D-PEARL: Dynamic Programming Execution Accelerator using Race Logic

Derek Chaw
drchaw2@illinois.edu

Ryan Cheng
chingc5@illinois.edu

Arcy Cruz
acruz86@illinois.edu

Zixiao (Alan) Huo
zixiaoh3@illinois.edu

Mahir Koseli
mkoseli2@illinois.edu

Ahmed Shafiuddin
ahmeds4@illinois.edu

***Abstract*—Dynamic programming–based sequence alignment is computationally intensive and motivates alternative computing paradigms beyond conventional digital architectures. Race Logic encodes information in signal propagation delay rather than logic levels, allowing additions and comparisons to be implicitly realized through signal races, making it well suited for shortest-path formulations common in dynamic programming algorithms.**

D-PEARL is a domain-specific hardware accelerator that accelerates global sequence alignment by expressing the Needleman–Wunsch dynamic programming formulation in Race Logic, rather than executing it using conventional digital computation. Compared to prior fixed-function race-logic designs, D-PEARL supports variable-length sequences up to 27 symbols, customizable on-chip scoring matrices, and full hardware-based traceback, improving both energy efficiency and applicability across a broader range of alignment workloads. The architecture is realized as a 27×27 race-logic mesh with programmable delays. Implemented in a 65 nm CMOS process with a fixed die area of 1 mm^2, D-PEARL achieves timing closure at 200 MHz.

I. Project Summary

A. Main Features and Functionality

D-PEARL is a hardware accelerator designed to speed up sequence alignment workloads that are traditionally solved using dynamic programming, with a primary focus on protein sequence alignment. Rather than directly executing a dynamic programming algorithm in hardware, D-PEARL accelerates the core computation by mapping the alignment problem onto a race logic mesh that exploits temporal signal propagation.

The target application is the Needleman–Wunsch algorithm, which computes the optimal global alignment between two biological sequences. An interactive demonstration of this algorithm can be found at https://bioboot.github.io/bimm143_W20/class-material/nw/.

The design draws inspiration from the Race Logic paradigm introduced by Madhavan et al. [1], which leverages relative signal arrival times through logic gates to efficiently evaluate dynamic programming recurrence relations. Instead of computing alignment scores through explicit arithmetic operations, D-PEARL encodes insertion, deletion, and substitution costs as programmable delays within a 2D grid of processing elements.

Key features include support for sequences up to 27 symbols with configurable alphabets of up to 20 characters, on-chip storage for substitution scoring matrices optimized for BLOSUM62, hardware-based path reconstruction for direct traceback extraction, and flexible streaming and non-streaming operation modes. The chip provides a complete solution, producing both the optimal alignment score and the corresponding edit path.

B. High-level Architecture

The D-PEARL accelerator is organized around a 27×27 race-logic mesh that physically represents the edit graph underlying the dynamic programming formulation of sequence alignment. The main components include the following.

The **I/O Controller** manages bidirectional communication with the host using 14-bit input and 13-bit output packet buses. The **Control Unit** orchestrates system operation as a finite state machine, coordinating initialization, computation, and result handling. **Sequence buffers** store input sequences using 5-bit encodings, while the **SRAM subsystem** holds substitution scoring matrices in a compact, symmetric format. An **Address Generator** bridges the control logic and SRAM, supporting both initialization and dynamic lookup operations.

The **Race Logic Mesh** forms the computational core of the accelerator. Each cell implements OR-based race logic with programmable delays, allowing computation to proceed as a temporal wavefront from the top-left to the bottom-right of the grid. Signal arrival times encode alignment costs, and each cell includes traceback logic to enable path reconstruction. A **Distance Counter** captures the final alignment score at the destination cell.

Operation begins with host-driven configuration and sequence loading, followed by grid initialization where penalties and substitution delays are programmed. Computation is initiated by injecting a signal into the origin cell, and results are collected upon completion and transmitted back to the host.

For detailed information on each component, including packet formats, SRAM organization, cell architecture, and path reconstruction mechanisms, please refer to the respective subsections in the Documentation section below. Block diagrams and additional illustrations are also provided to give a comprehensive view of the system's architecture and dataflow.

C. Physical Design

Fig. 1 illustrates the final physical layout of the proposed design. Only metal layers M1–M5 are shown, while upper metal layers dedicated to power rings and power stripes are

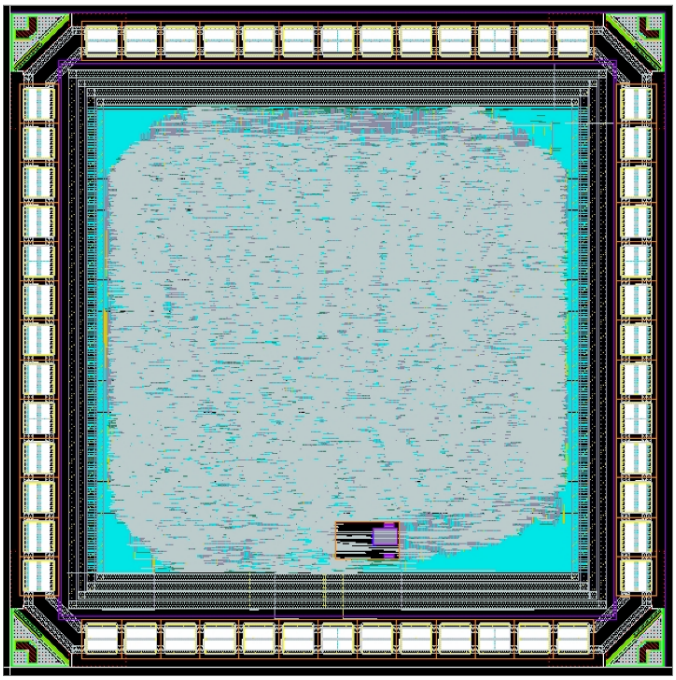

Fig. 1. Final layout

intentionally omitted to improve visualization. The die area is fixed at 1 mm^2, within which the core occupies an area of 708 μm $\times$ 708 μm (501,264 μm^2), resulting in a core area utilization of 84.319%.

The design achieves timing closure at a target operating frequency of 200 MHz. To maximize performance, flat synthesis is employed for the majority of the design. As shown in Fig. 1, the layout contains a single macro block corresponding to the register file, while all remaining logic is implemented using standard cells in a flat design style. At 200 MHz, the total power consumption of the design is 36.7 mW.

The design incorporates 13 I/O pads on each side of the core, resulting in a total of 52 I/O pads. Among these, 48 pads are utilized, including four dedicated power pads for VDD and VSS.

D. Design Choices

A central design choice in D-PEARL was to implement the dynamic-programming engine as a fully synchronous Race Logic grid rather than using the asynchronous delay elements from prior work. We chose synchronous counters and clocked control logic for each cell because it fits much better with the standard digital flow we had available. While asynchronous delay chains can be more efficient in terms of area and power, they would have required custom timing closure, non-standard verification, and more complicated bring-up. By using synchronous delay elements, we traded some area and performance for a design that we could confidently synthesize, time, and debug within the constraints of a one-semester tapeout project.

