CMP3020 — VLSI

Course Project

• Domain Specific Accelerator •

Muhammad Sayed

Your chaos is entirely valid

SYSTEM ARCHITECTURE

Operational Overview

The accelerator functions as a streaming coprocessor. It performs 2D convolution by buffering tiles of the input image and kernel weights into on-chip memory. The architecture is defined by three main functional blocks:

1. **Address Generation Unit (AGU):** This is the "brain" of the memory controller. Since the input is a 2D matrix but the memory is 1D, the AGU calculates the complex read/write addresses required to fetch 8×8 tiles and handle sliding window patterns.

2. **Memory Subsystem:** Acts as a high-speed cache between the slow external DRAM and the fast compute core. It stores input pixels, weights, and the calculated output pixels.

3. **Compute Core:** An 8×8 Systolic Array. It reads data from the Memory Subsystem, performs Multiply-Accumulate (MAC) operations, and streams the results back to memory.

High-Level Block Diagram

The diagram below illustrates the data flow, highlighting the separation between the **Off-Chip** DRAM and the **On-Chip** accelerator logic.

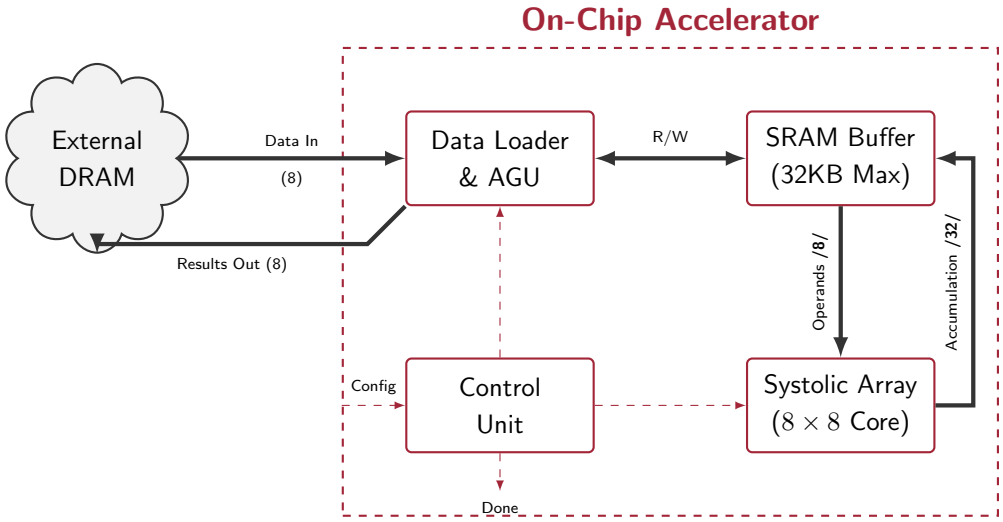

Figure 1: System Architecture: Data Path and Memory Hierarchy

TECHNICAL SPECIFICATIONS

Operational Parameters

The module must support the following processing parameters.

Parameter	Min	Max	Type/Unit
Input Matrix (N)	16×16	64×64	Variable
Kernel Size (K)	2×2	16×16	Variable
Stride (S)	1	1	Fixed*
Padding (P)	0	0	Fixed*
Arithmetic Precision			
Input/Weights	8-bit Unsigned		Fixed Point
Accumulation	32-bit (Internal)		Fixed Point (Max Reg)
Output Result	8-bit Unsigned		Truncated

** These are constant architectural features, not input ports.*

Table 1: Operational Parameters and Arithmetic Precision

Hardware Constraints

These parameters define the physical limits of the implementation.

Resource	Min	Max	Notes
On-Chip Memory	4 KB	32 KB	Total SRAM
Systolic Array	4×4	8×8	Processing Elements
Register Size	8-bit	32-bit	Data path registers
External Bus	8-bit	32-bit	DRAM Interface
Internal Bus	8-bit	128-bit	SRAM $\leftrightarrow$ Array

Table 2: Physical Hardware Constraints

Interface Definition

The design acts as an AXI-Stream-like slave. Data transfer is governed by a Valid / Ready handshake to manage backpressure.

