Design Document

1. Objectives & Scope

System Objective

The system aims to implement a **dynamic hand gesture recognition accelerator** using a **custom tiny-CNN backbone**, optimized through **systolic array-based acceleration**. The focus is on leveraging layer fusion and hardware-aware optimization techniques to improve the performance and efficiency of convolutional layers.

Key Requirements

- **Systolic array acceleration** must be applied specifically to convolutional layers **after layer fusion**, enabling high-throughput and parallel data movement during convolution operations.

- The design should maintain compatibility with the custom tiny-CNN architecture and support dynamic gesture recognition tasks with reasonable accuracy and latency.

Constraints

- **Limited On-Chip Memory (BRAM):**
  The intermediate parameters of CNN layers are too large to fit entirely within on-chip BRAM, leading to storage bottlenecks.

- **DSP Resource Limitation:**
  The available DSP blocks on the FPGA are limited, restricting the level of parallel computation that can be sustained.

Proposed Solutions

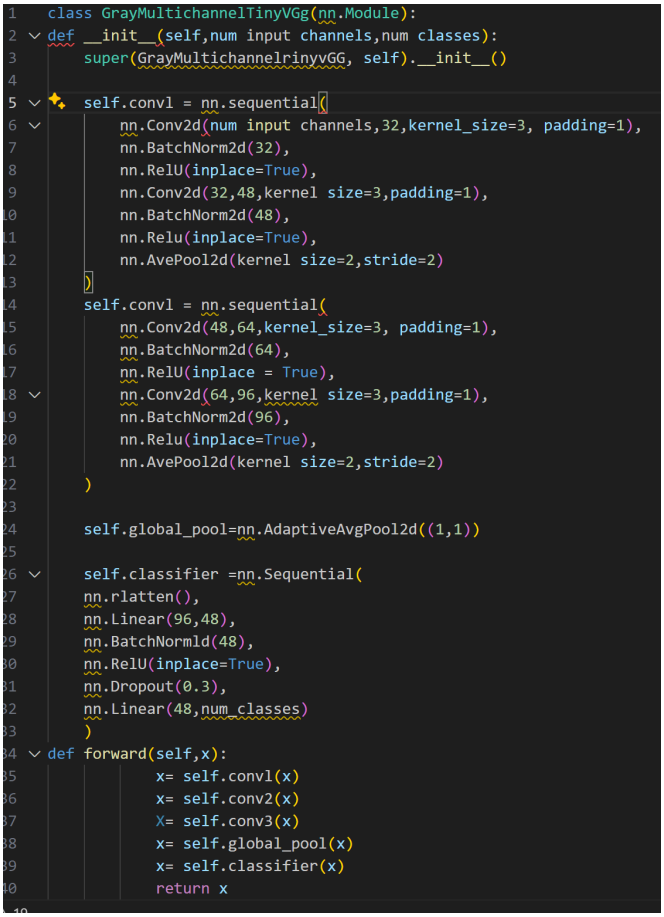
To address the above constraints, the design incorporates:

- **Time-Multiplexed Execution:** Only one convolutional layer is processed at a time, reducing simultaneous memory and compute demands.

- **Buffer-Based BRAM Utilization:** A buffer structure is used to temporarily store partial data in BRAM, facilitating efficient reuse and minimizing memory footprint.

- **Regular DRAM Interactions:** Most feature maps and weights are stored in off-chip DRAM. The buffer synchronizes with DRAM at controlled intervals to fetch/store data, thus balancing memory bandwidth and compute throughput.