Another major choice was to fix the grid dimensions and scoring representation up front. We targeted a grid of size 27×27 with bounded delay and preprocessed scoring matrices (e.g., BLOSUM) into a strictly positive delay domain. Fixing these parameters allowed us to size internal counters, SRAM width, and control-unit fields early, which simplified both RTL and physical design. The downside is reduced flexibility; our taped-out accelerator is optimized for a specific maximum sequence length and symbol set, but that was acceptable for a first implementation where successful silicon was more important than supporting every possible configuration.

On the control side, we opted for a relatively structured, centralized control unit with a well-defined set of opcodes and a status register, instead of distributing control across several loosely coupled blocks. The control unit manages SRAM loading, insertion and deletion (indel) configuration, grid initialization, run, and result output through a small command protocol. This choice let us keep the external interface narrow and made it easier to reason about legal state transitions and error handling. It also provided a natural place to encode status bits (e.g., “SRAM loaded,” “grid initialized,” “result ready,” “error”) that could be observed both in simulation and on silicon.

Originally we planned to include a full debug controller that could extract the entire grid, sequences, indel registers, and SRAM contents via a dedicated debug protocol. Late in the project we decided to drop this block from the taped-out netlist. The main reasons were schedule pressure and complexity; the debug controller added a significant amount of control logic, verification effort, and interface definition, and it became clear that fully validating it at SDF-annotated gate level would risk the overall tapeout schedule. Instead, we chose a lighter-weight debug strategy: we added dedicated debug pins that expose the control unit state, key status-register fields, and other high-level signals that provide better visibility into our ASIC, and ease the complexity of post-silicon validation. In other words, we prioritized a smaller set of robust, easy-to-use debug hooks over a more ambitious but high-risk debug subsystem.

Finally, throughout the design we favored structures that were friendly to verification and physical design, which include explicit resets on state and counters, packed typedefs for all external I/O packets, and clean separation between datapath and control. These choices were informed directly by issues we encountered when moving from behavioral simulation to SDF gate-level simulation. Ensuring that each flip-flop had a well-defined reset value and that all externally visible state could be traced through a few key signals improved our confidence that the taped-out design would behave predictably in silicon and would be debuggable with the limited observability available on the evaluation board.

E. Verification and Test Strategy

Verifying a novel, timing-driven architecture was one of the primary challenges of this project. From the conceptual stage, we performed systematic checks to ensure that the mapping

of the edit distance algorithm onto race-logic hardware was functionally correct and physically sound. Each architectural change was validated before further refinement.

At the RTL level, we built the entire verification flow from scratch, including a custom golden model and verification framework. We adopted a bottom-up verification strategy, starting with individual submodules and progressively integrating them into larger subsystems and the full design. At each hierarchical layer, we used transaction-level modeling (TLM) to abstract low-level signal interactions into higher-level transactions, enabling scalable stimulus generation and early bug isolation (see Fig. 2).

We relied heavily on SystemVerilog assertions, using immediate assertions for local, cycle-specific checks and concurrent assertions to verify multi-cycle temporal properties. Given the timing-sensitive nature of race logic, these assertions were critical to enforcing correct sequencing and event ordering.

DFT considerations focused primarily on post-silicon observability and reliable bring-up rather than full scan-based testing. All major state-holding elements in the control logic and datapath were given explicit reset values to ensure deterministic behavior and avoid uncontrolled X-propagation. Dedicated debug pins expose high-level signals such as the control unit state and key status-register flags (initialization progress, result readiness, error), providing immediate insight into the accelerator's operational state. A power-observation pin tied to VDD allows verification of proper ASIC power before functional testing.

A full debug controller capable of extracting internal grid, SRAM, and sequence state was considered early but excluded from the taped-out design due to complexity, verification cost, and timing risk. The chosen lighter-weight DFT strategy prioritizes strong reset discipline and coarse-grained observability, significantly reducing design and verification overhead while still supporting effective post-silicon validation.

F. Comparison with Proposal

The original D-PEARL proposal described a flexible Race Logic accelerator for dynamic programming workloads, with support for programmable scoring, variable-length sequences, path reconstruction, and extensive post-silicon observability (see Fig. 3). By the midterm milestone, the core architectural goals had been achieved, including a fully implemented and verified Race Logic grid, control unit, sequence buffers, processing elements, and I/O controller. Several features were also added beyond the proposal, such as stream mode operation and enhanced multi-length sequence support, improving practicality for real-world workloads.

During development, verification and post-synthesis validation proved significantly more complex than anticipated, particularly due to the non-conventional timing-based nature of Race Logic. Building constrained-random test infrastructure, golden models, and coverage required substantial effort, and gate-level simulation exposed issues related to reset behavior, FSM optimization, and X-propagation that were not visible at the RTL level. These challenges led to a shift in focus from feature expansion toward design robustness and tapeout reliability.

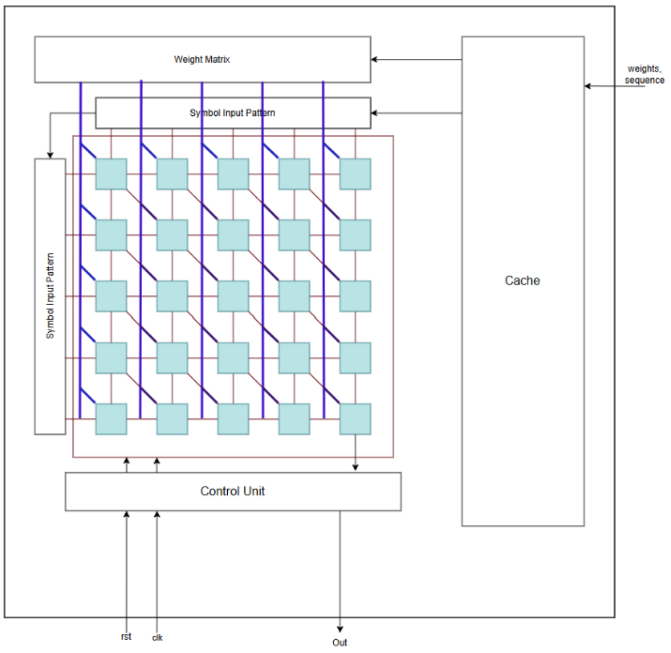

Fig. 3. Initial D-PEARL block diagram (May 2025)

Although a dedicated debug core was designed, verified, and integrated at midterm, it was ultimately excluded from the final taped-out design due to its verification and timing risk. Instead, a simplified debug approach was adopted, providing visibility into key control and status signals through dedicated pins. This trade-off reduced complexity while maintaining sufficient observability for post-silicon bring-up.

Overall, the final design preserves the proposal's core vision while reflecting practical trade-offs driven by verification effort and schedule constraints. The project demonstrates both the feasibility of Race Logic for dynamic programming acceleration and the importance of adapting design scope to ensure a reliable tapeout.

II. Post-Silicon Bring-Up

In Spring 2026, we plan to perform post-silicon bring-up and validation of the fabricated chip using a custom-designed printed circuit board (PCB) and an FPGA-based test harness. The PCB will house the ASIC and provide power delivery, clocking, and I/O connectivity to the FPGA, which will act as the primary control and data interface during testing. The FPGA will be used to stream input sequences to the chip, configure runtime parameters, and capture output timing and results for comparison against a software golden model.