Signal Group	Name	Dir	Description
Global	clk	In	System Clock
	rst_n	In	Active-Low Asynchronous Reset
Control	start	In	Pulse to begin computation
	cfg_N	In	Input Matrix dimension (N)
	cfg_K	In	Kernel dimension (K)
	done	Out	Asserted when entire output is written
Data Stream	rx_data	In	**8 to 32-bit** Input stream (matches bus)
	rx_valid	In	DRAM has valid data to send
	rx_ready	Out	Accelerator is ready to accept data
Output Stream	tx_data	Out	**8 to 32-bit** Output stream (matches bus)
	tx_valid	Out	Accelerator has valid result to send
	tx_ready	In	DRAM is ready to accept result

Table 3: Top-Level Interface Signals

THEORY OF OPERATION

This section bridges the gap between high-level concepts and hardware implementation. It details the architectural style, the atomic building blocks, and the protocols required for the design.

Systolic Architecture Overview

A **Systolic Array** is a network of identical processing elements (PEs) that rhythmically compute and pass data through the system. The goal of this architecture is to maximize **data reuse** and **throughput** by pumping data from one PE to the next, rather than fetching it from memory for every single operation.

The most intuitive example is **Matrix Multiplication**. As illustrated in the **Portland State University (PDX) Lecture Slides**, data is often "skewed" in time so that the correct row of Matrix A meets the correct column of Matrix B at the exact right moment inside the array.

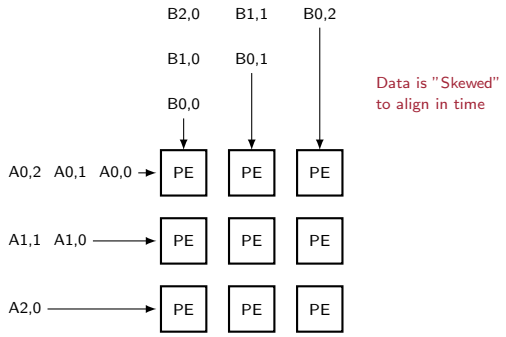

Figure 2: Concept: Rhythmic Data Flow (Time Alignment)

For a visual explanation of this flow, the **NJIT Systolic Array Visualization** video demonstrates how the "wavefront" of computation moves across the chip.

The Processing Element (PE)

To enable the grid operation, every node in the network must be identical. This atomic unit is called the **Processing Element (PE)**. Its function is to perform a specific arithmetic operation (usually Multiply-Accumulate) and forward data to its neighbors.
The schematic below illustrates a **Baseline Example** of a PE.
Note on Flexibility: This architecture is *not restrictive*. Modification of the internal logic of the PE to support specific calculation methods is permitted. For example, extra registers may be required to buffer the next weight while computing, or additional control logic included to handle accumulation resets and data validity flags.

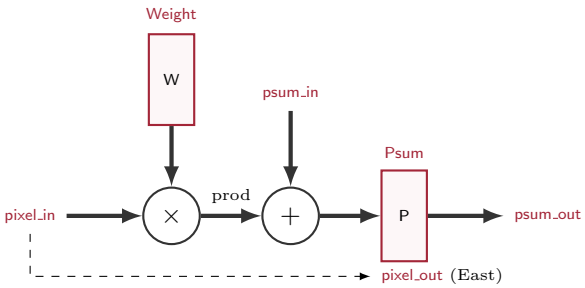

Figure 3: Example PE: A Multiply-Accumulate (MAC) Unit

Data Flow Taxonomies

The "Data Flow" defines which operands stay fixed inside the PE and which ones move across the array. Choosing the correct dataflow is critical for minimizing memory access energy. There are three primary taxonomies:

- **Output Stationary (OS):** The **Partial Sums** stay fixed in the PE accumulating results, while Inputs and Weights flow across the array. (This is the method used in the PDX lecture slides).
- **Input Stationary (IS):** The **Input Pixels** stay fixed, while Weights and Partial Sums flow.

- **Weight Stationary (WS):** The **Weights** are pre-loaded and stay fixed, while Input Pixels and Partial Sums flow.

Recommendation: For convolution operations where a small kernel is applied repeatedly across a large image, a **Weight Stationary** approach is generally preferred. It minimizes the energy cost of reading weights and simplifies the control logic for kernel tiling.

