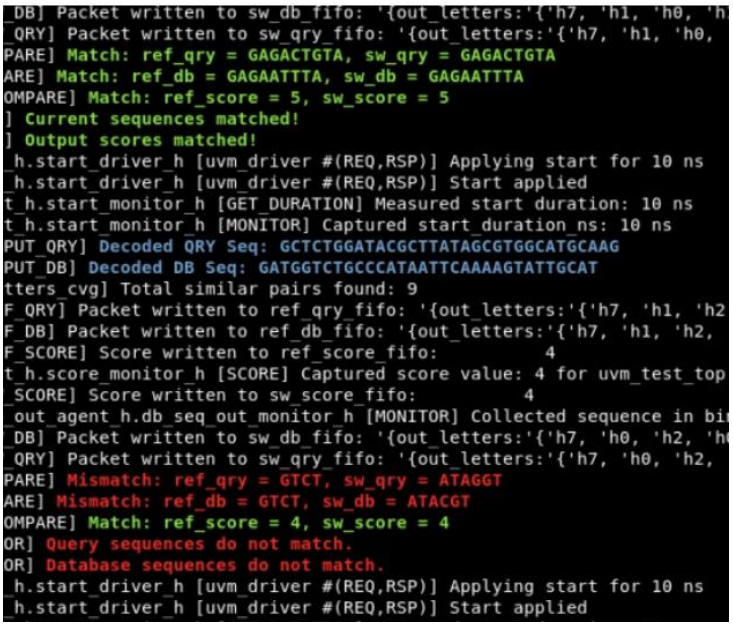


Figure 61: traceback bug

We observed that the traceback process is not correct in some cases. Although the maximum score remains the same, the alignment is incorrect.

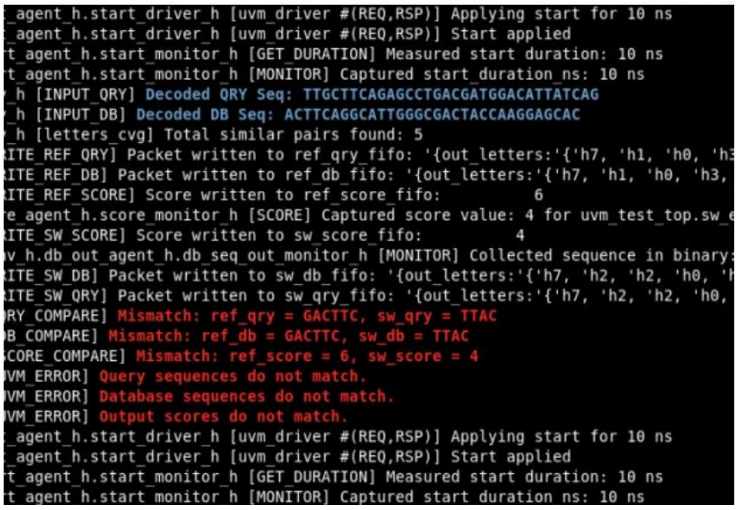


Figure 62: Calculation process bug

We found that the calculation process is wrong in some cases. We can observe this by looking at the max score we got from the reference model and the chip.

22 Verification Analysis and Conclusions

The SW accelerator presented a flawless design combining an effective coding approach with accurate functionality. However, there are a few things to consider:

- **Redundant Code**: The matrix calculation coverage highlights sections of the code where the design will never satisfy certain conditions. These parts result in redundant hardware that serves no functional purpose. To optimize area and power, it is advisable to exclude these sections from synthesis, ensuring they are ignored during the hardware generation process.

- **Incomplete Functionality**: Verification has uncovered corner cases where the design does not perform as intended. Although the design generally behaves as expected, issues arise in specific scenarios, particularly during the calculation or traceback processes.

- **Corner Case Definitions and Design Clarity**: While the overall design is robust, certain corner cases lack clear definitions or are ambiguously specified. This gap makes it challenging to determine the intended behavior of the design under rare input conditions or atypical scenarios. Providing comprehensive and detailed definitions for these edge cases will enhance the clarity of the design, ensure proper handling of all scenarios, and facilitate smoother verification and debugging processes.

SECTION D:
Backend Design

In fulfillment of the requirements for Project B (044169)

23 Main Stages

Figure 63 shows the main stages we performed in the backend design. These stages will be described in detail on this section.

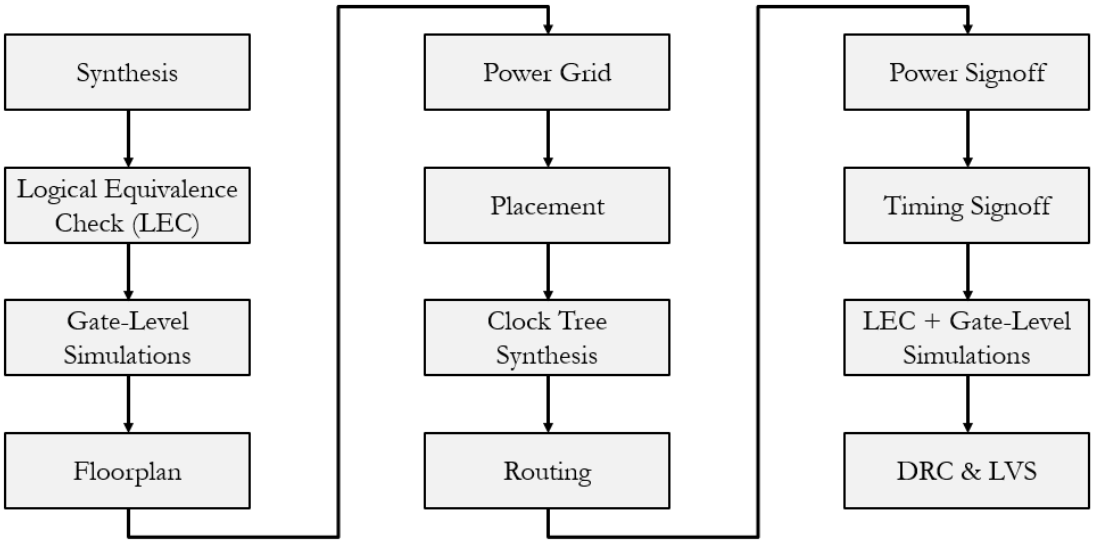


Figure 63: Main Backend Stages

24 Synthesis

Synthesis is the process of converting RTL code into an optimized gate-level netlist, targeting the technology process while ensuring compliance with area, timing, and power constraints. The logic synthesis flow described in figure 64.