The FPGA-to-ASIC interface will be implemented as a direct electrical connection on the PCB, without intermediate components or complex protocols. This straightforward interfacing strategy is intended to reduce PCB design complexity

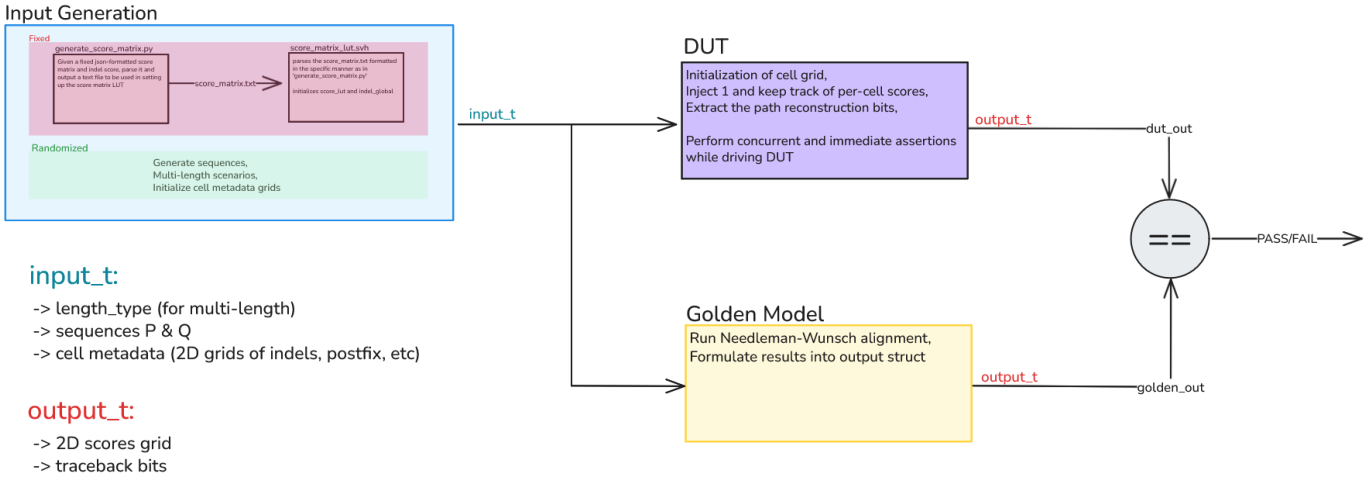

Fig. 2. Mesh-level transaction-level modeling (TLM) (click to expand)

and minimize sources of integration error during early bring-up.

Testing will proceed incrementally, beginning with basic power-on checks and advancing toward full system-level evaluation. A preliminary timeline, assuming chip delivery in February 2026, is outlined below:

- **December – February:** FPGA test harness development, PCB design and fabrication, bring-up scripts, and software golden model validation
- **Late February – Early March:** Initial power-on, clock and reset verification, and basic functional checks
- **March:** Functional validation using short-sequence tests and characterization of Race Logic grid timing behavior
- **Late March – April:** Full-workload evaluation, frequency and power measurements, and stress testing
- **April – May:** System-level integration and final analysis of results

One of the primary goals of this ASIC is to accelerate compute-intensive components of large-scale bioinformatics workloads, such as the Basic Local Alignment Search Tool (BLAST), which operates on sequence lengths exceeding millions of elements. Post-silicon testing will therefore include representative workloads designed to evaluate performance scaling and correctness under realistic operating conditions.

The members planned to work on post-silicon bring-up and validation during the Spring 2026 semester are Ahmed Shafiuddin, Derek Chaw, and Arcy Cruz.

III. Individual Contributions

- **Derek**
  - Collaborated on the initial unit cell architectural design.
  - Developed synthesis strategy, workflow, and scripts for timing optimization.
  - Optimized SRAM architecture and implemented it with a behavioral model and layout generation.
  - Collaborated with Ryan on PnR scripts.
  - Handled everything post-PnR and final layout integration, including IO pad integration, manually rerouting wires to fix IO congestion, metal filling to meet density rules, manually fixing layout to pass DRC, and performing final DRC and LVS signoff to meet foundry requirements.
- **Ryan**
  - Owned the physical design flow, including running synthesis and PnR, and resolving timing violations, DRC, and related implementation issues.
  - Developed and maintained synthesis and PnR Tcl scripts.
  - Collaborated on the RTL design of the address generator.
  - Collaborated on verification coverage and developed the golden model for design verification.
- **Arcy**
  - Created and debugged the given tool chain.
  - Collaborated on the PD flow, working on synthesis, GDSII output, and debugging issues along the way.
  - Debugged and handled PnR Tcl scripts.
- **Zixiao (Alan)**
  - Developed the initial RTL implementation of the unit cell.
  - Drafted the controller flow into specific states.
  - Collaborated on the PD flow and created scripts for STA.

- **Mahir**
  - Implemented RTL for the Sequence Buffers.
  - Developed verification (DV) for the Processing Unit.
  - Implemented the Race Logic Grid RTL.
  - Developed FSM and RTL for the Control Unit.
  - Handled top-level integration and debugging.
  - Designed and verified RTL for the Debug Controller.
- **Ahmed**
  - Led the team by assigning tasks and coordinating progress.
  - Architected the unit cell, address generator, and top-level design.
  - Developed and finalized RTL for the unit cell, address generator, and top-level modules.
  - Owned the entire verification flow.
  - Adapted the golden model to accommodate advanced features.

IV. Major Challenges

One of the primary challenges in the D-PEARL project was post-PnR verification. While our design behaved correctly in behavioral RTL simulation, gate-level simulation with SDF annotation exposed issues that were not visible earlier in the flow. In particular, missing or insufficient reset logic led to widespread X-propagation once real timing was introduced, and synthesis optimizations such as FSM re-encoding altered internal signal behavior. Debugging these issues required careful analysis of the synthesized netlist, timing reports, and waveform traces, significantly increasing verification effort and extending our project timeline. This experience underscored the importance of designing with gate-level behavior in mind from the beginning and incorporating post-synthesis and post-PnR verification earlier in the development cycle.

A second major challenge was achieving meaningful functional coverage for a non-conventional, timing-based architecture. Unlike traditional datapath designs with well-defined input–output relationships, Race Logic encodes computation in signal arrival times, making it difficult to define intuitive coverage metrics. We had to develop custom golden models, constrained-random stimulus generators, and assertions that could validate both functional correctness and timing-dependent behavior. Ensuring adequate coverage of grid configurations, sequence lengths, and control-flow paths required substantially more effort than anticipated and revealed that verification for novel architectures often dominates overall development time.

The third major challenge involved DFT and post-silicon observability. Early in the project, we explored implementing a comprehensive debug controller capable of extracting internal grid, SRAM, and sequence state. However, the complexity of verifying this module, particularly at gate-level, introduced significant risk to the tapeout schedule. Balancing observability against design complexity required difficult trade-offs. Ultimately, we opted for a simplified DFT approach that emphasized deterministic reset behavior and coarse-grained visibility through dedicated debug and status pins. While this reduced internal visibility, it provided a more robust and verifiable path to post-silicon bring-up.