Memory & Data Management

Efficient data movement is just as critical as computation. This section details how to store data using SRAM IPs and how to manage the flow to keep the array active.

SRAM Integration & IP Models

In Digital Design, while small arrays can be synthesized directly into Flip-Flops, the large storage required by this project (up to 32KB) must be implemented using **Hard Macros** (SRAM blocks) to meet area and power constraints. must be implemented using **Hard Macros** (SRAM blocks) to meet area and power constraints. The provided Verilog simulation models must be used to interface with them correctly.

Method A: Standard Configuration For most designs, the standard configurations provided by the VLSIDA Sky130 SRAM Macros Repository are sufficient.

- **Available Sizes:** The repository provides pre-hardened macros in 1KB, 2KB, and 4KB configurations.
- **Recommended Type:** The 1rw1r configuration (Pseudo-Dual Port) is required. This allows one Read and one Write operation every cycle, enabling the Ping-Pong buffering scheme.
- **Required Files:** The .v file is required for simulation, along with the .lef and .gds files for the physical design flow.

Method B: Custom Generation If the selected tiling strategy requires a specific memory size not found in the standard list (e.g., optimizing for exact tile dimensions to save area), the **OpenRAM Memory Generator** can be utilized.
Generation Procedure:

1. **Clone the Tool:**

```
git clone https://github.com/Baungarten-CINVESTAV/SKY130-Macro-
    Memory-Cell-Generator.git
```

2. **Run the Generator:**

```
python3 imem_generator.py [mt] [wn] [ad] [p]
```

3. **Configuration Parameters:**
   - mt (Memory Type): Base cell type ($0 = 8 \times 1024$, $1 = 32 \times 256$, $2 = 32 \times 512$).
   - wn (Word Width): The data bus width (e.g., 32).
   - ad (Addresses): Total depth needed (e.g., 2048).
   - p (Placement): Physical layout shape (g=grid, r=row, c=column).
4. **Output:** The script generates a designs/ folder containing the required Verilog models and OpenLane configuration files.

Interface & Control Logic

Examination of the Verilog model (e.g., sram_1rw1r_32_256_8_sky130.v) reveals specific control signals.

- **Active Low Signals:** Signals ending in b (like csb0, web0) are Active Low. They must be driven to **0** to activate.
- **Chip Select (csb0):** The "Main Switch".
  - 1: Port is disabled (Power saving).
  - 0: Port is enabled for operation.
- **Write Enable (web0):** Determines the mode.
  - 1: **Read Mode** (Data appears on dout0).
  - 0: **Write Mode** (Data from din0 is written).
- **Write Mask (wmask0):** Allows byte-level writing. For a 32-bit word, a mask of 4'b0001 writes only the lowest 8 bits.

