



Cairo University
Faculty of Engineering
Computer Engineering Department

CMP3020 — VLSI

Lab Assignment 4

3 Dec 2025 • Datapath Optimization

Muhammad Sayed

Who knows? We might find peace by daring to try

INTRODUCTION

In digital VLSI design, the "Iron Triangle" of PPA (**P**ower, **P**erformance, and **A**rea) requires constant balancing.

Previous labs focused on optimization basics. This lab explores **Pipelining** and **Stream Processing**. While parallelization increases throughput by duplicating hardware, pipelining increases throughput by overlapping the execution of sequential tasks. This technique allows for higher clock frequencies and ensures that arithmetic units remain active, minimizing the "busy cycles" often found in non-pipelined architectures.

The focus of this analysis is the atomic unit of Artificial Intelligence: the **Dot Product**.

$$Y = \sum_{i=0}^{N-1} (A_i \cdot B_i)$$

This single operation is the computational backbone of Deep Learning, powering the massive Matrix Multiplications (GEMM) and Convolutions found in every modern Neural Network accelerator (such as TPUs and GPUs).

Traditional "Store-then-Compute" architectures suffer from latency bottlenecks, as entire vectors must be buffered before processing can begin. The objective is to analyze these limitations and design a **Streaming Pipelined** architecture capable of processing scalar inputs continuously, thereby maximizing hardware utilization and system throughput.

THE BASELINE: SEQUENTIAL DESIGN

The provided module, `dot_product.v`, is an area-minimized implementation. A single Multiplier and a single Adder (MAC unit) are utilized to process the vectors over N clock cycles.

Architecture Overview

The design relies on a **State Machine** to manage the Handshake Protocol (valid/ready) and a **Counter** to slice the input vectors one element at a time.

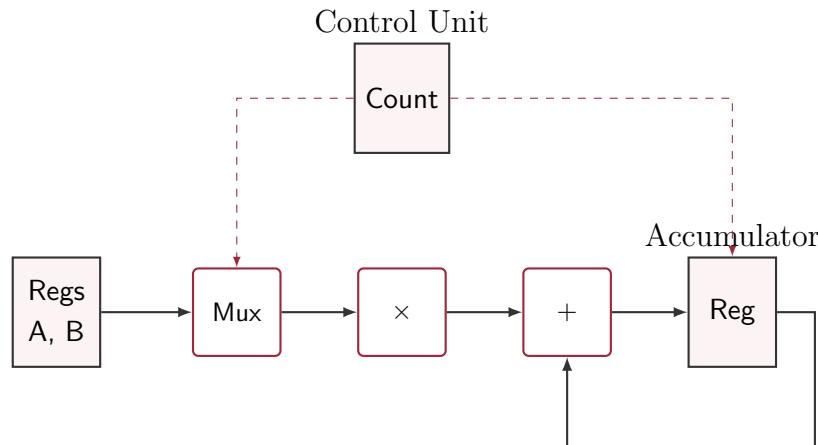


Figure 1: Sequential Architecture: Reusing Logic over Time

The figure above illustrates the "Time-Multiplexed" architecture. Instead of calculating the entire vector at once, the system uses a control loop to reuse hardware:

- **Select:** In every clock cycle, the **Mux Logic** extracts one pair of elements (A_i, B_i) from the Input Registers.
- **Compute:** These values are fed into a single, shared Multiply-Accumulate (MAC) unit.
- **Accumulate:** The result is stored in the **Accumulator** register and fed back to the adder to build the final sum step-by-step.

Provided Implementation

The source file `dot_product.v` is provided as a reference. This module implements the sequential baseline described above. Key features of this reference design include:

- **Full Parameterization:** The module accepts `WIDTH` and `N` parameters, allowing it to scale to any vector size without code modification.
- **Handshake Interface:** A robust `valid/ready` protocol is implemented. Crucially, `input_ready` goes **LOW** immediately after accepting data, signaling that the core is busy processing.
- **Dynamic Slicing:** Verilog's Indexed Part-Select syntax (`+:`) is utilized to dynamically multiplex the large input vectors into the single multiplier based on the current counter value.
- **Resource Minimization:** Exactly **one** Multiplier and **one** Adder are instantiated, regardless of the vector length.

DESIGN TASK: STREAMING PIPELINE

The primary objective is to create a new module, `dot_product_stream.v`, that processes scalar inputs continuously using a pipelined datapath to maximize frequency and utilization.

The Temporal Architecture

Instead of instantiating N multipliers in space (Parallel) or blocking input for N cycles (Sequential), a single Pipelined MAC unit is utilized to process a continuous stream of scalar inputs.

- **Latency:** The computation requires N cycles to ingest the vector plus the pipeline depth to drain the result. For a 2-stage pipeline, the total latency for the first result is $N + 2$ cycles. However, due to the reduced critical path, the duration of these cycles (T_{clk}) is significantly shorter than the sequential baseline.