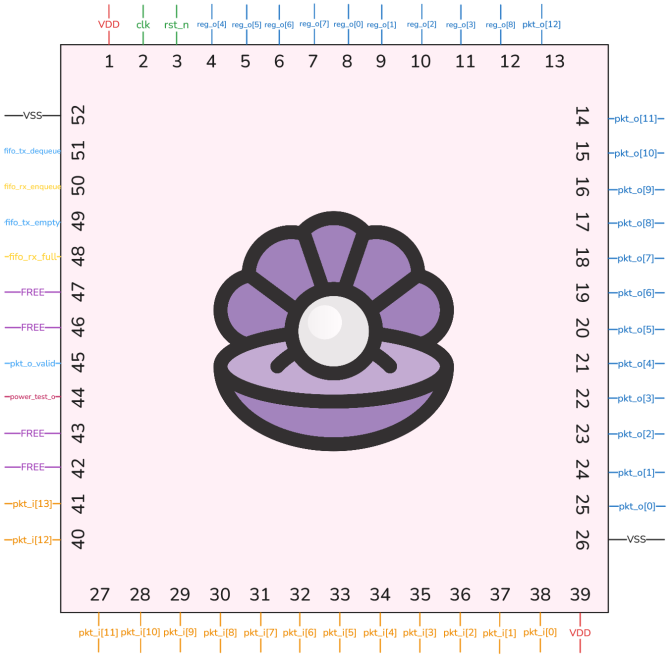

Fig. 4. Pinout (click to expand)

Together, these challenges shaped both the technical outcome and our development process. Addressing post-PnR verification issues, building coverage for a novel architecture, and making pragmatic DFT trade-offs provided us with valuable experience that closely mirrors real-world ASIC design and tapeout workflows.

V. Documentation

A. Top-Level Pinout

Figure 4 shows the top-level pinout of the D-PEARL chip. The interface includes control, data, and power signals, arranged for efficient routing and easy access during both pre-silicon simulation and post-silicon bring-up. Pins are color-coded by function, with orange for inputs, blue for outputs, and other groups labeled distinctly.

The primary input bus `pkt_i[13:0]` receives instructions and data from the host, while the output bus `pkt_o[12:0]` returns computation results. Status pins `reg_o[8:0]` expose internal state, progress, and configuration registers for real-time monitoring.

The `clk` and `rst_n` pins provide the global clock and active-low reset, distributed via balanced clock trees. Power is supplied through `VDD` and `VSS` pins along multiple edges, with an extra power test pin for basic power verification.

B. High-level Diagram

Figure 5 presents the top-level architecture of the system. Data enters through the I/O controller, which, under the

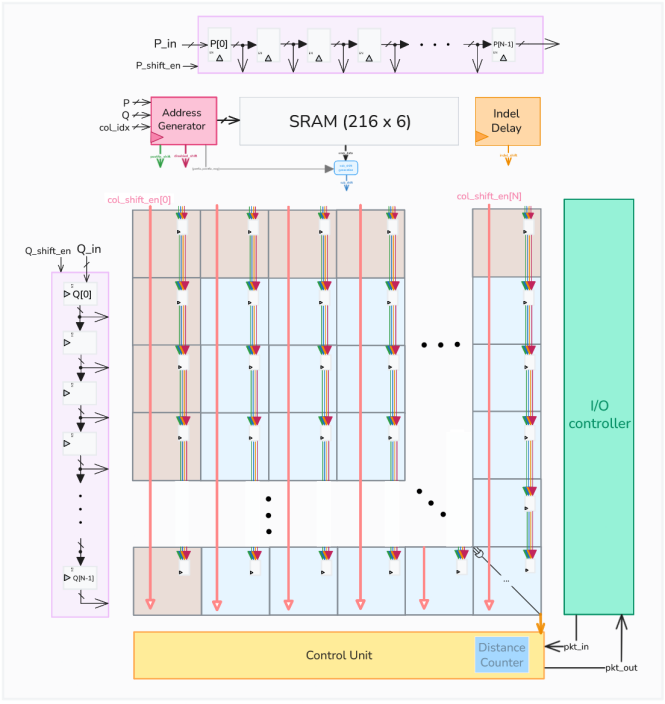

Fig. 5. Top-level block diagram (click to expand)

direction of the control unit, initializes the SRAM, the P/Q sequence SIPO shift registers, and the indel register. The address generator supports both SRAM read/write operations and grid initialization. Computation begins by injecting a logic high into the top-left cell of the grid. The process completes at the bottom-right cell, where the signal arrival time encodes the final edit distance or score; this value is captured by the distance counter and forwarded to the control unit. The control unit then packages the results, after which the I/O controller transmits the output packets and proceeds to the next transaction and does so concurrently if in stream mode.

C. I/O Controller

The I/O controller provides the primary communication interface between the external host and our chip. Incoming traffic is delivered over the 14-bit `pkt_i[13:0]` bus, which carries both instruction and data packets into the system, while outgoing traffic is transmitted back to the host via the 13-bit `pkt_o[12:0]` bus.

As shown in Figure 6, these packet buses use a structured encoding in which individual bits are multiplexed across multiple roles depending on the packet type. This packed representation enables a single physical interface to support several classes of transactions without additional pins. Figure 6 details the bit-level breakdown of each packet format and the interpretation of the fields for different operations. The packet format follows a unionized bit structure, such that each traversal of an edge in the packet tree uniquely identifies a packet type. Here, tb stands for trace back.

Internally, the I/O controller is implemented as a pair of synchronous FIFOs, one for incoming packets and one for outgoing packets.

D. SRAM organization

In our design, the on-chip SRAM is responsible for storing the substitution scoring matrices and is sized to support up to 20 unique symbols. This design choice is motivated by our primary target application: protein sequence alignment using the BLOSUM62 matrix, which defines substitution scores for the 20 standard amino acids. A complete scoring matrix therefore consists of a 20×20 matrix containing 400 entries.

To reduce storage requirements, we exploit the inherent symmetry of substitution matrices. Since the matrix is symmetric about its diagonal, only the lower triangular portion (including the diagonal) can be stored. This reduces the total number of required entries to $\frac{N(N+1)}{2}$, which evaluates to 210 entries for $N = 20$, yielding nearly a 50% reduction in memory usage.

Fig. 7. Scoring matrix entry to SRAM address mapping

Figure 7 provides an illustration of the address mapping used to index scoring matrix entries within the SRAM.

The primary tradeoff of this optimization is increased lookup complexity, as the address generation logic must account for the ordering of the symbol pair when indexing into the matrix. The details of this lookup mechanism are discussed further in the address generator section.

Taking this storage requirement into account, along with the fact that the maximum substitution score in BLOSUM62 is 31, we selected a 216×6 single-port register-file SRAM IP. This configuration provides sufficient capacity and bit width while maintaining a modest design margin.

E. Sequence buffers and symbol encodings

The sequence buffers are implemented as serial-in, parallel-out (SIPO) shift registers and store the two sequences, P and Q, currently being processed. These buffers supply the symbol streams required by the alignment engine during computation.

Each symbol in the input sequences is represented using a 5-bit binary encoding. In our implementation, the 20 standard symbols are mapped to unique binary codes, with two additional encodings reserved to support variable-length sequence handling, as discussed later. The specific mapping of binary codes to symbols is not fixed by the architecture, but is instead determined by the ordering of the scoring matrix entries.