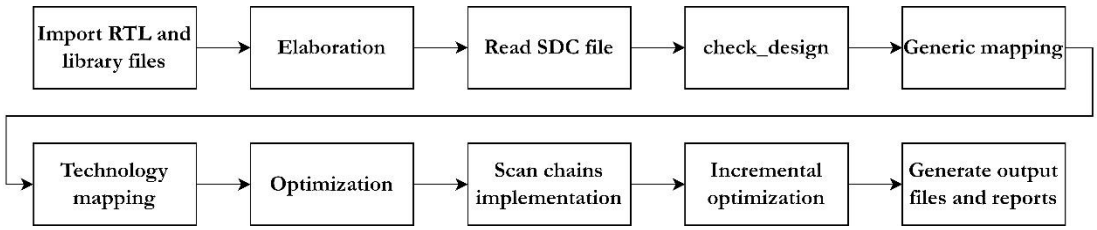


Figure 64: Logic Synthesis Flow

We used Design Compiler tool by Synopsys to perform the synthesis with TSMC 65nm technology library (tcbn65lplvttc1d0.db library) and DesignWare IPs library (dw_foundation.sldb library). We defined the following constraints:

- **Clock frequency of 125 [Mhz]**: The actual work frequency of the chip is 100 [Mhz], but performing the optimizations with a higher frequency helped us to maximize the timing results and pass the timing signoff with a higher work frequency.
- **High mapping effort**: It is crucial to make the most out of the tool and get better timing and power results. We performed it using '`compile -map_effort high`' command.

- **2 scan chains**: Each scan chain has a single input and a single output, both enabled using a single pin, scan_en.
- **Fix multiple port nets**: Prevent direct connection between I/O ports (we had 'VDD' and 'VSS' connected directly to multiple signals).
- **Disable of register merging**: It was easier for LEC to pass.
- **Deletion of unloaded sequential cells**.

During the synthesis stage, we noticed that the tool interpreted the RTL code of the Max Registers unit differently than what we intended. The computation of the maximum cell performed serially, so we had a serious timing problem. The critical timing path was dependent on the Max Registers unit. We identified the problem and fixed the RTL code so that the computation will be performed in parallel. Following the fix, we were able to increase the frequency dramatically.

At the end of the process, we got the synthesized netlist, SDC file, scandef file, as well as synthesis reports. Here is the QoR report we got after the synthesis process:

```
****************************************
Report : qor
Design : top
Version: R-2020.09-SP2
Date   : Sat Mar  2 10:01:14 2024
****************************************

  Timing Path Group 'clk'
  -----------------------------------
  Levels of Logic:               65.00
  Critical Path Length:           6.56
  Critical Path Slack:            1.36
  Critical Path Clk Period:       8.00
  Total Negative Slack:           0.00
  No. of Violating Paths:         0.00
  Worst Hold Violation:           0.00
  Total Hold Violation:           0.00
  No. of Hold Violations:         0.00
  -----------------------------------

  Cell Count
  -----------------------------------
  Hierarchical Cell Count:         238
  Hierarchical Port Count:       12200
  Leaf Cell Count:               24286
  Buf/Inv Cell Count:             4658
  Buf Cell Count:                 1934
  Inv Cell Count:                 2724
  CT Buf/Inv Cell Count:             0
  Combinational Cell Count:      20266
  Sequential Cell Count:          4020
  Macro Count:                       0
  -----------------------------------

  Area
  -----------------------------------
  Combinational Area:    47467.801250
  Noncombinational Area: 52544.880710
  Buf/Inv Area:           5745.240227
  Total Buffer Area:          2802.96
  Total Inverter Area:        2942.28
  Macro/Black Box Area:      0.000000
  Net Area:                  0.000000
  -----------------------------------
  Cell Area:             100012.681959
  Design Area:           100012.681959

  Design Rules
  -----------------------------------
  Total Number of Nets:          24925
  Nets With Violations:              0
  Max Trans Violations:              0
  Max Cap Violations:                0
  -----------------------------------

  Hostname: vlsi-ria67

  Compile CPU Statistics
  ----------------------------------------
  Resource Sharing:                  27.37
  Logic Optimization:                 8.98
  Mapping Optimization:              14.49
  ----------------------------------------
  Overall Compile Time:              59.62
  Overall Compile Wall Clock Time:   60.16

  ------------------------------------------------
  Design  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0

  Design (Hold)  WNS: 0.00  TNS: 0.00  Number of Violating Paths: 0
  ------------------------------------------------
```

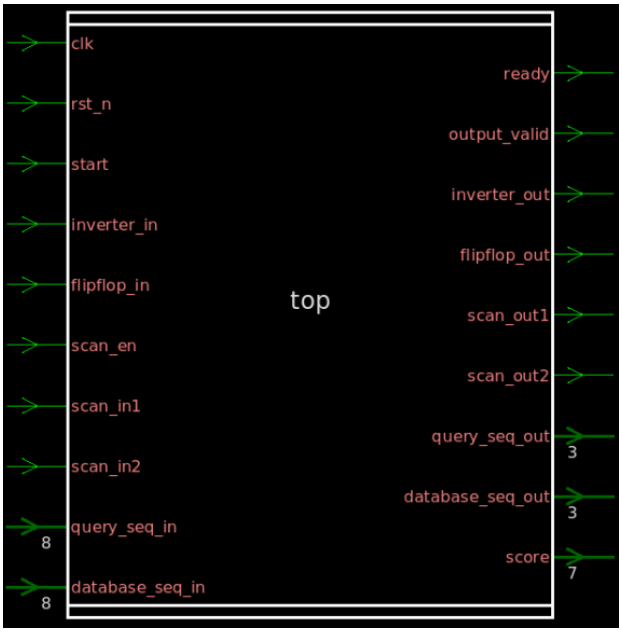


Figure 65: Schematic View of Top Cell After Synthesis

25 Logic Equivalence Check

Logic equivalence check (LEC) is a formal verification step used in VLSI design to compare between two different representations of a digital circuit. It ensures that the logical behavior of the two representations is identical for all possible input combinations. This stage is essential to confirm that any change made to the netlist, such as optimization or file format change, did not cause any change of the logical behavior.