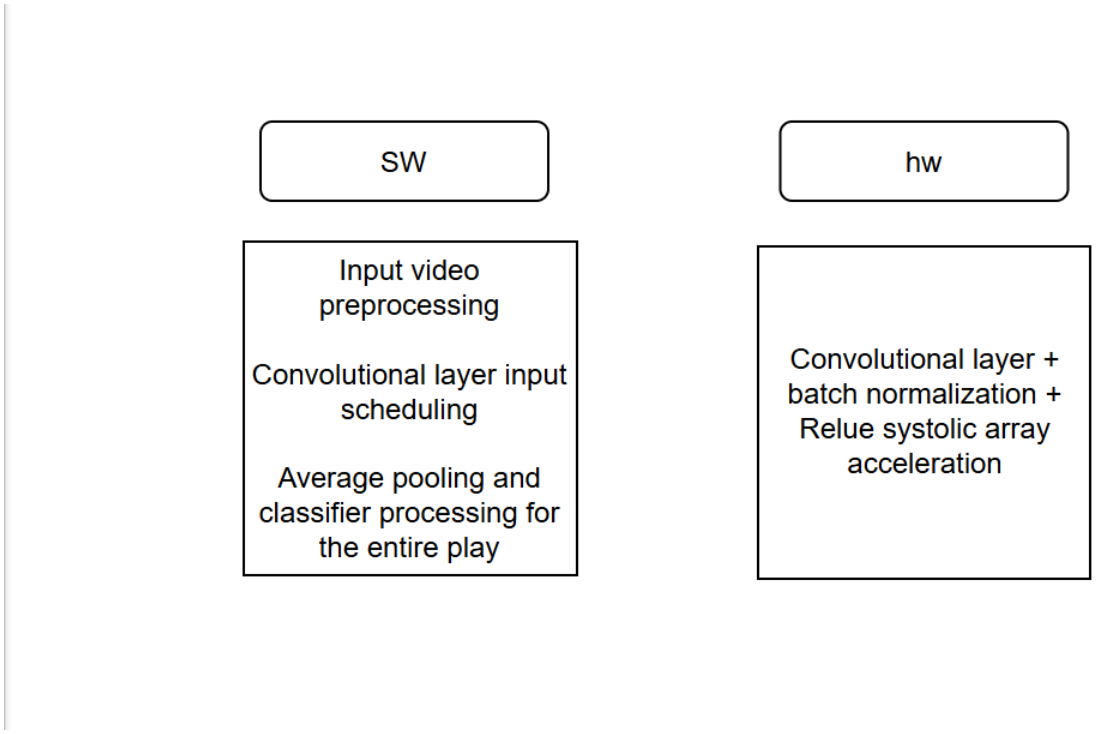
Model Architecture & Software Design



```
class GrayMultichannelTinyVGg(nn.Module):
def __init__(self,num input channels,num classes):
    super(GrayMultichannelrinyvGG, self).__init__()

    self.convl = nn.sequential(
        nn.Conv2d(num input channels,32,kernel_size=3, padding=1),
        nn.BatchNorm2d(32),
        nn.RelU(inplace=True),
        nn.Conv2d(32,48,kernel size=3,padding=1),
        nn.BatchNorm2d(48),
        nn.Relu(inplace=True),
        nn.AvePool2d(kernel size=2,stride=2)
    )
    self.convl = nn.sequential(
        nn.Conv2d(48,64,kernel_size=3, padding=1),
        nn.BatchNorm2d(64),
        nn.RelU(inplace = True),
        nn.Conv2d(64,96,kernel size=3,padding=1),
        nn.BatchNorm2d(96),
        nn.Relu(inplace=True),
        nn.AvePool2d(kernel size=2,stride=2)
    )

    self.global_pool=nn.AdaptiveAvgPool2d((1,1))

    self.classifier =nn.Sequential(
    nn.rlatten(),
    nn.Linear(96,48),
    nn.BatchNormld(48),
    nn.RelU(inplace=True),
    nn.Dropout(0.3),
    nn.Linear(48,num_classes)
    )
def forward(self,x):
            x= self.convl(x)
            x= self.conv2(x)
            X= self.conv3(x)
            x= self.global_pool(x)
            x= self.classifier(x)
            return x
```