- **Throughput:** The design accepts a new scalar input pair (A_i, B_i) **every cycle**. Consequently, for a vector length N , a new dot product result is valid every N cycles. Crucially, these are **shorter cycles** (lower T_{min}) due to the reduced critical path, resulting in higher real-time throughput compared to the baseline.
- **Control:** The pipeline logic must be **self-draining**. Computation must continue for data already inside the pipeline even if the input stream pauses (valid drops to 0).

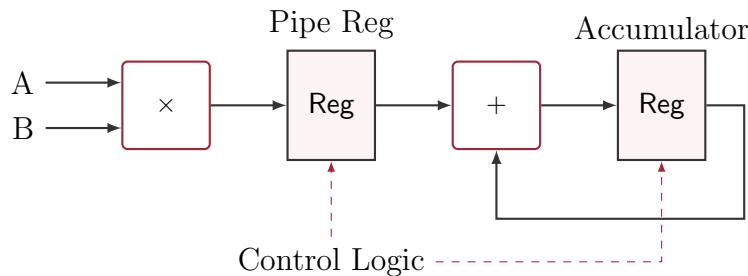


Figure 2: Temporal Architecture: 2-Stage Pipeline

Implementation Strategy

1. **Pipeline Stage 1 (Multiplication):** Compute the product of the incoming scalar inputs ($P = A \times B$) and store the result in an intermediate pipeline register.
2. **Pipeline Stage 2 (Accumulation):** Add the registered product to the running accumulator count on the subsequent clock cycle.
3. **Flow Control:** Implement self-draining logic. If `input_valid` goes low, the input stage stops accepting new data, but the pipeline registers must continue to propagate existing data to the accumulator.
4. **Output Logic:** The `output_valid` signal must be asserted for exactly one cycle when the summation of the current vector is complete (every N cycles during continuous operation).

VERIFICATION

Verification for this lab differs from the previous assignment because the module interface changes from **Vector-based** (Baseline) to **Scalar-based** (Streaming). Therefore, a new testbench is required for the pipelined design.

Verifying the Baseline

Before implementing the pipeline, the provided sequential module must be verified to establish a performance baseline using the standard vector-based testbench.

1. **Environment Setup:** Ensure the OpenLane Nix Shell is active.
2. **Execution:** Run the verification script targeting the sequential implementation.

```
./scripts/verify.sh sequential dot_product_tb.v
```

Verifying the Pipelined Design

Since the `dot_product_stream.v` module accepts one element per cycle, the original testbench cannot be used directly. A new testbench, `src/dot_product_stream_tb.v`, must be created.

Testbench Requirements

The testbench must robustly verify the pipeline by implementing the following logic:

- **Scalar Driver Task:** Implementation of a helper `task` is recommended. This task should iterate N times to drive the scalar inputs (A, B) cycle-by-cycle while asserting `input_valid`.
- **Functional Parity:** The four mandatory test scenarios (Identity, Weighted Sum, Zero Multiplication, and Saturation) must be re-executed. The arithmetic results must match the baseline exactly.
- **Self-Draining Check:** Verify that if `input_valid` is de-asserted immediately after the N -th input element, the pipeline continues to process the inflight data and successfully asserts `output_valid` after the pipeline depth delay.

Execution

Once the testbench is created, verify the design using the script:

```
# Usage: ./scripts/verify.sh <type> <new_tb_file>
./scripts/verify.sh stream dot_product_stream_tb.v
```

Successful execution will generate a waveform file in `outputs/waves/`, demonstrating the continuous flow of data and the correct draining behavior.

SYNTHESIS & PHYSICAL IMPLEMENTATION

With the RTL functional verification complete, the next step is to push the design through the OpenLane flow to generate the physical layout and extract accurate PPA metrics.

Baseline Synthesis

A configuration file, `configs/config_sequential.json`, is provided for the sequential design. To ensure OpenLane can locate the source files correctly while keeping the configuration separate, the flow must be run from the **Lab Directory** (the parent of `src` and `configs`).

1. Open the terminal in the root of the lab folder (e.g., `lab-4/`).
2. Execute the following command to synthesize the baseline:

```
openlane configs/config_sequential.json --design-dir . --run-tag  
sequential_run
```

This will generate the baseline metrics in `runs/sequential_run/reports/metrics.csv`.

Pipelined Synthesis

A configuration for the streaming design must be created.

1. Duplicate `configs/config_sequential.json` and rename it to `config_stream.json`.
2. Update the `VERILOG_FILES` path to point to the new `src/dot_product_stream.v`.
3. **Constraint Freeze:** Do NOT change `FP_CORE_UTIL`, or `PL_TARGET_DENSITY_PCT`.
 - Scientific comparison requires that the environment remains constant. Changing the utilization or density would skew the Area results.
4. Run the pipelined synthesis:

```
openlane configs/config_stream.json --design-dir . --run-tag  
stream_run
```

Post-Synthesis Verification (GLS)

Synthesis tools can sometimes alter logic incorrectly, or timing violations might break functionality. **RTL verification is not enough.**

Creation of a new script (e.g., `scripts/verify_gls.sh`) is strongly encouraged to verify the final **Pipelined Netlist**.