We used the LEC Conformal tool by Cadence to perform 2 LEC checks during the physical implementation process:

1. RTL code against synthesized gate-level netlist.
2. Pre-layout gate-level netlist against post-layout gate-level netlist.

We flattened the golden and revised designs in order to cope with different signal names, sequential constant optimizations, merging of flops, redundant sequential logic that was removed during synthesis, etc. The tool performed hierarchical comparisons and all of them were equal.

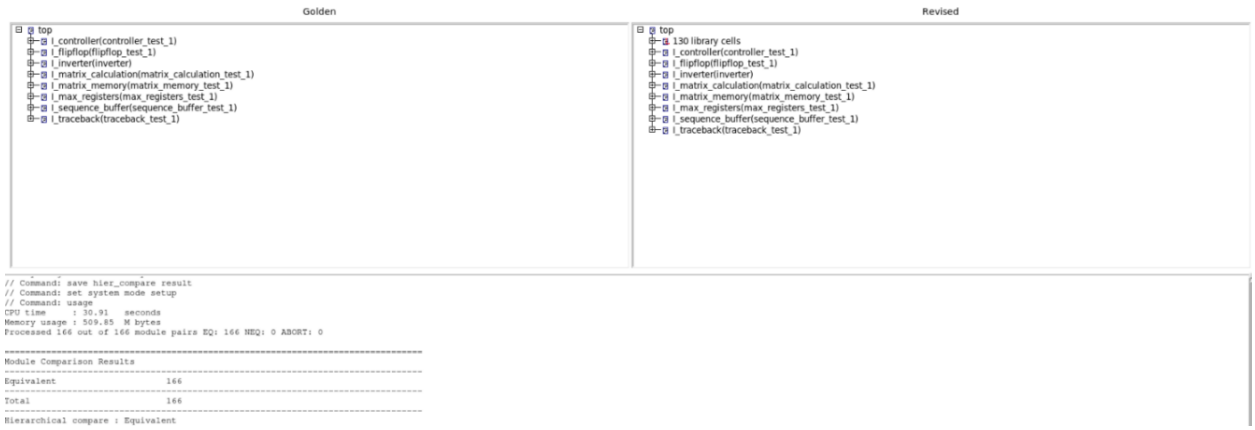


Figure 66: LEC Output Results

26 Gate-Level Simulations

Gate-level simulation (GLS) is the process of taking gate-level netlist and run it under a simulation that operates with zero delays, unit delays or full timing delays. This step is a crucial verification step – it checks how a circuit behaves dynamically when implemented with real gates.

We used the Verdi tool by Synopsys to perform 2 gate-level simulations during the physical implementation process:

1. Synthesized gate-level netlist – the output netlist of the synthesis stage, including pads.
2. Post-layout netlist – flat layout design, including pads.

Figure 41 shows the waveform of the gate-level simulation, which is identical to the RTL waveform described in detail in chapter 11.

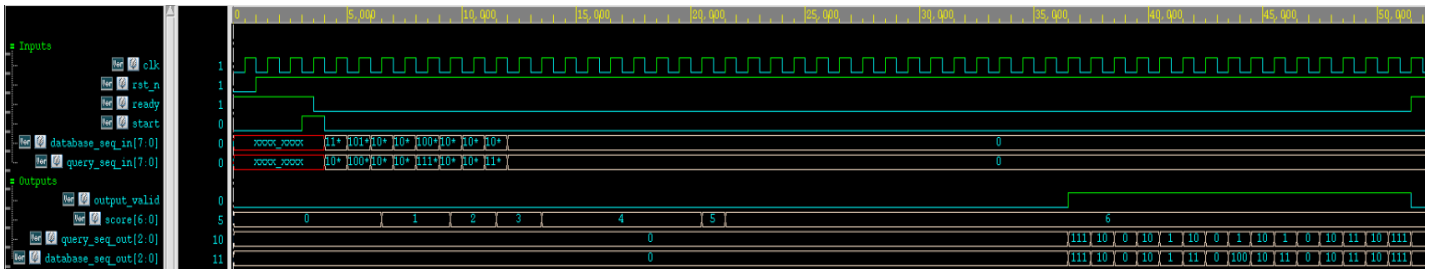


Figure 67: Synthesized Gate-Level Simulation

27 Floorplan

Floorplan is the process of determination of the die size and pads' location. In this stage we created the inner core area, specifying the core to I/O boundary spacing, standard cell rows, and placing I/O and corner pads. There is high importance for the location of the pads to achieve better area, to avoid congestion and to avoid IR drop.

Our design is a pad limited design, which means that the size of the core is determined by the number of pads. We took advantage if this fact in the placement stage to sparse the distance between the cells and get better power results.

We created the die with the following parameters:

- Total of 56 pins – 21 input pins, 17 output pins, 13 power pins, and 5 pins for scan chains.
- The total die size is 830x1090 [μm^2], without bond pads that add 170 [μm] to the height and the width. The bond pads were added at a later stage before the backend verification checks. The total height of the chip was determined to 1 millimeter due to fabrication requirements.
- The inner core size is 512x772 [μm^2].
- The core to I/O boundary spacing is 39 [μm].

After placing the pads, we added I/O filler cells between them to fill in the gaps, as well as tie cells used to connect nets to VDD and VSS.