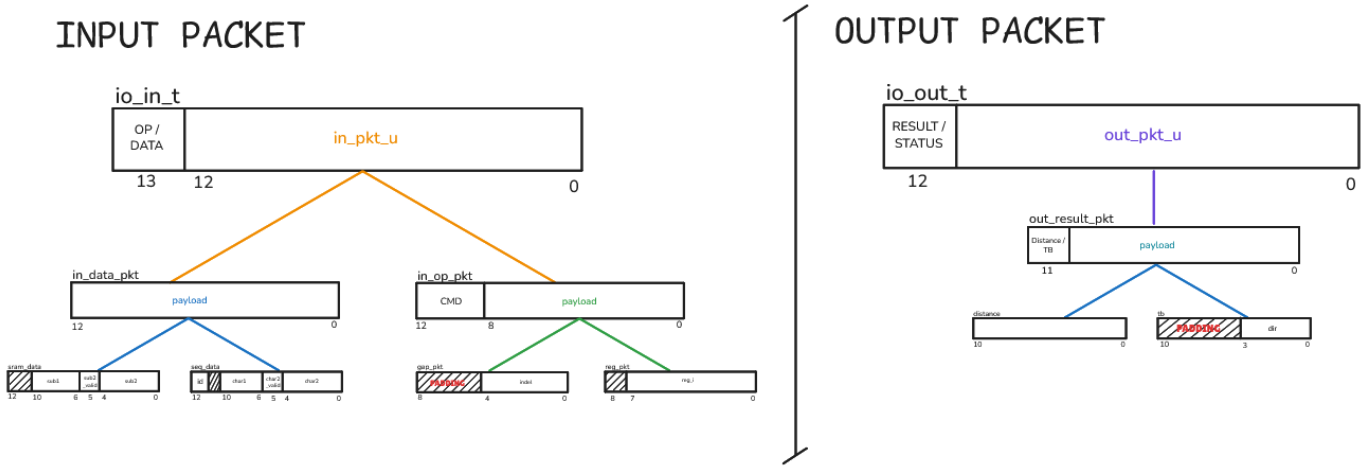

Fig. 6. I/O packet breakdown (click to expand)

Specifically, the encoding order must align with the layout of the scoring matrix stored in SRAM. The first entry in the matrix corresponds to a match between two symbols both encoded as zero, following a row-major traversal of the lower-triangular matrix as described in the SRAM section. Maintaining this correspondence ensures correct substitution score lookup during alignment.

F. Address Generator

The address generator provides the interface between the control logic and the scoring-matrix SRAM. Its implementation is shown in Figure 8. The module operates in two distinct modes that are controlled by the control unit: SRAM initialization and race-logic grid initialization.

During SRAM initialization (state `INIT_SRAM`), the address generator is used to write the scoring matrix into memory. In this mode, the control unit supplies sequential SRAM addresses along with the corresponding substitution scores, while the write-enable signal is asserted. This process populates the SRAM with the precomputed scoring matrix prior to alignment execution.

During race-logic grid initialization (state `INIT_GRID`), the address generator performs dynamic lookup of substitution scores for the currently processed symbol pair. The symbols $P[i]$ and $Q[j]$, where i and j correspond to the column and row indices of the grid, are retrieved from the sequence buffers and compared in their encoded form. Since the scoring matrix is symmetric, the symbol pairs (a, b) and (b, a) are equivalent. However, because only the lower-triangular portion of the matrix is stored, the address generator must reorder the pair such that the larger encoding appears first.

Once the symbol ordering is determined, the SRAM address is computed using a lookup table (LUT) that provides the base address for the corresponding row of the lower-triangular matrix. For example, encodings beginning at zero map to base addresses of 0, 1, 3, 6, and so on. An offset derived from the smaller symbol encoding is then added to this base address to produce the final SRAM index. This address generation process is illustrated in the lower portion of Figure 8.

While this approach may appear more involved than a direct arithmetic computation, it was intentionally chosen to avoid instantiating a hardware multiplier. Eliminating the multiplier reduces area overhead and better aligns with the design goal of maintaining a compact and resource-efficient implementation.

G. Control Unit

The Control Unit is the central controller of the D-PEARL accelerator, responsible for decoding host instructions, sequencing computation phases, and coordinating interactions between submodules such as the SRAM, sequence buffers, Race Logic grid, and output logic. It governs all high-level chip operations, from initialization to computation and result output, ensuring the system behaves deterministically and in accordance with the expected pipeline of dynamic programming execution.

The Control Unit (CU) receives 14-bit input packets from the external host through the I/O controller's incoming FIFO. Each packet encodes either an instruction or a data payload. The CU decodes the packet, determines its type, and issues the appropriate control signals to downstream modules. For outbound communication, the CU transmits 13-bit packets containing computation results to the outgoing FIFO. This bidirectional communication forms the foundation of how the host FPGA interacts with and supervises the accelerator.

The Control Unit operates as a finite state machine with several major states corresponding to distinct phases of operation. During the configuration and initialization phase, the CU handles host-driven loading of SRAM contents, indel penalties, configuration registers, and sequence data. After initialization, the CU transitions to the computation phase, where it prepares the Race Logic grid and triggers and monitors execution. Once computation completes, the CU enters the result handling

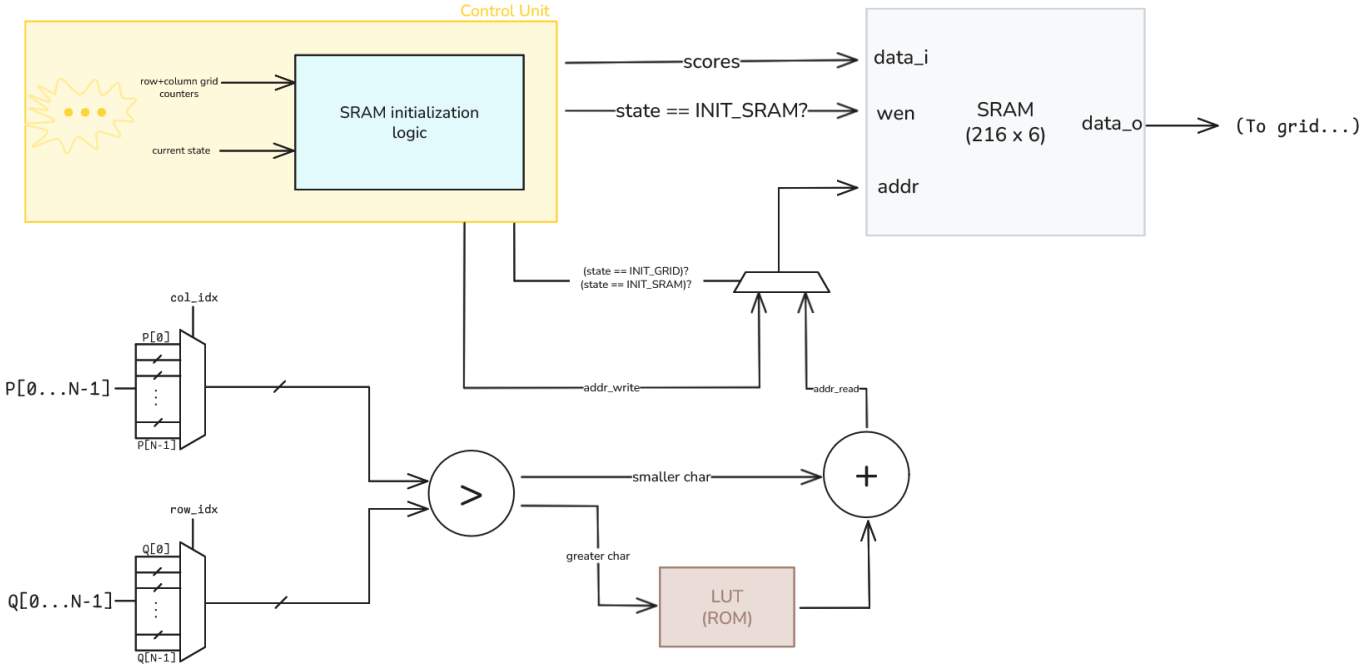

Fig. 8. Address Generator Diagram (click to expand)