2. **Top-Level HW/SW Block Diagram & Roles**

   – Single diagram showing HW vs. SW partition



– Call out each HW block and exactly what the SW will do

a. Block Name & Function & I/O & Handshake Detail

1. **PE (Processing Element)**
   - **Block Name**: PE
   - **Function**: Basic computation unit that performs multiply-accumulate (MAC) operations for convolutional neural networks
   - **Input/Output**:
     - **Inputs**: CLK (clock), RESET (active low reset), EN (enable signal), SELECTOR (weight selection signal), W_EN (weight enable), active_left (input activation value), in_sum (input partial sum), in_weight_above (weight input from above)
     - **Outputs**: active_right (activation output to right), out_sum (output partial sum), out_weight_below (weight output to below)
   - **Handshake Details**: Enabled by EN signal, weight flow controlled by W_EN signal, SELECTOR signal chooses which weight set to use
2. **PE_row**
   - **Block Name**: PE_row
   - **Function**: Creates a row of processing elements that process data in a systolic manner
   - **Input/Output**:
     - **Inputs**: CLK, RESET, EN, W_EN, SELECTOR, active_left, in_weight_above, in_sum

        - **Outputs**: out_weight_below, out_sum, (optional debug output: out_active_right)
    - **Handshake Details**: Activations pass from left to right, weights pass from top to bottom, partial sums accumulate from left to right
3. **PE_array**
    - **Block Name**: PE_array
    - **Function**: Creates a 9×8 2D array of processing elements for parallel CNN computation
    - **Input/Output**:
        - **Inputs**: CLK, RESET, EN, SELECTOR, W_EN, active_left, in_weight_above
        - **Outputs**: out_weight_final, out_sum_final, (optional debug output: active_right)
    - **Handshake Details**: Coordinates operation of multiple PE rows, implementing systolic array computation pattern
4. **dffr**
    - **Block Name**: dffr
    - **Function**: D flip-flop with asynchronous reset, used as register elements throughout the design
    - **Input/Output**:
        - **Inputs**: clk, rst_n (active low reset), d (data input)
        - **Outputs**: q (data output)
    - **Handshake Details**: Transfers input data to output on rising clock edge, controlled by asynchronous reset signal
5. **weight_buffer_ping/weight_buffer_pong**
    - **Block Name**: weight_buffer_ping/weight_buffer_pong
    - **Function**: Ping-pong design for weight buffers, allowing simultaneous read/write operations
    - **Input/Output**:
        - **Inputs**: clk, rst_n, wen (write enable), ren (read enable), waddr (write address), raddr (read address), din (data input)
        - **Outputs**: dout (data output)
    - **Handshake Details**: Read/write operations controlled by wen and ren signals, two buffers work alternately
6. **input_buffer_ping/input_buffer_pong**
    - **Block Name**: input_buffer_ping/input_buffer_pong
    - **Function**: Ping-pong buffers for input feature maps, enabling overlapped computation and data loading
    - **Input/Output**:
        - **Inputs**: clk, rst_n, wen, ren, waddr, raddr0-8, din
        - **Outputs**: dout0-8 (9 independent outputs)
    - **Handshake Details**: One buffer writes new data while the other provides data for current computation
7. **Output_Buffer**

- **Block Name**: Output_Buffer
- **Function**: Accumulates and stores output feature maps
- **Input/Output**:
  - **Inputs**: clk, ren, wen, A (write address), r_A (read address), D (128-bit data input), rst_n
  - **Outputs**: Q (64-bit data output)
- **Handshake Details**: Read/write operations controlled by wen and ren signals, supports accumulation operations

8. **address_calc**
   - **Block Name**: address_calc
   - **Function**: Calculates memory addresses for sliding window pattern access
   - **Input/Output**:
     - **Inputs**: clk, rst_n, en
     - **Outputs**: addr0-8 (nine address outputs), done (completion signal)
   - **Handshake Details**: Enabled by en signal, indicates completion through done signal
9. **Read_top**
   - **Block Name**: Read_top
   - **Function**: Top-level module for data reading that manages ping-pong input buffers
   - **Input/Output**:
     - **Inputs**: clk, rst_n, datain, ready, select
     - **Outputs**: out0-8, done
   - **Handshake Details**: Ready signal controls reading process, select signal chooses which buffer to use, done signal indicates completion
10. **Conv2D_control**
    - **Block Name**: Conv2D_control
    - **Function**: Main control module for the accelerator, orchestrating data flow and computation
    - **Input/Output**:
      - **System**: clk, rst_n, start, input_buffer_put_done
      - **DRAM**: Various DRAM control signals for weight, input, and output
      - **Buffers**: Control signals for weight, input, and output buffers
      - **PE array**: Control signals (pe_en, pe_selector, pe_w_en)
      - **Output**: conv_done (convolution complete signal)
    - **Handshake Details**: State machine controls system operation flow, starts via start signal, indicates completion via conv_done, uses different handshake signals for submodules based on current state