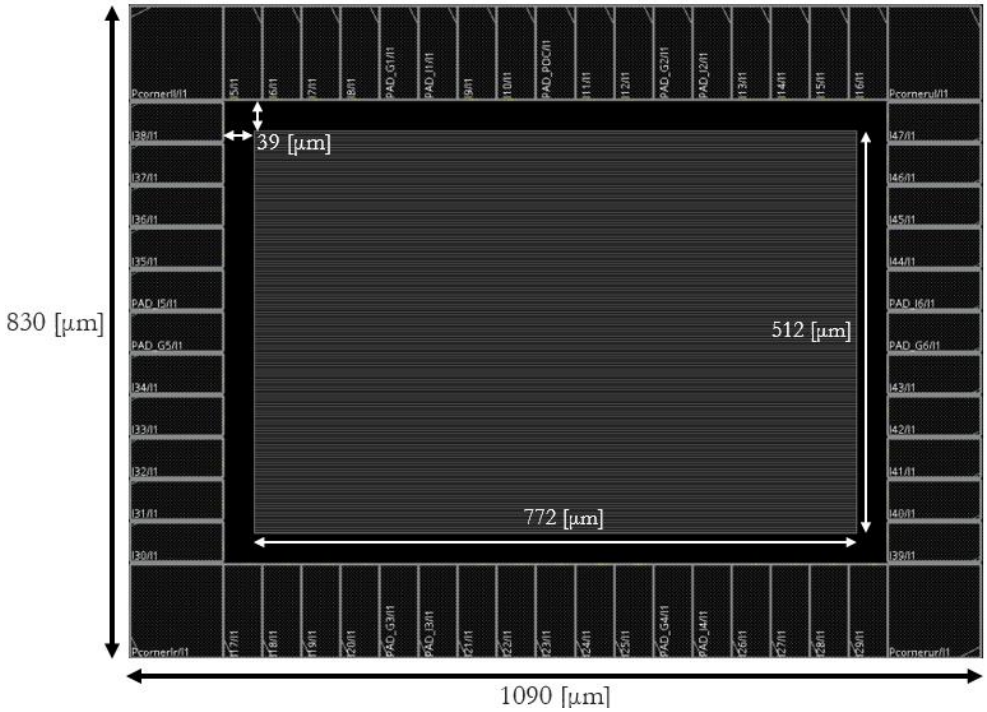


Figure 68: Chip's Floorplan

28 Power Grid

The main goal of the power plan is to create a proper way to supply power to the standards cells and macros through power routes. The die is connected to a power source via the power pads. The power is distributed to the chip from the power pads to the power ring and then to the power stripes and rails which connected to the cells.

We defined the power grid with the following parameters:

- **Power ring**: There are 2 power rings (for VDD and VSS) which are made of M7 and M8 layers. The vertical lines are made of M7 and the horizonal lines are made of M8.
- **Power stripes**: There are 2 types of power stripes. The vertical stripes are made of M8 layer, and the horizontal stripes are made of M7 layer. As shown in figure 43, the spacing between adjacent VDD and VSS stripes is 2.6 [µm]. The width of each stripe is 3.9 [µm]. The distance between a pair of VSS or VDD stripes is 21.45 [µm].

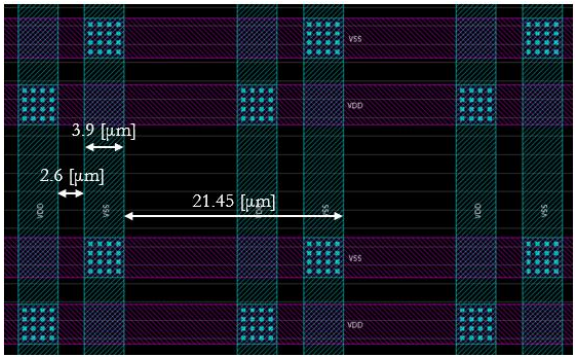


Figure 69: Power Stripes Distances

- **Power pads**: There are 4 pairs of power pads that provide power to the core and 2 additional pairs that provide power to the ring.

Figure 44 shows the power grid, as well as the power pads locations. We planned a dense power grid because the logic cells, and especially the matrix memory unit which contains 1024 3-bit registers, are very power consuming, so it is necessary to plan a proper power grid to reduce the IR drop effect to the minimum.

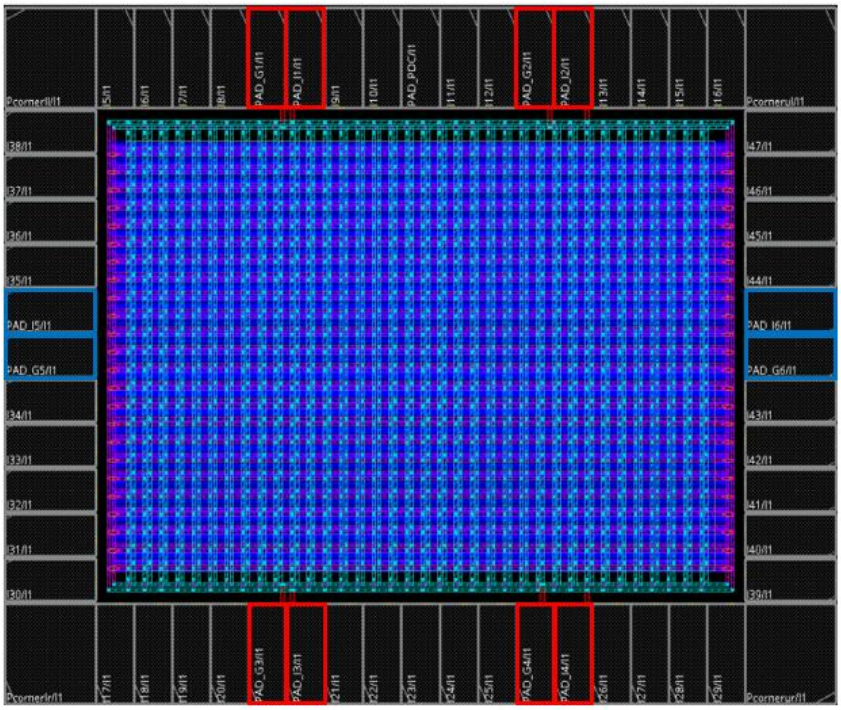


Figure 70: Power Pads Location
(The red pads supply power to the core and the blue pads supply power to the ring)

29 Placement

Placement is the process of placing the standard cells inside the core boundary in an optimal location. The tool tries to optimize the cells' location according to pre-determined parameters that are inserted into the tool. These parameters can prioritize timing, power, congestion, area, etc. This stage also includes placement of physical cells and adding of buffers and inverters to meet timing, DRV and DRC requirements. The placement process flow described in figure 71.