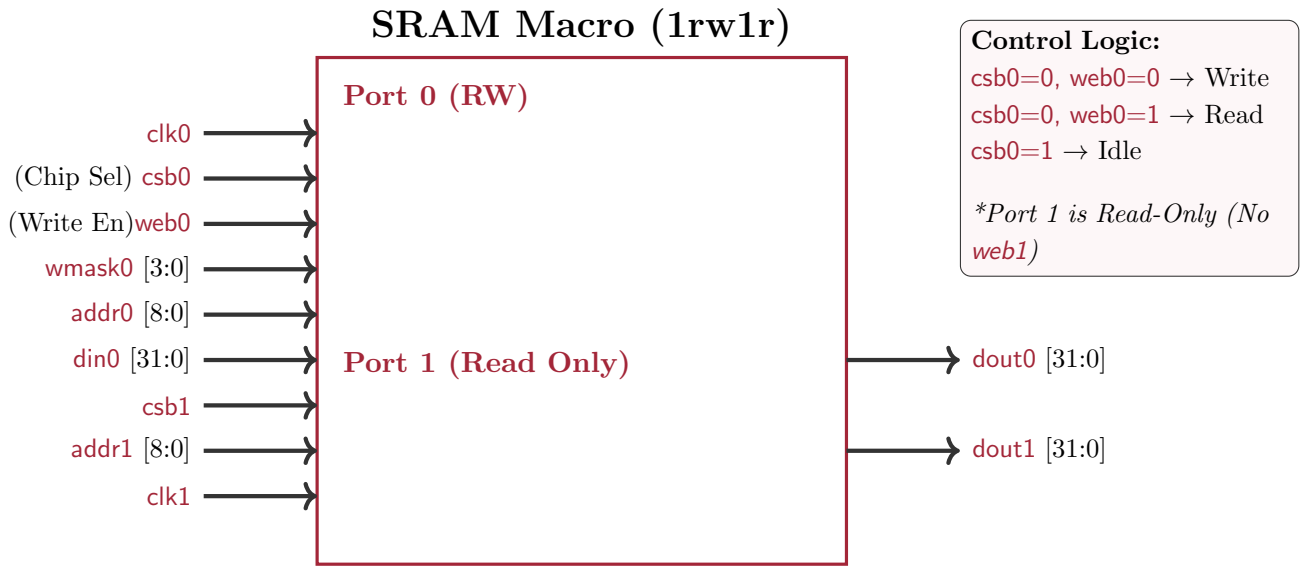

Figure 4: SRAM Interface Signals (Based on sky130_sram_1rw1r models)

The following Verilog snippet demonstrates how to instantiate the macro shown above and perform Read/Write operations.

```
// Define internal signals
wire [31:0] sram_dout0, sram_dout1;
reg  [31:0] sram_din0;
reg  [7:0]  sram_addr0; // 256 words = 8-bit address
reg         sram_csb0;  // Chip Select (Active Low)
reg         sram_web0;  // Write Enable (Active Low)
reg  [3:0]  sram_wmask0;// Write Mask (1 bit per byte)

// 1. INSTANTIATION (Matches the Diagram)
sram_1rw1r_32_256_8_sky130 memory_inst (
    // Port 0: READ / WRITE Port
    .clk0   (clk),
    .csb0   (sram_csb0),  // 0 = Active
    .web0   (sram_web0),  // 0 = Write, 1 = Read
    .wmask0 (sram_wmask0),// 4'b1111 to write all bytes
    .addr0  (sram_addr0),
    .din0   (sram_din0),
    .dout0  (sram_dout0), // Data available 1 cycle after Read

    // Port 1: READ ONLY Port (Used for simultaneous reads)
    .clk1   (clk),
    .csb1   (1'b1),       // Set to 0 to enable Port 1
    .addr1  (8'b0),       // Address for Port 1
    .dout1  (sram_dout1)  // Data out for Port 1
);

// 2. CONTROL LOGIC (Inside FSM)
always @(posedge clk) begin
    if (!rst_n) begin
        sram_csb0 <= 1'b1; // Disable at reset
    end else begin
        // Example: WRITE 0xDEADBEEF to Address 5
        if (state == WRITE_STATE) begin
            sram_addr0  <= 8'd5;
            sram_din0   <= 32'hDEADBEEF;
            sram_wmask0 <= 4'b1111; // Write full word
            sram_csb0   <= 1'b0;    // Enable Port
            sram_web0   <= 1'b0;    // Write Mode
        end
        // Example: READ from Address 5
        else if (state == READ_STATE) begin
            sram_addr0 <= 8'd5;
            sram_csb0  <= 1'b0;    // Enable Port
            sram_web0  <= 1'b1;    // Read Mode
            // Note: sram_dout0 will be valid NEXT cycle
        end
    end
end
```

Handshake Protocol (Valid/Ready)

To communicate between the Data Loader and the memory (or between modules), a standard handshake is required to prevent data loss. The goal is to ensure that the receiver is ready to accept data before the sender releases it.

- **Valid (Source):** "I have valid data on the bus."
- **Ready (Dest):** "I am ready to accept data."

Rule: Data is transferred *only* on the rising edge of the clock when **BOTH** Valid and Ready are High.