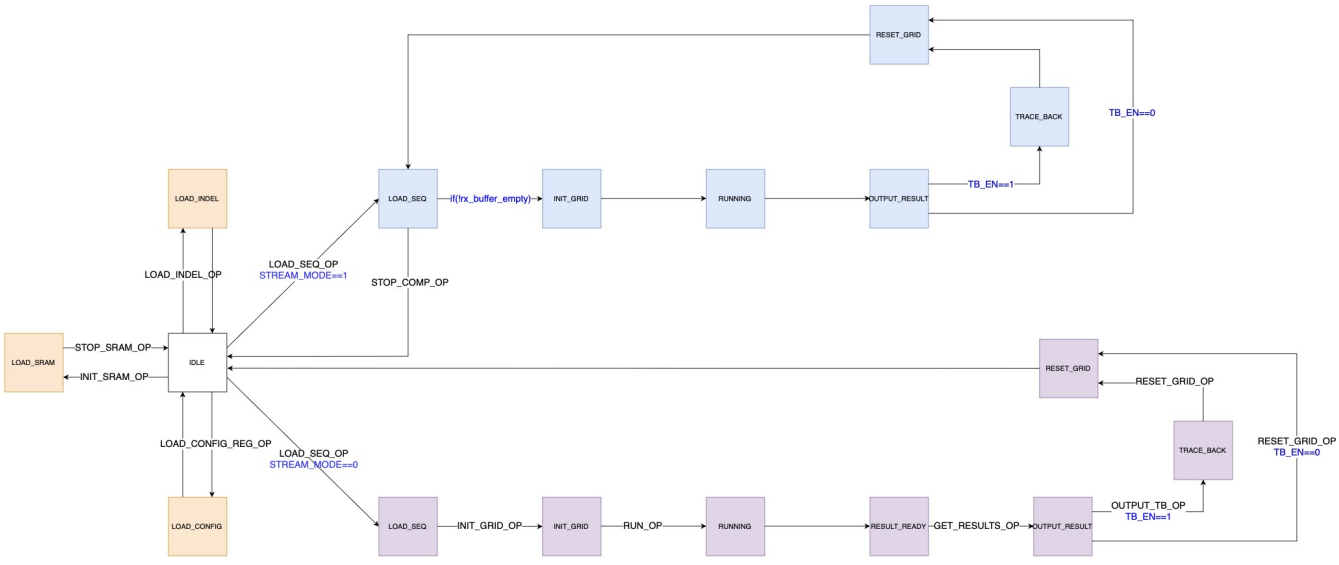

Fig. 9. Control Unit FSM (click to expand)

phase, which includes both alignment results and traceback data. The CU is also responsible for resetting submodules, such as the grid and sequence buffers, between workloads, as well as clearing key status flags.

The CU supports both streaming and non-streaming modes of operation. When stream mode is enabled, results are automatically output as soon as they are computed. When disabled, results are transmitted only after the host explicitly requests them. This flexibility allows the accelerator to adapt to different workload characteristics and host interface requirements.

During operation, the CU coordinates closely with the Address Generator to fetch substitution edge weights from SRAM and manages row-wise enable signals to drive data through the grid. It monitors completion conditions, such as detecting a logic high at the bottom-right cell or triggering a timeout if computation stalls.

H. Race Logic Mesh and Unit Cell

At the core of the D-PEARL architecture is a 27×27 two-dimensional mesh of race-logic unit cells. Collectively, this mesh represents an edit graph, a directed acyclic graph (DAG) commonly used to formulate edit distance and sequence alignment problems, as illustrated in Figure 10. Each unit cell corresponds to a node in the edit graph, and together the grid supports a race-logic wavefront that performs temporal computation across the graph.

Computation begins at the origin cell located at the top-left corner of the mesh. Cell metadata is initialized by shifting configuration data through the top row of the grid and propagating it downward to subsequent rows. This process is coordinated by the address generator and controlled by the global control unit.

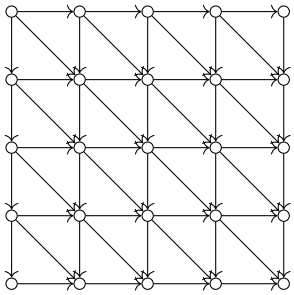

Fig. 10. Edit graph representation for two sequences of length 4

Within the edit graph, nodes represent dynamic programming states. Horizontal edges correspond to deletions, vertical edges correspond to insertions, and diagonal edges correspond to substitutions. Each edge is associated with a weight in the range of 1 to 31, which is encoded directly into the metadata of the destination cell. A given node can therefore be mapped to a race-logic circuit, where logical operations model state transitions and edge weights are represented as propagation delays.

As shown in Figure 11, each unit cell implements OR-based (min) race logic. This formulation is preferred over AND-based (max) race logic because the earliest-arriving signal determines the result, leading to lower computation latency. Delay is implemented using a synchronous saturating counter, and each cell stores metadata consisting of an insertion/deletion (indel) penalty, a substitution score, and a postfix flag.

All metadata is programmed during the `INIT_GRID` phase and is shifted into the grid rather than being individually addressed. The indel penalty determines the delay applied to vertical and horizontal transitions, while the substitution score governs diagonal propagation. The saturating counter compares the accumulated delay against these programmed values to decide when a temporal wave is permitted to propagate in each direction. Each outgoing edge includes a set-on-arrival feedback mechanism, ensuring that once the delay threshold is met, the corresponding input is permanently asserted until a new computation begins.

I. Path Reconstruction

The design supports hardware-backed path reconstruction by embedding traceback logic directly within each race-logic unit cell (Figure 11), eliminating the need for software-based backtracking and enabling efficient extraction of the optimal edit path.

Traceback follows the standard dynamic programming principle: the optimal path can be recovered by traversing the edit graph in reverse and selecting, at each node, the predecessor edge that contributed to the node's final score. In race logic, this is done by identifying the earliest-arriving signal for OR-based (minimum) computation.

Each unit cell includes a small set of per-input flip-flops (one for each incoming edge: top, left, top-left). Flip-flops are initially cleared; when a signal arrives, the corresponding flip-flop is set, and a local enable prevents others from toggling, ensuring only the winning transition is recorded. The resulting structure (Figure 12) uses three shift-enabled registers to capture the selected incoming direction. Traceback extraction exploits the mesh interconnectivity: direction bits are shifted serially through the grid to form traceback packets, removing the need for a centralized memory or controller.