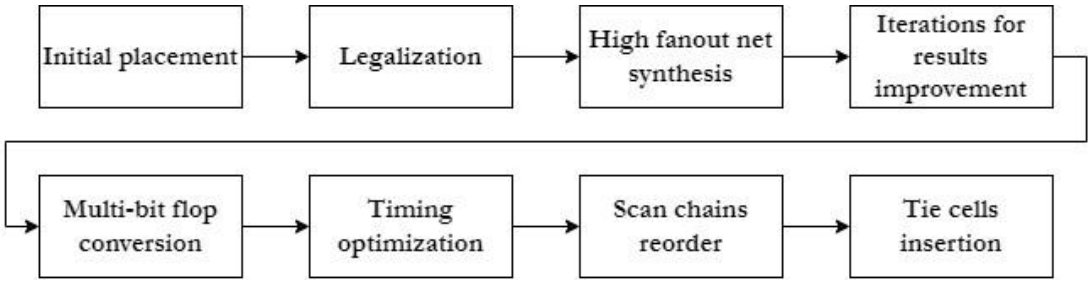


Figure 71: Placement Process Flow

We defined the following requirements:

- **High congestion effort**: We had several congestion problems in the design, so we used high congestion effort to solve the problem.

- **Maximum density of 25%**: In order to avoid IR drop, we wanted to place the cells sparsely enough to have similar power consumption on all the core area. Low density is also affecting timing because the distance between the cells becomes bigger and the signals takes more time to propagate through the wires. Because the logic is very power consuming, we had to compromise the timing results to achieve more homogeneous power distribution inside the core area.

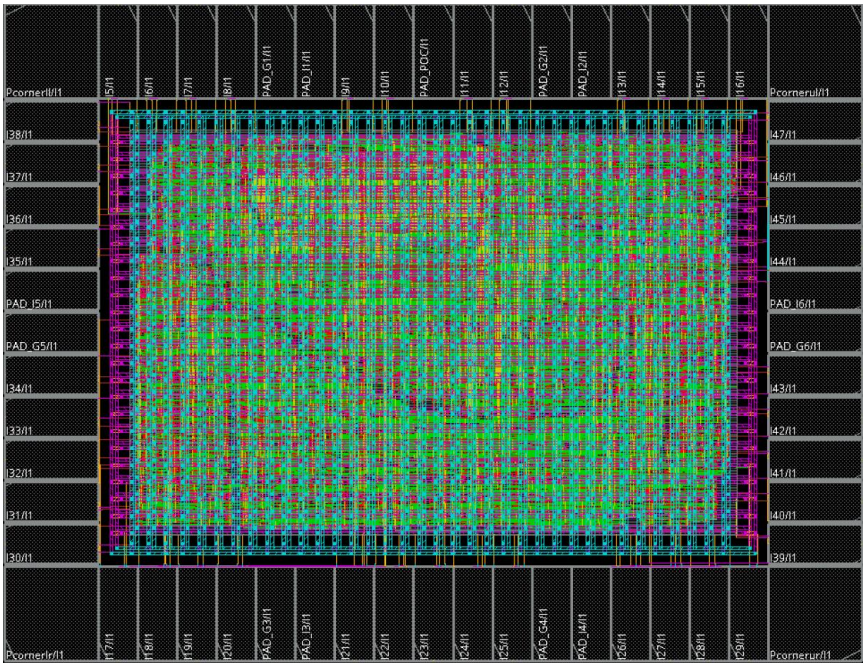


Figure 72: Chip's Layout After Placement

Figure 73 shows the amoeba view of the chip. It can be seen that the matrix memory registers occupy approximately 50% of the inner core area.

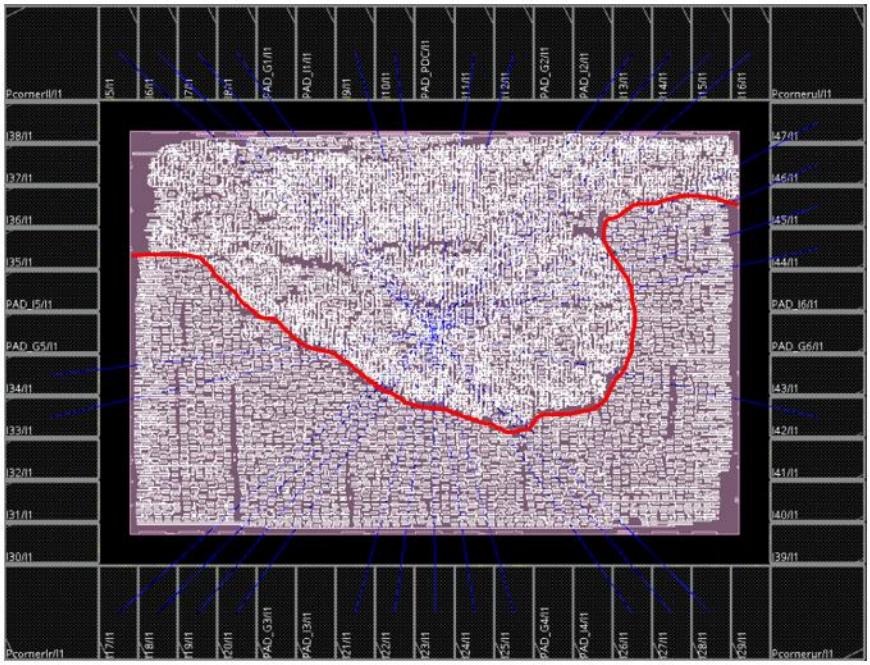


Figure 73: Amoeba Layout View
(The logic is placed above the red line and the matrix memory registers are placed below it)

30 Clock Tree Synthesis

Clock tree synthesis (CTS) is a technique for distributing the clock signal among all the sequential cells in the design. The purpose of the clock tree synthesis is to reduce the skew between cells getting the same clock and the insertion delay from the clock source to all end points. It does so by inserting buffers and inverters (or resizing them) to the clock routes. Clock tree synthesis includes both tree construction and clock tree balancing.

We defined the following constraints:

- Target skew of 0.3 nanoseconds between each two end points.
- Target maximum transition of 250 picoseconds.
- Top layer is M4 (for the top nets and branches), and bottom layer is M3 layer (for the leaves).
- Use specific buffers and inverters in different sizes (most efficient in terms of timing and area).

We got maximum clock skew of 55 picoseconds and maximum insertion delay of 0.322 nanoseconds, so the clock tree is relatively balanced. A scheme of the clock tree is shown in figure 74.