- **Target:** Compile the synthesized netlist (`runs/stream_run/results/final/spi/lvs/dot_product_stream.spi` or `.v`) instead of the RTL.
- **Dependencies:** The PDK primitives (standard cell Verilog models) must be included in the compilation.
- **Goal:** Run the *exact same* testbench (`dot_product_stream_tb.v`) against the physical netlist. If it passes, the hardware is valid.

DELIVERABLES & ANALYSIS

Submission Requirements

A single `.zip` file must be submitted to Google Classroom. The directory structure must be preserved.

- **Penalty:** Submissions not in a `.zip` format will be penalized.

- **Contents:**

- `src/dot_product_stream.v` (The RTL Design).
- `src/dot_product_stream_tb.v` (The Testbench).
- `configs/config_stream.json` (The OpenLane Config).
- `ANALYSIS.md` (Answers to the questions below).
- `waves/` (Screenshots of the successful testbench waveforms).

Waveform Evidence

Clear, readable screenshots of the waveforms for the Pipelined design must be provided.

- **Requirement:** The waveform must demonstrate continuous throughput (a new valid output is asserted every N cycles while the stream is active) and the **Self-Draining** behavior (the final output appears 2 cycles after the input stops).
- **Penalty:** Unreadable or missing screenshots will result in grade deduction.

Analysis Questions (`ANALYSIS.md`)

The following questions must be answered based on the `metrics.csv` and `runs.log` files generated by OpenLane.

1. Area Comparison

Compare the physical size of the Sequential Baseline versus your Pipelined Implementation.

- **Metric Selection:** Which specific key in `metrics.csv` provides the most accurate comparison of the logic hardware cost? Explain why `design_die_area` might be misleading in this context.
- **Growth Trend:** Unlike the Parallel design, the Pipelined design reuses the same arithmetic units. Calculate the percentage change in area observed in your run ($N = 4$) compared to the baseline:

$$\% \text{ Change} = \frac{\text{Area}_{\text{pipe}} - \text{Area}_{\text{seq}}}{\text{Area}_{\text{seq}}} \times 100$$

- **Physical Reason:** Explain the structural source of any area increase. Since the Multiplier/Adder count is identical (1 each), what specific components (Registers/Flip-Flops) account for the difference?

2. Timing & Throughput

Using the `timing_setup_ws` (Worst Slack) from the logs, calculate the **Minimum Clock Period** (T_{min}) for both designs using the formula: $T_{min} = T_{clk} - \text{Slack}$.

- **Frequency Analysis:** Which design has a higher maximum frequency (F_{max})? Explain how inserting a Pipeline Register affects the critical path length compared to the Sequential baseline.
- **Single Vector Latency:** Calculate the time required to process a **single input vector** for both designs. Note that the Sequential design requires N cycles, while the Pipelined design requires $N + \text{Depth}$ cycles.

$$T_{latency} = (\text{Cycles per Vector}) \times T_{min}$$

Calculate the percentage difference in processing time:

$$\% \text{ Difference} = \frac{T_{latency_seq} - T_{latency_pipe}}{T_{latency_seq}} \times 100$$

- **Throughput Scale:** Calculate the total time required to process **1,000 vectors** continuously. Does the higher clock speed (F_{max}) of the Pipelined design result in a significant speedup for large workloads, or is the benefit negligible?

3. Log Analysis (Warnings)

Examine the synthesis and PnR logs. Locate the specific warnings regarding SDC files and Timing Violations to answer the following:

- **SDC Definition:** The logs mention using a fallback SDC file. What is an SDC (Synopsys Design Constraints) file? Give a brief definition of its purpose in the design flow.
- **Flow Stages:** The log checks for two different SDC files: one for PnR and one for Signoff. Why do physical design flows typically require different constraint files for these two stages?
- **Corner Decoding:** Decode the corner name `max_ss_100C_1v60`. Explain exactly what the following terms represent:
 - **max** (regarding RC Corner)
 - **ss** (regarding Transistor Process)
 - **100C** (regarding Temperature)
 - **1v60** (regarding Voltage)
- **Critical Violations:** The tool reported setup and slew violations **only** in the slow corners (e.g., `max_ss_100C_1v60`).
 - Briefly explain the physical meaning of a **Setup Violation** versus a **Max Slew Violation**.

- **Real-World Scenario:** Based on the corner name, describe the specific physical conditions under which this chip would fail.
 - Suggest one specific configuration change (in `config.json`) that could fix these violations without changing the Verilog RTL.
- **Flow Continuity:** Did the OpenLane flow **stop immediately** (crash) when these timing violations were detected? If not, where exactly (which file) must a verification engineer look to definitively confirm if the chip passed or failed timing?