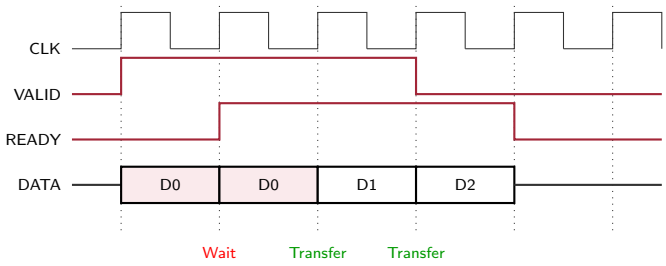

Figure 5: Handshake Timing: D0 is held until Ready goes High.

Ping-Pong Buffering

To maximize the utilization of the Systolic Array, the system must avoid "Stop-and-Go" cycles. This is achieved using **Ping-Pong Buffering** (or Double Buffering).
In this scheme, the memory space is logically divided into two banks: **Bank 0** and **Bank 1**.

- **Phase A:** The Data Loader writes to Bank 0, while the Array reads from Bank 1.
- **Phase B:** The roles are swapped.

This effectively hides the latency of memory access, allowing the array to process data continuously.

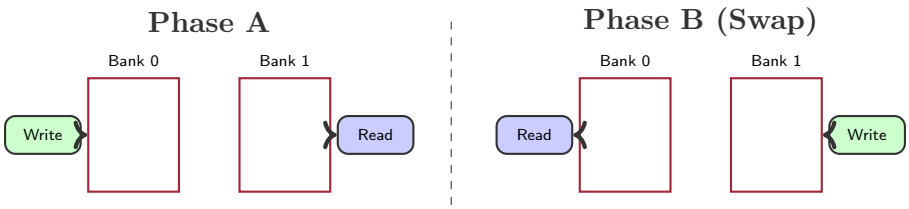

Figure 6: Logical Bank Swapping for Continuous Data Flow

Address Generation (AGU)

The input image is 2D ($N \times N$), but the memory is 1D (Linear). The **Address Generation Unit (AGU)** is responsible for mapping 2D coordinates (x, y) to a linear address: $Addr = y \times Width + x$.
Furthermore, because the Systolic Array processes data in tiles (e.g., 8×8), the AGU must generate non-sequential "Sliding Window" address patterns, rather than simple incrementing counters.

Computational Strategy

This section covers the algorithmic challenges of mapping large data onto limited hardware and the arithmetic rules for signal processing.

The Mapping Problem (Tiling)

Since the input matrix size (up to 64×64) exceeds the physical dimensions of the Systolic Array (8×8), the entire image cannot be processed at once. The input matrix must be divided into smaller **Tiles** or Blocks.
The Halo Effect: When tiling for convolution, tiles are not independent. To produce valid output pixels at the edge of a tile, input pixels from the neighboring tile (the "Halo") are typically required. The control logic must account for this overlap when loading data from DRAM.

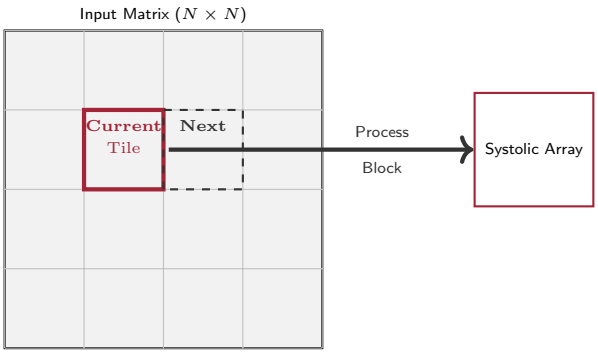

Figure 7: Tiling: Breaking a large matrix into hardware-sized blocks

Handling Halos (Overlapping Inputs): To simplify memory write-back, the control logic must be designed to always produce a fixed 8×8 valid output block. This requires the AGU to fetch an overlapping input window sized to include the necessary "halo" pixels:

$$\text{Input Size} = \text{Array Size} + (K - 1)$$

For example, with a 3×3 kernel, the AGU fetches a 10×10 input tile to produce a clean 8×8 output. **Crucially, when moving to the next tile, the AGU strides by the Output Size (8), ensuring the inputs overlap correctly.** This strategy is compatible with both Weight Stationary and Output Stationary dataflows.

Arithmetic Representation

The accelerator operates using **Fixed Point Arithmetic**, specifically treating data as 8-bit Unsigned Integers (equivalent to Q8.0 format). This representation simplifies hardware by removing the need for floating-point logic. For a primer on this concept, refer to the GeeksForGeeks Fixed Point Tutorial.