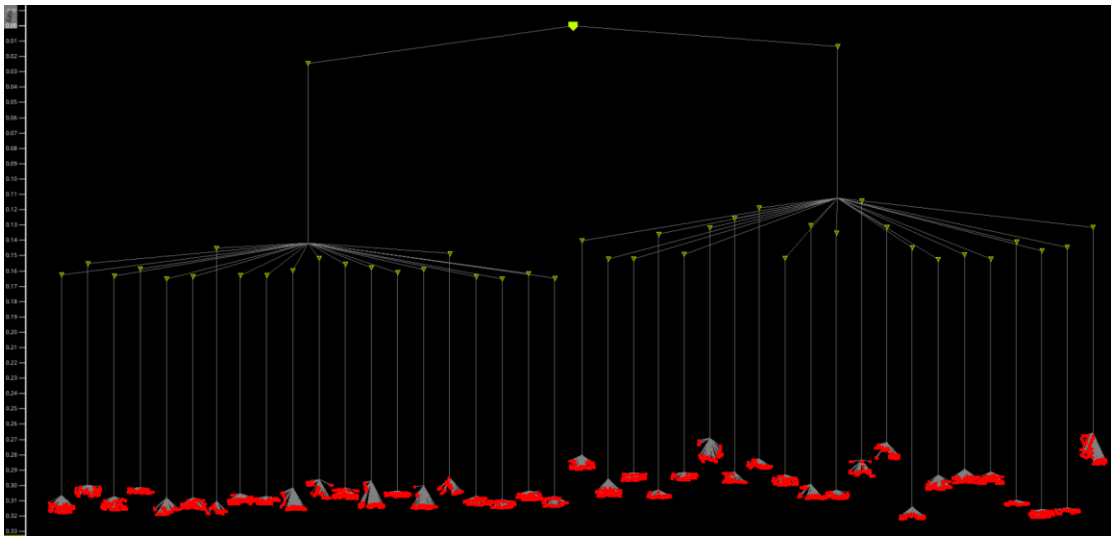


Figure 74: Clock Tree Synthesis

31 Routing

Routing is the process of creating physical connections between signal pins using metal layers, while following DRC rules and other constraints (timing, power, capacitance, etc.).

We used the `'sroute'` command (special route) to connect all the cells to VDD and VSS and then performed the following steps to route all the signals:

1. **Global Routing:** The tool performs global routing by assigning nets to metal layers and global routing cells without making any actual connections. It is necessary step to route the core in an efficient way.
2. **Track Assignment:** The tool replaces the global routes with actual metals. During the replacement, violation of DRC rules, signal integrity and timing violations may occur. Those violations are fixed on the next steps.
3. **Detailed Routing:** The tool takes care of the routing by completing it without leaving DRC violations and improving timing and signal integrity (if possible). To do this, the tool divides the core into sub boxes that will carry multiple global cells. Multiple iterations can be done to get the optimal routing without violations.
4. **Post Routing:** The tool makes signal integrity optimizations.

We used the `'nanoroute'` command that includes all these steps. We determined an equal congestion and timing effort, to balance between the DRC requirements and the timing results. Figure 75 shows the layout after routing.

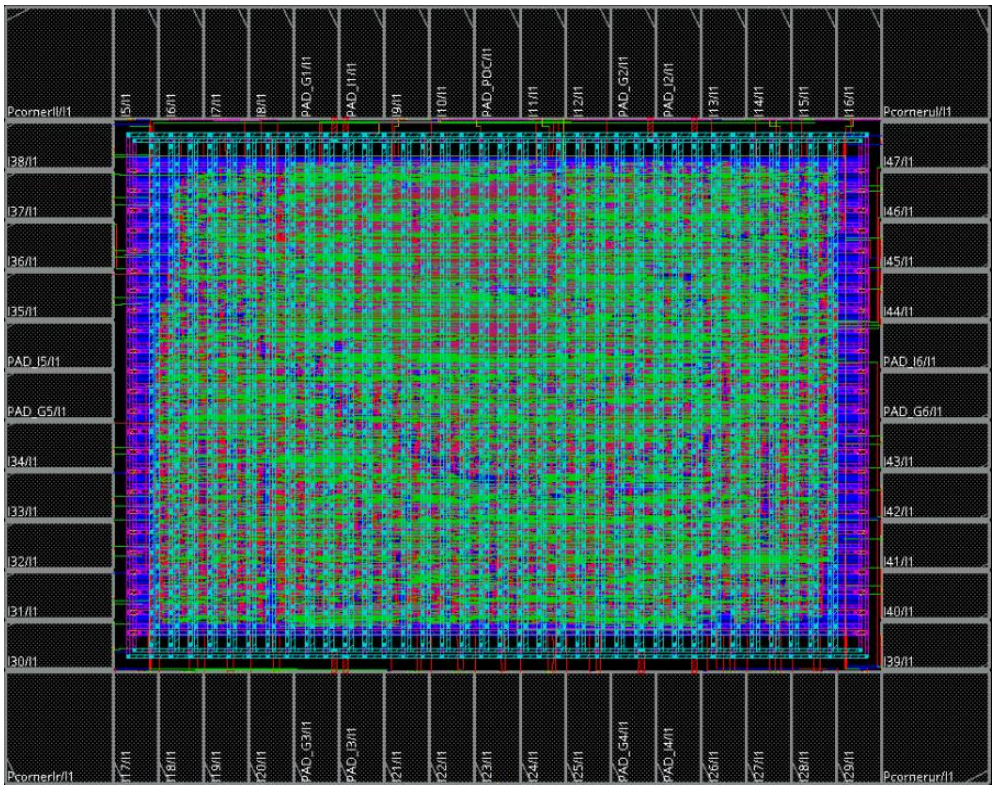


Figure 75: Chip's Layout After Routing

32 Power Signoff and IR Drop

The main goal of the power signoff is to make sure that the power consumption is not too high in several conditions. We used Innovus tool by Cadence to perform static power analysis under fast and slow corners. We defined the following parameters:

- Sequence activity: 0.2
- Input activity: 0.3
- Clock gate activity: 0.1

We made several attempts to reduce the power consumption to the minimum. The main efforts include a dense power grid, sparse cells placement, and high power optimization efforts. Table 16 shows the final power consumption results.