b. Control/status registers exposed to software

   input clk

   input rst_n

input start

input Dram_ready

localparam INPUT_CH   = 7'd32;

localparam OUTPUT_CH   = 8'd64;

localparam KERNEL_SIZE = 3'd3;

localparam PE_ROWS   = 4'd9;

localparam PE_COLS   = 4'd8;

localparam MAX_CYCLES   = 4'd16;

localparam IMG_WIDTH   = 8'd128;

localparam IMG_HEIGHT   = 8'd128;

localparam OUT_WIDTH   = 8'd126;

localparam OUT_HEIGHT   = 8'd126;

localparam WEIGHT_BASE_ADDR ;

localparam INPUT_BASE_ADDR;

localparam OUTPUT_BASE_ADDR;

output:cov_done

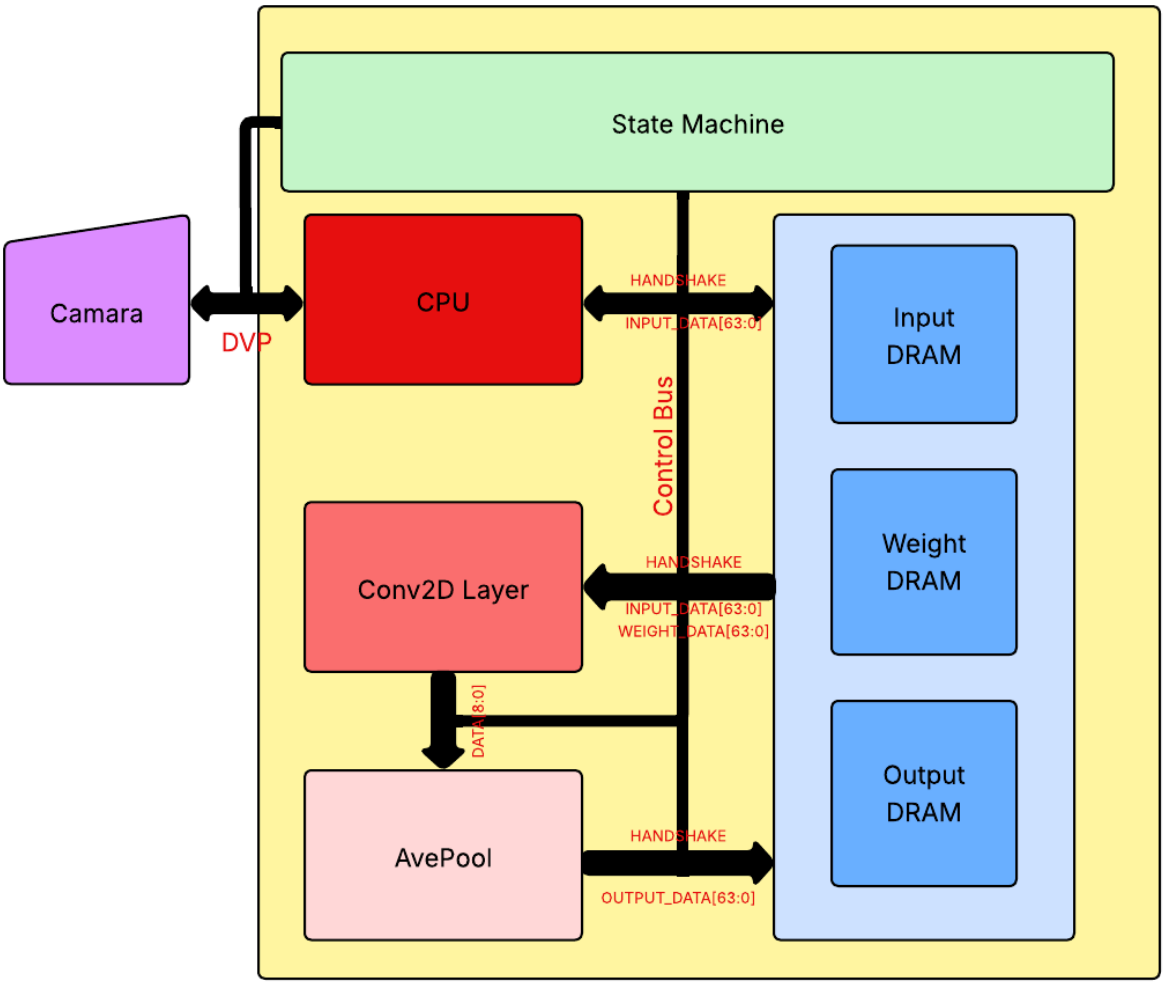


Figure 2: Block Diagram

.

3. **Dataflow & Algorithm Overview**
   – High-level pseudocode and flowchart of core processing steps

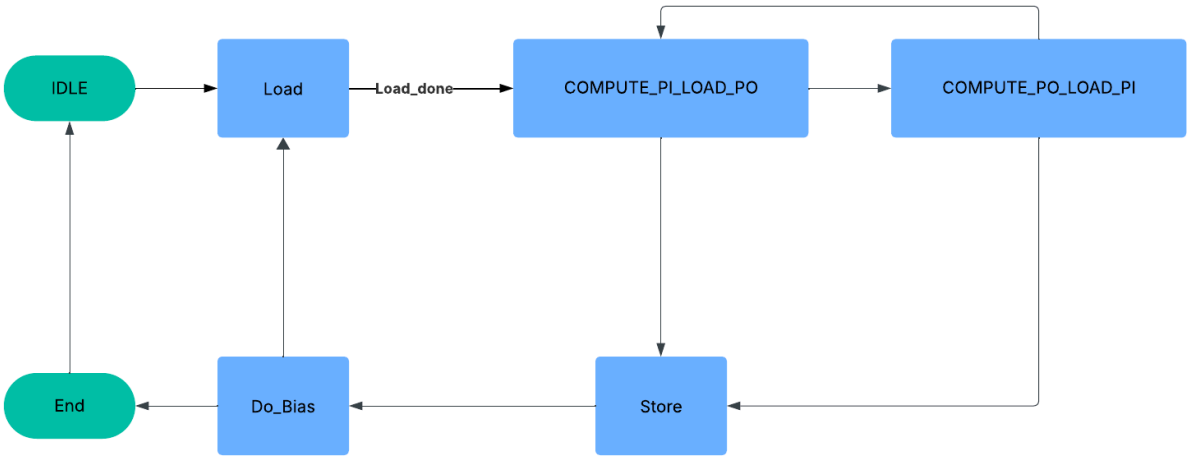


Figure 3: Flowchart

4. **FPGA Mapping & Resource Plan**（**resource budgets, register maps**）

Mapping Strategy

To maximize resource efficiency and overcome on-chip limitations, the design employs a time-multiplexed execution strategy. Each convolutional layer is executed sequentially using a shared `Conv2D` hardware module. This approach allows the reuse of limited compute and memory resources across layers, simplifying scheduling and control logic while minimizing area.

– Estimated LUT/BRAM/DSP usage

Ressource Type	Usage Estimate	Description
DSP	8 × 9	72 Processing Elements
BRAM	~280 KB Total	Detailed breakdown below

LUT	TBD	Dependent on controller, FSM, and Interfacing logic

Table 1

Buffer Type	Dimensions	Estimated Size
Input Buffer	128 × 128 × 2 (int8)	32KB
Weight Buffer	3 × 3 × 8 × 2 (int8)	144 B
Output Buffer	126 × 126 × 8(int8)	124 KB
Bias Buffer	126 × 126 × 8(int8)	124 KB

Table 2

– Critical-path outlook and clock-frequency fallback plan

Our design currently does not show signs of timing violations on any critical paths based on preliminary synthesis and timing analysis. However, should timing issues arise—particularly in the worst-case path delays—we have a fallback strategy: reducing the clock frequency to meet timing requirements.