- **Inputs/Weights:** 8-bit Unsigned.
- **Accumulation:** The internal PE data path must be designed to handle **Bit Growth** to prevent overflow.
- **Output Truncation:** The final accumulated result will be wider than the 8-bit output bus. Logic must be implemented to convert this wide result back to an 8-bit format (e.g., via saturation or truncation) before writing it to DRAM.

IMPLEMENTATION GUIDE

The implementation of the accelerator is divided into sequential stages to ensure systematic development and verification. Adhering to this order minimizes integration complexity.

Stage 1: The Compute Core

- **PE Design:** Implementation of a single Processing Element (PE) capable of performing the Multiply-Accumulate (MAC) operation with the required internal precision.
- **Unit Verification:** Development of a testbench for a single PE to verify that the psum_out accumulates input streams correctly.
- **Array Assembly:** Instantiation of PEs in an 8×8 grid structure. Verification must confirm that data flows correctly across the array boundaries (West to East, North to South).

Stage 2: Memory Integration

- **Macro Instantiation:** Integration of the SRAM Hard Macro (selected via Method A, B, or C from Section 3.5) into the design.
- **Read/Write Verification:** Creation of a testbench to validate write and read operations, specifically confirming the 1-cycle read latency of the IP.
- **Ping-Pong Logic:** Implementation of the bank swapping logic. Tests must demonstrate the ability to write to Bank 0 and read from Bank 1 simultaneously without data corruption.

Stage 3: Control & Address Generation

As shown in the System Architecture diagram, this stage requires the development of two distinct but interacting blocks:

1. **The Control Unit (Main FSM):** This module acts as the system orchestrator. It manages the global states (e.g., IDLE, LOAD, COMPUTE, DRAIN). It is responsible for latching the configuration inputs (cfg_N, cfg_K), handling the handshake with the Host, and triggering the AGU.
2. **The Data Loader & AGU:** This module handles the complex address arithmetic. It receives "Go" signals from the Control Unit and generates the specific Read/Write address sequences required for:
   - **Loading:** Linear writing of incoming data into the SRAM buffers.
   - **Streaming:** Generating the "Sliding Window" read patterns to feed the Systolic Array.
   - **Unloading:** Reading results from the array output and sending them to the external bus.

Stage 4: System Verification (Golden Model)

Functional correctness is established by comparing the hardware outputs against a software reference model.

1. **Python Reference Script:** A Python script is required to perform 2D convolution on a random 64×64 matrix. The input matrix, kernel, and expected output matrix must be exported to text files.
2. **Verilog Testbench:** A self-checking testbench must be created to:
   - Read the input text files into the simulated DRAM.
   - Execute the accelerator processing.
   - Write the accelerator's output to a results.txt file.
3. **Automated Comparison:** The generated results.txt is compared against the expected output from Python.
4. **Precision Tolerance:** For fixed-point comparisons, a tolerance of ± 1 decimal point is permitted to account for rounding differences in the truncation logic.

Stage 5: Optimization & Physical Design

Following functional verification, the design is optimized for **Clock Frequency**, **Total Area**, **Power Consumption**, and **Latency**. Designs will be evaluated based on these metrics.

- **RTL Optimizations:**
  - **Area:** Reduction of counter bit-widths and resource sharing between PEs where applicable.
  - **Power:** Implementation of Clock Gating for PEs that are idle during halo loading or array filling.
- **OpenLane Configuration:**
  - **Timing:** Adjustment of CLOCK_PERIOD to identify the maximum operating frequency.
  - **Utilization:** Tuning of FP_CORE_UTIL to minimize the final die area.
  - **Metrics Analysis:** Review of the metrics.csv report after every run to track improvements in PPA (Power, Performance, Area).

DELIVERABLES & EVALUATION

Submission Requirements

The team must submit a single .zip file to Google Classroom containing the following directory structure. Please ensure your repository is clean and does not contain unnecessary temporary build files.