	Slow RC	**Fast RC**
Internal Power	29.29 [mW] (51.34%)	45.45 [mW] (57.39%)
Switching Power	27.67 [mW] (48.49%)	32.65 [mW] (41.22%)
Leakage Power	0.096 [mW] (0.17%)	1.096 [mW] (1.38%)
Total Power	**57.06 [mW]**	**79.19 [mW]**

Table 16: Power Consumption Under Fast and Slow Corners

IR drop is a voltage drop due to the resistance of a conductor. It occurs when signal propagates through a wire or circuit, resulting in a loss of voltage proportional to the resistance and current in the circuit. After the power analysis, we checked for the IR drop of the VDD and VSS signals, as can be seen in figure 50. The maximal IR drop is 6.6% for VDD and 7.1% for VSS. As expected, the maximal IR drop occurs in the bottom of the inner core area, where the matrix memory registers are placed.

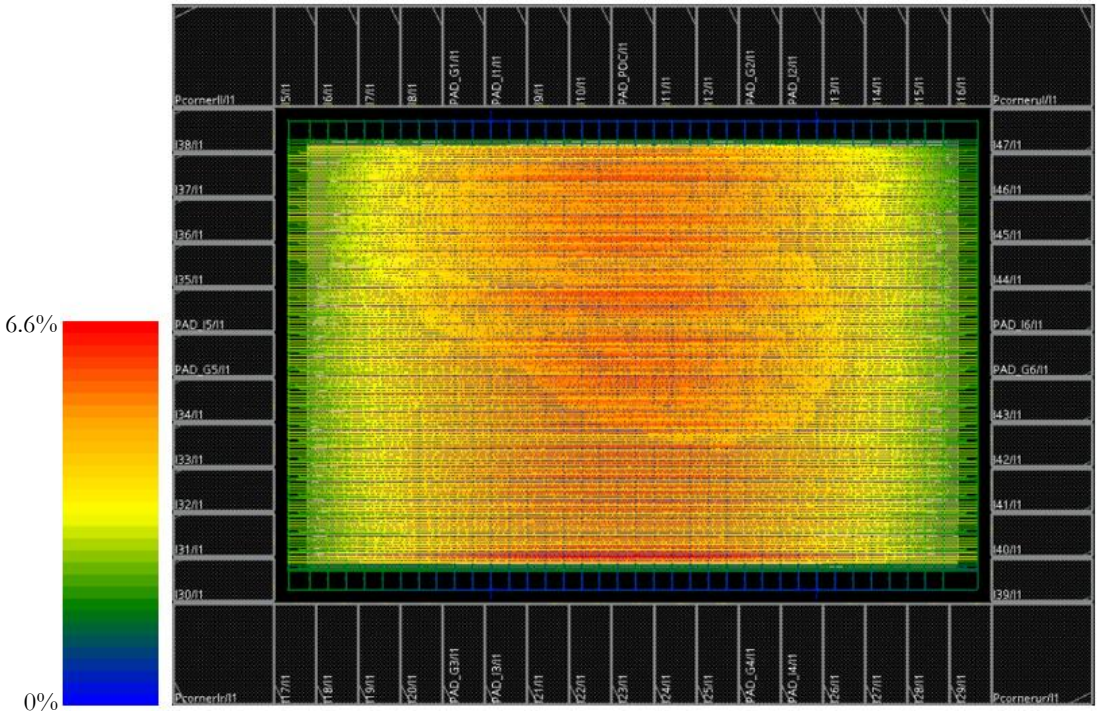


Figure 76: IR Drop of VDD Signal

33 Timing Signoff

The main goal of the timing signoff is to make sure that all signals propagate through the core within the specified timing constraints, without causing setup or hold violations. The tool divides the design circuit into 4 sets of timing paths: input → reg, reg → reg, reg → output, and input → output. Then, it analyzes the delay of all the signals paths and determines the worst case and best case for various paths. This information is important for us to know that the design functions properly and do not violate setup and hold conditions.

We used PrimeTime tool by Synopsys to perform MCMM (Multi-Corner Multi-Mode) analysis and checked for setup and hold violations under On-Chip Variations. We defined a derate factor of 15% (upper and lower limit) and a clock frequency of 100 [MHz]. Table 17 shows the information of the critical paths.

	Setup Condition		Hold Condition	
Corner	**Max Path**	**Slack**	**Min Path**	**Slack**
Fast	3.44 [ns]	+6.65 [ns]	0.19 [ns]	+0.02 [ns]
Typical	5.33 [ns]	+4.78 [ns]	0.26 [ns]	+0.06 [ns]
Slow	9.14 [ns]	+1.00 [ns]	0.41 [ns]	+0.14 [ns]

Table 17: Timing of Critical Paths

We did several attempts to achieve a successful timing signoff with a frequency of 100 [MHz]. We had to take into account the I/O pads location around the core and to consider the tradeoff between power consumption and timing results. Some of the effort include attempts to implement clock gating, various placement density options, different power grid plans, and different placement optimization constraints.

34 Backend Verification

LVS (Layout Versus Schematic) check is an essential step in design verification, ensuring the correspondence between the physical layout and the intended circuit schematic. This process compares the geometric layout of transistors, interconnections, and components against the design schematic to identify any inconsistencies or errors. By analyzing factors such as connectivity, device sizes, and parasitic elements, LVS validation confirms that the layout is equivalent to the original netlist.

DRC (Design Rule Check) is an important check for ensuring compliance with fabrication constraints and guidelines. This process meticulously examines the layout of transistors, metal layers, and interconnects against a set of predefined rules and specifications provided by the fabrication process. DRC identifies violations such as minimum spacing between features, width constraints, overlap, and other geometric irregularities that could compromise the manufacturability or performance of the chip.

Before the backend verification checks, we used a layout view of the chip and added bond pads to create an interconnection between the chip and the board, using Virtuoso tool by Cadence. We also added more metals to connect the power pins to the power grid with a smaller resistance and added a small logo (in the AP layer) at the core's top left corner to

identify the chip's orientation. Then, we performed the LVS and DRC checks using Calibre tool by Siemens. We did several attempts and manually fixed violations until we passed both checks without warnings and errors. At the end of this stage, we had the final layout ready for fabrication as shown in figure 77.