Each traceback packet is encoded in a `tbk_t`, which stores the direction of the winning transition for a single unit cell:

```
typedef struct packed {
    logic left;
    logic tl;
    logic top;
    logic disabled;
} tbk_t;
```

The origin cell contains a hard-coded zero-valued `tbk_t`, which serves as the termination condition for path extraction. Since all valid paths originate at this cell, encountering the origin implicitly signals the completion of the traceback process.

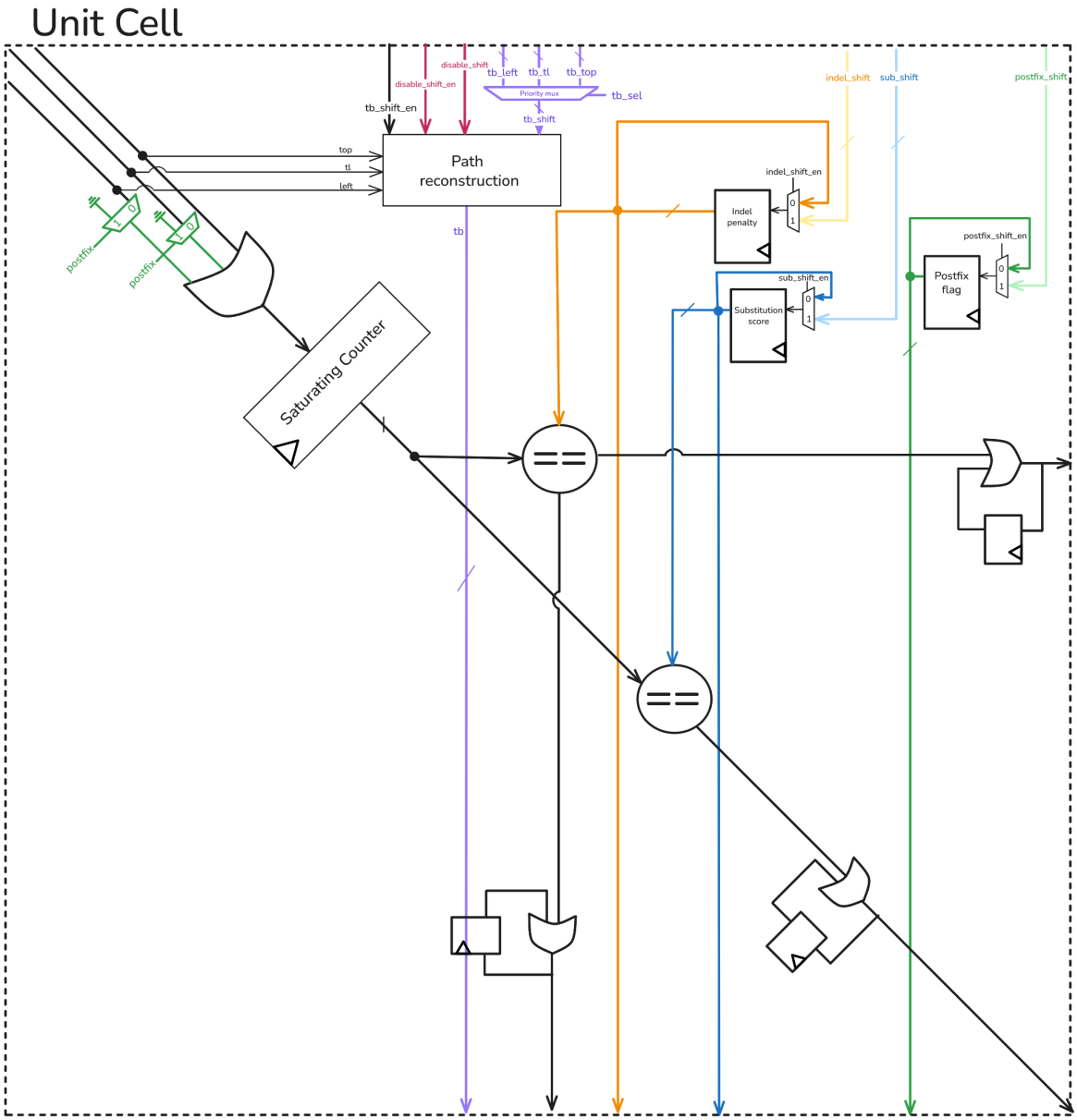

Fig. 11. Unit cell diagram (click to expand)

J. Alignment of Different Sequence Lengths

To support alignment of sequences with differing lengths, we introduce special padding symbols, PREFIX_CHAR (-) and POSTFIX_CHAR (?), which ensure that the input sequences properly map onto the fixed-size Race Logic grid without requiring dynamic resizing or reconfiguration.

The PREFIX_CHAR is added to the beginning of shorter sequences so that both input sequences extend to the full grid length. This guarantees that computation begins at the correct spatial offset within the grid. When a PREFIX_CHAR is matched with another PREFIX_CHAR, the delay between them is defined as 1, representing a minimal propagation delay. In contrast, when a PREFIX_CHAR is matched with any other character, the delay is set to the maximum allowable delay value, effectively disabling that path until the valid computation region is reached. This mechanism ensures that the Race Logic signal cleanly propagates through the padding region into the active sub-grid where meaningful alignment occurs. After computation, the user subtracts the number of PREFIX_CHARs from the reported alignment score to recover the final, position-correct result.

The POSTFIX_CHAR serves a complementary role in handling trailing sequence mismatches when the two input sequences differ in length. POSTFIX_CHARs are appended to the shorter sequence such that the overall sequence lengths are equalized. In hardware, POSTFIX_CHARs are configured