- **rtl/**: All Source Verilog files (Accelerator, PE, Memory Wrapper).
- **scripts/**: Any auxiliary Python or Shell scripts developed for testing, testbench generation, or result analysis.
- **config/**: The config.json files used for OpenLane synthesis.
- **final/**: The specific final output directory from your best OpenLane run (containing the merged GDSII, LEF, and final Reports). **Do not** submit the entire runs folder or temporary build steps.
- **report.pdf**: A comprehensive project document that reflects the entire lifecycle of the project, detailing the development journey and integration steps.

  Recommendation: Teams are strongly encouraged to start this document on the first day of development and maintain it collaboratively using tools like GitHub, Office 365, or Google Docs to accurately capture the design evolution.

  The report must include:
  - **Development Log:** A summary of how the project came to life, including integration steps and major design decisions.
  - **Strategy:** State Diagram of the Control Unit and Explanation of your Tiling and Address Generation logic.

- **Results:** The Optimization Table comparing your Baseline vs. Optimized metrics.
- **Workload Division:** A clear breakdown of the specific tasks assigned to each team member and their contributions.

Logistics & Deadlines

Team & Submission Rules

- **Team Size:** 7 to 8 members.
- **Submission Platform:** Google Classroom (One submission per team).
- **Deadline:** Week 13 (Unless officially changed by the Faculty).

Evaluation Criteria (Total: 20 Marks)

The evaluation is split into Functional Correctness, Optimization Performance, and Individual Contribution.

1. Functional Verification (10 Marks)

- **Golden Model Matching:** The output of your Verilog module must match the Python Golden Model reference with a precision tolerance of ± 0.1.
- **Test Coverage:** The design must handle varying Matrix sizes (N) and Kernel sizes (K) as specified in the constraints.

2. Performance Optimization (5 Marks)

Your design will be ranked against the rest of the class based on the **PPA Metrics** (Power, Performance, Area). Grades are assigned based on your performance quartile:

- **Top 25%:** Full Marks.
- **Middle 50%:** Scaled Marks (25% - 75%).
- **Bottom 25%:** Penalized.
- **Note:** Significant outliers (exceptionally good or bad) may receive extra awards or penalties.

3. Personal Evaluation (5 Marks)

During the discussion, every team member must speak for **under one minute** to concisely explain their specific contribution to the project.

WARNING: Contribution Requirement

Failure to demonstrate an effective contribution to the project may result in a **deduction of 50% of the total project marks** (loss of 10 Marks), regardless of the group's technical success.

Bonus: Creativity (Up to 20%)

Extra marks may be awarded for unique architectural innovations, such as:

- Implementing advanced dataflows (e.g., Input Stationary vs Weight Stationary analysis).
- Clever use of memory banking to reduce stall cycles.
- Developing a sophisticated automated regression testing suite.

RECOMMENDED RESOURCES

Review these resources to understand the core concepts required for the project.

Articles

- **Systolic Arrays (PDF):** Matrix Multiplication using Systolic Arrays – Prof. Shaaban (Portland State Univ).
- **TPU Architecture:** Google Cloud: What makes TPUs fine-tuned for Deep Learning? – Kaz Sato (Google Cloud AI).
- **Ping-Pong Buffering:** StackExchange: Creating Ping Pong Buffer using Dual Port RAM – Community Discussion.
- **Fixed Point Math:** GeeksForGeeks: Fixed Point Representation.
- **Number Formats:** Electronic Design: Fixed vs Floating Point – William G. Wong.

Repositories

- **Standard Macros:** VLSIDA Sky130 SRAM Macros (Pre-Hardened Standard Sizes) – Matthew R. Guthaus (UC Santa Cruz).
- **Custom Generator:** OpenRAM Memory Generator (Python Script for Custom Sizes) – Emilio Baungarten (Cinvestav).
- **Ping-Pong FIFO:** FPGA Implementation of Ping Pong FIFO – Dave McCoy (MIT).

Videos

- **Systolic Arrays:** Matrix Multiplication on Systolic Arrays – Mahmood Naderan-Tahan (NJIT).
- **Buffering Concept:** Introduction to Ping Pong Buffers – Eli Hughes (Wavenumber LLC).
- **Memory Hierarchy:** Memory Hierarchy Explanation – Prof. Nitin Chandrachoodan (IIT Madras).