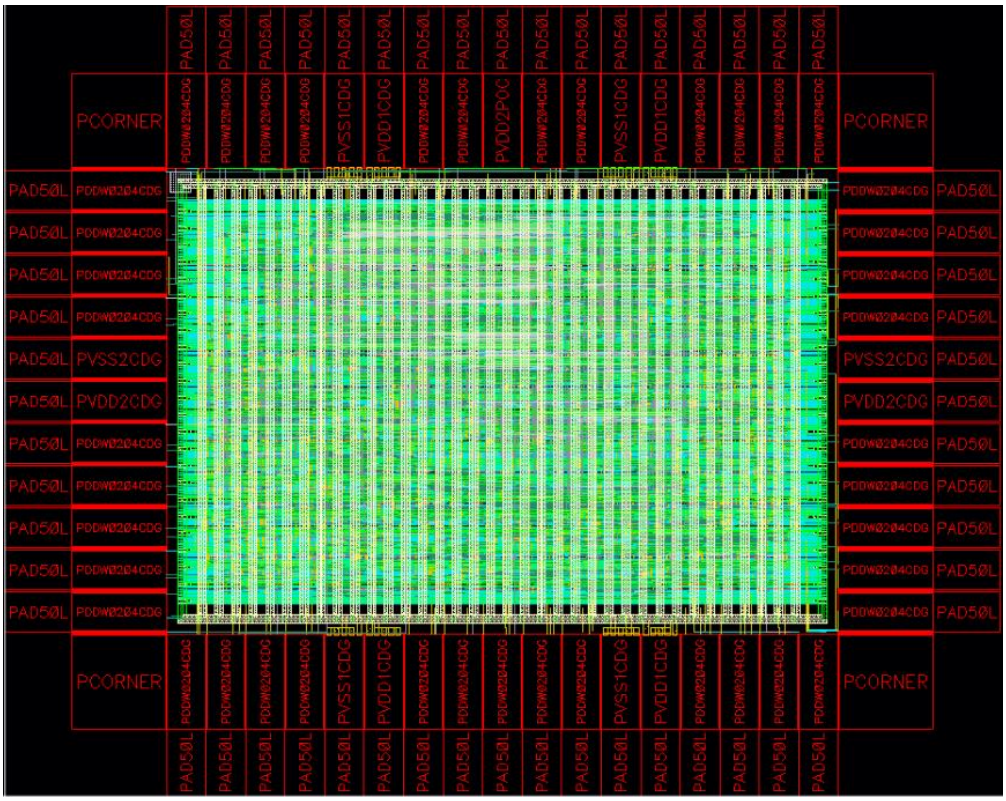


Figure 77: Final Layout for Fabrication

Possible Improvements

In order to meet the time frame of the project work, we had to simplify its specifications and requirements. This chapter will cover some possible improvements we think can be implemented with future work:

1. **Pipelined architecture**: The current chip was designed in a multi-cycle architecture, so it supports alignment of a single pair of sequences at a time. A more effective option is to design the chip with a pipelined architecture, so it will support alignment of multiple pairs at a time and will improve the chip's throughput.
2. **Matrix calculation granularity**: The matrix calculation unit divides the substitution matrix into groups of 2x2 cells and performs the calculations in a granularity of 4 cells using the PU logic. Because all other algorithm stages have a simpler logic, the computation is shorter. We could have designed the calculation in a granularity of 1 cell, so the clock cycle will be shorter (but the total matrix computation will take 63 cycles instead of 31 cycles). It could help to increase the work frequency as well as enhancing the chip's throughput but increase the latency.
3. **Implementation of clock gating**: During the backend design work we faced some power consumption and IR drop problems. To solve this, we tried to implement clock gating. After analyzing the flop activity of the design, we decided to apply the clock gating on the Matrix Memory unit, which is a large power consumer and has a high flop activity. We implemented it for each diagonal in the matrix to get optimal results. In this way we reduced both static and dynamic power consumption dramatically. However, we encountered some serious bugs that we did not have time to solve because of a strict fabrication deadline, so we removed the clock gating.
4. **Modular operation**: Supporting of variable scoring system or alignment of sequences with unequal lengths could make the accelerator more modular but requires design of much more complex system.

Summary and Conclusions

This project introduced a parallel implementation of Smith-Waterman algorithm by a hardware accelerator. It has been a great milestone in our way to become electrical engineers.

Before starting work on this project, the sequence alignment problem was unfamiliar for us. We had to dive into a new topic and to understand comprehensively the algorithm. We learned during this project how to plan a complex engineering system, different approaches of chip architecture, tradeoffs and limitations we must consider, System Verilog skills, and much more. We also learned how to solve problems independently and how very important preliminary planning is.

We would like to express our sincere appreciation to the VLSI lab staff, especially to our instructor, Goel Samuel. His invaluable assistance played a pivotal role in the realization of our project. We are also thankful to Apple Israel for their collaboration and generous funding of this project, and especially to Dalia Haim, who is leading this collaboration. We are extending our gratitude to everyone who contributed to the project's success.

This project has been a remarkable experience of learning, development, and expanding horizons. We are looking forward to see our chip working!

References

[1] Eddy, S. R. (2008). A probabilistic model of local sequence alignment that simplifies statistical significance estimation. *PLoS computational biology*, *4*(5), e1000069.

[2] Liu, J. S., Neuwald, A. F., & Lawrence, C. E. (1995). Bayesian models for multiple local sequence alignment and Gibbs sampling strategies. *Journal of the American statistical Association*, *90*(432), 1156-1170.

[3] Roĭtberg, M. A. (2004). Comparative analysis of primary structure of nucleic acids and proteins. *Molekuliarnaia Biologiia*, *38*(1), 92-103.

[4] Mentor Graphics. (n.d.). *UVM Cookbook*. Verification Academy. Retrieved from https://verificationacademy.com

[5] Synopsys. (2020/2021). *VHDL/Verilog Simulation and Coverage, UVM, VLSI Lab*. Electrical and Computer Engineering, Technion.

[6] UVM 2020-3.1 Reference Implementation, *UVM v2020.3.1 Library Code for IEEE 1800.2*. Public UVM GitHub Repository, 3.1 release, Aug. 2024. [Online]. Available: *https://github.com/uvm/reference-2020-3.1*

[7] *IEEE Standard for Universal Verification Methodology Language Reference Manual*, IEEE Standard 1800.2-2020, IEEE, 2020. doi:10.1109/IEEESTD.2020.9088527.

[8] "Smith-Waterman Tool," [Online]. Available: http://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Smith-Waterman.