This trade-off is acceptable for our application, as performance is not strictly constrained by clock speed. The primary goal is correctness and functional behavior; thus, operating at a lower frequency in exchange for timing closure is an acceptable compromise. Furthermore, the design is modular and parameterizable, making such frequency adjustments relatively low-cost.

5. **Verification & Schedule**（**have a working prototype in some language as evidence that it will work.**）

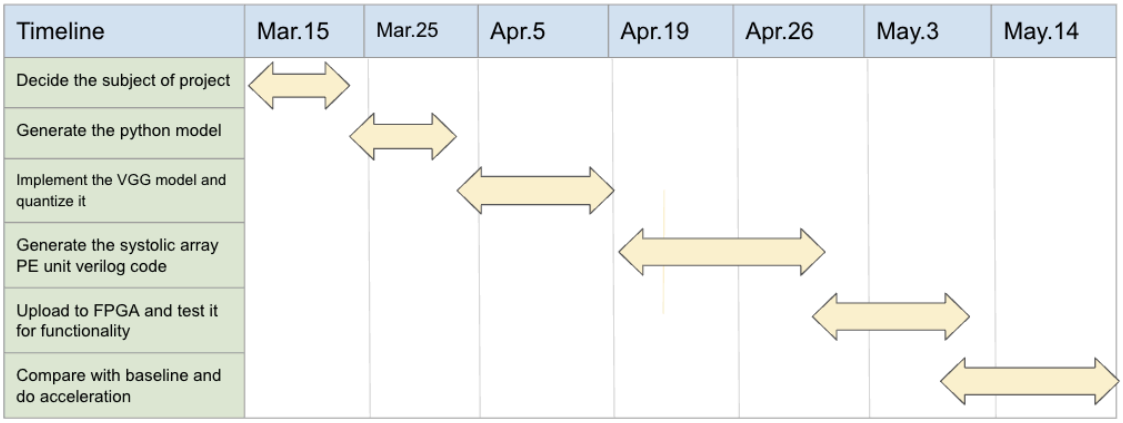
Test & Verification Plan

The following multi-stage verification strategy will be used:

- **Unit Testing**: Each hardware module (PE, buffer, address generator, control logic, etc.) will be independently simulated in Verilog testbenches to verify functional correctness and boundary conditions.
- **Integration Testing**: Subsystems like the PE array and buffer controller will be integrated and tested for dataflow correctness using simulated input and expected output feature maps.

- **Hardware-in-the-loop Testing**: After synthesis and place-and-route, the design will be deployed on an FPGA board. Input test vectors and golden outputs from the PyTorch model will be used for on-board validation.
- **Performance Profiling:** Measure latency, throughput, and utilization, and compare against target metrics.
- **Application-Level Testing**: Real-world hand gesture sequences will be used to evaluate the end-to-end accelerator performance. Input frames will be passed through the CNN layers executed on the hardware accelerator, and classification results will be compared against the software baseline.

Implementation milestones



- **Decide the Subject of the Project (March 15 – March 25)**
  - Define the scope and goals of the project
  - Choose the target model, platform, and application scenario
- **Generate the Python Model (March 25 – April 5)**
  - Build basic software components
  - Implement data preprocessing, model structure, and utility scripts
- **Implement the VGG Model and Quantize It (April 5 – April 19)**
  - Construct the VGG neural network
  - Apply model quantization techniques for hardware optimization
- **Generate the Systolic Array PE Unit Verilog Code (April 19 – April 26)**
  - Design and implement the processing element (PE) logic
  - Ensure synthesizability and correctness of the hardware description
- **Upload to FPGA and Test for Functionality (April 26 – May 3)**
  - Deploy the design onto the FPGA board
  - Conduct initial functional verification
- **Compare with Baseline and Perform Acceleration (May 3 – May 14)**
  - Benchmark against the original baseline model
  - Evaluate acceleration results in terms of performance, power, and area.