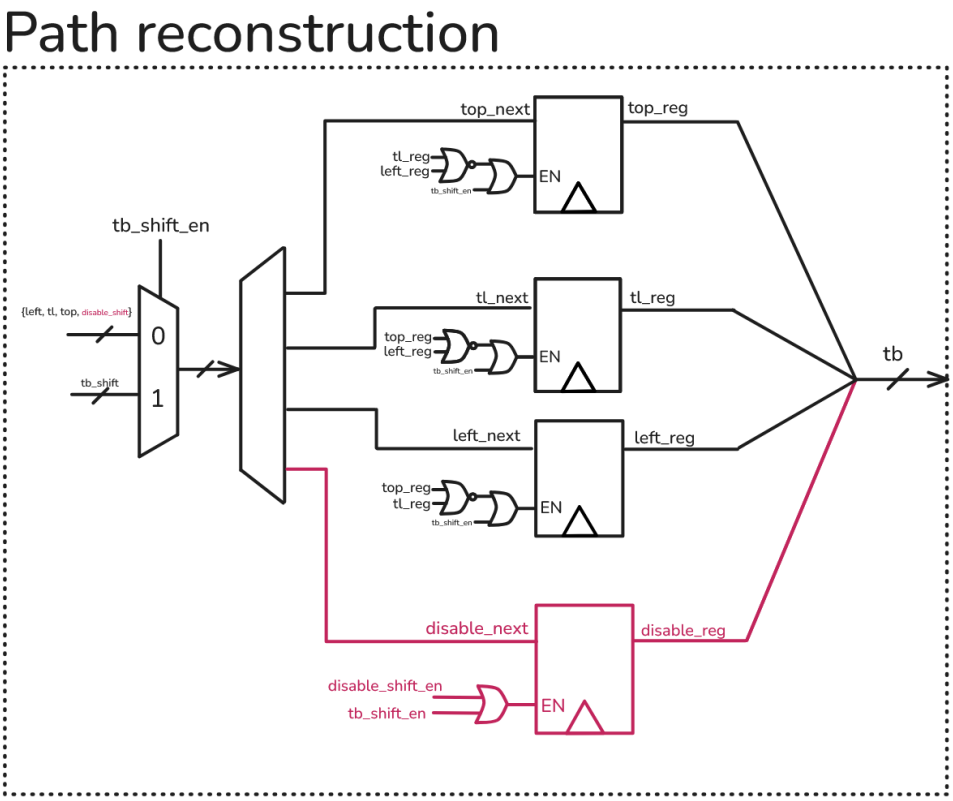

Fig. 12. Path reconstruction circuit (click to expand)

to only allow downward signal propagation, effectively forcing the alignment to terminate at the bottom-right corner of the sub-grid where the valid computation concludes. This ensures that the alignment result is retrieved correctly with minimal additional area overhead, as only the active computation region of the grid participates in the delay-based evaluation. Note that only sequence Q supports POSTFIX chars, since we always assume it to be the shorter sequence.

Consider two sequences P=AGTC, and Q=GTC for sequence buffers of length 5. To align these sequences properly, both will be padded with a PREFIX char, until longer sequence (P) has the same length as the sequence buffers. After, the shorter sequence (Q) will be padded with POSTFIX until it has the same length as the sequence buffers. For this example, the padded input sequences will be P=-AGTC, and Q=-GTC?.

VI. Simulation Guide

To verify the design using the Synopsys VCS toolchain, follow the steps below:

A copy of the project can be found in:

```
/groups/ece427-group4/ahmeds4/D-PEARL
```

Alternatively, the repository can be cloned from the following link:

```
https://github.com/ahmed23shaf/D-PEARL.git
```

(NOTE: the repository is private; contact ahmeds4@illinois.edu for access.)

Directory Overview

- `PnR/`: Automated PnR Tcl and Python scripts
- `syn/`: Synopsys Design Compiler setup for synthesis
- `rtl/hdl/`: RTL source (top-level: `d_pearl.sv`)
- `rtl/hvl/`: Testbenches, golden model, coverage, file I/O helpers
- `rtl/bin/`: Utility Python scripts, directed test vectors, verbose outputs
- `rtl/inc/`: Include files (macros, packages, constants)

Prerequisites

Install the BLOSUM Python module:

```
python3 -m pip install blosum
```

Navigate to the RTL directory (or an equivalent cloned one):

```
/groups/ece427-group4/ahmeds4/D-PEARL/rtl
```

Behavioral Simulation

a) Constrained randomized testing:

- **`rtl/inc/macros.sv`**: Comment out `ENABLE_SIM_VERBOSE` and `SIM_VECTOR`. Enable `ENABLE_SIM`. Enable only one of `SIM_BLOSUM` or `SIM_ATCG`.

- **rtl/inc/types.sv**: Set SEQ_LEN (note: SEQ_LEN > 25 increases runtime) and NUM_SYMBOLS to match the chosen symbol set.
- **Optional** (rtl/hvl/dpearl_tb.sv): Adjust num_iterations.
- **Run**: make vcs/top_tb

The testbench reports the number of successful iterations and total num_computations (often higher due to back-to-back computations).

b) Directed testing:

- **rtl/inc/macros.sv**: Enable ENABLE_SIM, ENABLE_SIM_VERBOSE and SIM_VECTOR. Select SIM_BLOSUM or SIM_ATCG.
- **Vectors**: Modify the appropriate *.csv test vectors in rtl/bin/tests/ (follow the existing format; both sequences per line must use the same SEQ_LEN).
- **rtl/inc/types.sv**: Set SEQ_LEN and NUM_SYMBOLS accordingly.
- **Run**: make vcs/top_tb

Results are written to rtl/bin/verbose/, with one subfolder per test pair (indexed from 0), containing:

- dut_output.txt: Output per test
- dut_trace.txt: Path reconstruction and trace

Results are cross-checked against the golden model and assertions.

Post-Synthesis (LEC) Simulation

c) Constrained randomized testing:

- **rtl/inc/macros.sv**: Disable ENABLE_SIM, ENABLE_SIM_VERBOSE, SIM_VECTOR. Enable SIM_BLOSUM (or SIM_ATCG if the netlist was synthesized for ATCG).
- **Optional** (rtl/hvl/synth_verif/synth_dpearl_tb.sv): Adjust num_iterations.
- **Run**: make no_synth_sdf

d) Directed testing:

- **rtl/inc/macros.sv**: Disable ENABLE_SIM. Enable ENABLE_SIM_VERBOSE and SIM_VECTOR. Select SIM_BLOSUM.
- **Vectors**: Edit rtl/bin/tests/vector_blosum.csv. Use SEQ_LEN = 27 if not using custom synthesis.
- **Run**: make no_synth_sdf

Results under rtl/bin/verbose/:

- dut_output.txt, dut_trace.txt

Note: at gate-level, due to limited visibility, dut_trace.txt and parts of dut_output.txt will be largely empty.

Post-PnR Simulation

e) Constrained randomized testing:

- **rtl/inc/macros.sv**: Disable all except SIM_BLOSUM.
- **Optional** (rtl/hvl/PnR_verif/PnR_dpearl_tb.sv): Adjust num_iterations.
- **Run**: make pnr_tb

f) Directed testing:

- **rtl/inc/macros.sv**: Enable SIM_BLOSUM, SIM_VECTOR, ENABLE_SIM_VERBOSE (disable everything else).
- **Vectors**: Edit rtl/bin/tests/vector_blosum.csv. Use SEQ_LEN = 27 (similar reason as in post-synth).
- **Run**: make pnr_tb

Results under rtl/bin/verbose/ (dut_output.txt, dut_trace.txt); visibility limitations apply as in post-synthesis.

PD Verification

The gate-level netlists referenced in the overview above are located in:
/groups/ece427-group4/final_GDS/D-PEARL.

The final GDSII file used for submission, along with the files used for physical design (PD) checks, are located in:
/groups/ece427-group4/drchaw2.

VII. Acknowledgment

We would like to thank Apple for sponsoring this course and Muse Semiconductor for facilitating the final stages of the tapeout process. We also thank Professor Dong Kai Wang and the course TAs, Stanley Wu and Jerry-Tianchen Wang, for their guidance, feedback, and support throughout the project. Finally, we would like to thank our Apple mentor, Skanda Srinivasa, for sharing his industry experience and providing valuable insight during the course of this work.

References

[1] A. Madhavan, T. Sherwood, and D. Strukov, "Race logic: A hardware acceleration for dynamic programming algorithms," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, Minneapolis, MN, USA, 2014, pp. 517–528, doi: 10.1109/ISCA.2014.6853226.