Design Document

CNN Accelerator on DE1-SoC

Chengwen Chu (cc5397)
Chengcheng Xu (cx2355)
Harvey Lu (hl3999)
Linxiao Wu (lw3227)
Mingyuan Zheng (mz3143)

April 2026

Contents

1 Introduction **2**

2 Algorithm **2**

2.1 Algorithm: CNN development and training 2

3 System Block Diagram **5**

3.1 Software Blocks 5

3.2 Hardware Blocks 6

3.3 Quantization internal computation 7

3.4 Pooling pipeline 7

3.5 Memory Resource Allocation 8

3.6 Communication Pathways 11

4 Resource Utilization **12**

5 The Hardware/Software Interface **12**

5.1 Register Map 13

5.2 Register Fields 13

5.3 Scratchpad Address Map 14

5.4 Transaction Sequence 15

1 Introduction

Field-programmable gate arrays (FPGAs) offer a compelling platform for neural network inference: they provide reconfigurable datapath logic, deterministic latency, and significantly lower power consumption than general-purpose processors, while avoiding the non-recurring engineering cost of custom silicon. However, mapping a trained deep-learning model onto an FPGA requires careful co-design of the network architecture, numerical precision, and hardware datapath to stay within the device's logic, memory, and DSP budget.

This project presents a three-layer integer-quantized convolutional neural network (CNN) accelerator implemented on the DE1-SoC platform. The accelerator recognizes hand gestures corresponding to the digits 0–9 from 64×64 grayscale images, performing end-to-end inference entirely in INT8 arithmetic. The network achieves 96.77% top-1 validation accuracy, and post-training quantization introduces zero accuracy degradation relative to the floating-point baseline.

The system is partitioned into two domains: an *offline software toolchain* that trains the model, applies full-integer post-training quantization, and packages the parameters into binary load streams; and a *synthesizable RTL pipeline* that receives those streams through a 32-bit streaming port, stores parameters in on-chip SRAM, and executes a pipelined convolution→quantization→pooling datapath followed by a fully-connected classifier and argmax reduction.

The remainder of this report is organized as follows. Section 2 describes the dataset, network architecture, training procedure, and post-training quantization. Section 3 presents the system block diagram and details each hardware and software block. Section 4 specifies the hardware/software interface, including the MMIO register map, scratchpad memory layout, and inference transaction sequence.

2 Algorithm

2.1 Algorithm: CNN development and training

Dataset

We use the open *ASL Digits 0–9* dataset published on Kaggle by Victor Anthonywhich contains RGB photographs of American Sign Language hand gestures for the digits 0–9. To match the compact input format expected by the hardware accelerator, every image is converted to single-channel grayscale and resampled to 64×64 using `cv2.INTER_AREA`, giving an input tensor of shape $(64, 64, 1)$ with pixel values normalized to $[0, 1]$ via a 1/255 rescaling.

The 2,062 processed images are partitioned per class into a 70%/15%/15% train / validation / test split, producing 1,444 training, 310 validation, and 308 test images. Class balance is preserved across all three splits (roughly 144 training and 31 validation images per digit).

Network architecture

The classifier is a deliberately small convolutional network chosen so that every layer maps cleanly onto the accelerator's MAC array and on-chip memory budget. It contains three 3×3, stride-1, `valid`-padding convolutions with ReLU activation, each followed by a 2×2 max-pooling stage, a flatten, and a single fully connected classification head (Table 1). The resulting model has only 3,810 trainable parameters, which makes it tractable for FPGA deployment.

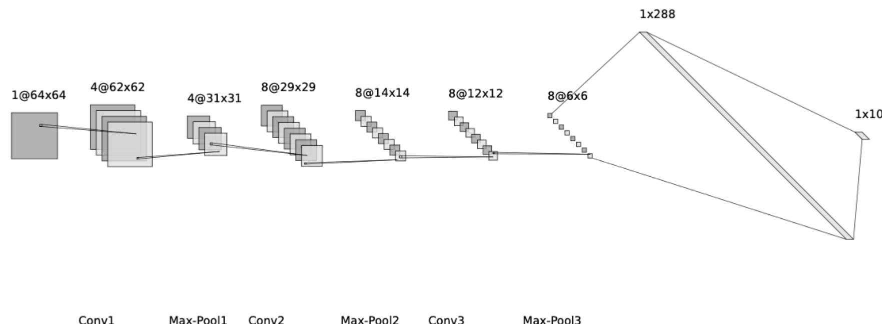

Figure 1: CNN Architecture

Table 1: CNN architecture for 64×64 grayscale digit gestures.

Layer	Kernel / Size	Output shape	Params
Input	–	$64 \times 64 \times 1$	0
Conv2D + ReLU	3×3, $C_{\text{out}} = 4$	$62 \times 62 \times 4$	40
MaxPool	2×2	$31 \times 31 \times 4$	0
Conv2D + ReLU	3×3, $C_{\text{out}} = 8$	$29 \times 29 \times 8$	296
MaxPool	2×2	$14 \times 14 \times 8$	0
Conv2D + ReLU	3×3, $C_{\text{out}} = 8$	$12 \times 12 \times 8$	584
MaxPool	2×2	$6 \times 6 \times 8$	0
Flatten	–	288	0
Dense + Softmax	$288 \rightarrow 10$	10	2,890
Total			**3,810**

Training procedure

The network is trained in TensorFlow/Keras with categorical cross-entropy loss and the Adam optimizer (default learning rate 10^{-3}). We use a batch size of 32 and train for up to 200 epochs. Each epoch iterates over $\lceil 1444/32 \rceil = 46$ training steps; validation uses all 310 images. After convergence the network reaches 96.77% top-1 accuracy on the validation split, with per-class F1 scores at or above 0.90 for every digit (Table 2).

Table 2: Per-class validation metrics for the float32 model (support = 31 per class).

Digit	Precision	Recall	F1	Support
0	0.939	1.000	0.969	31
1	0.969	1.000	0.984	31
2	1.000	0.968	0.984	31
3	0.968	0.968	0.968	31
4	0.968	0.968	0.968	31
5	1.000	1.000	1.000	31
6	0.903	0.903	0.903	31
7	0.968	0.968	0.968	31
8	0.966	0.903	0.933	31
9	1.000	1.000	1.000	31
Accuracy	**0.9677 (310 images)**			

Post-training quantization

Floating-point MAC units are costly to implement on FPGA — they require wide mantissa/exponent datapaths, normalization and rounding logic, and significantly more DSP and logic resources than their integer counterparts. To make the network hardware-friendly, every weight, activation, and accumulator in the inference graph must therefore be reduced to fixed-width integer arithmetic. We achieve this with *full-integer post-training quantization* (PTQ) to `int8` using the TensorFlow Lite converter.

After PTQ, we evaluated the int8 model on the full 310-image validation split end-to-end in `int8` gives 96.77% top-1 accuracy, matching the float32 baseline (Δacc = 0.00) with a float/int8 prediction agreement of 98.39%, confirming that the move to integer arithmetic costs essentially no accuracy for this network. The quantized weights, biases, and per-layer scales are then exported and used by the MATLAB hardware-aligned reference model and the downstream FPGA bitstream.

3 System Block Diagram

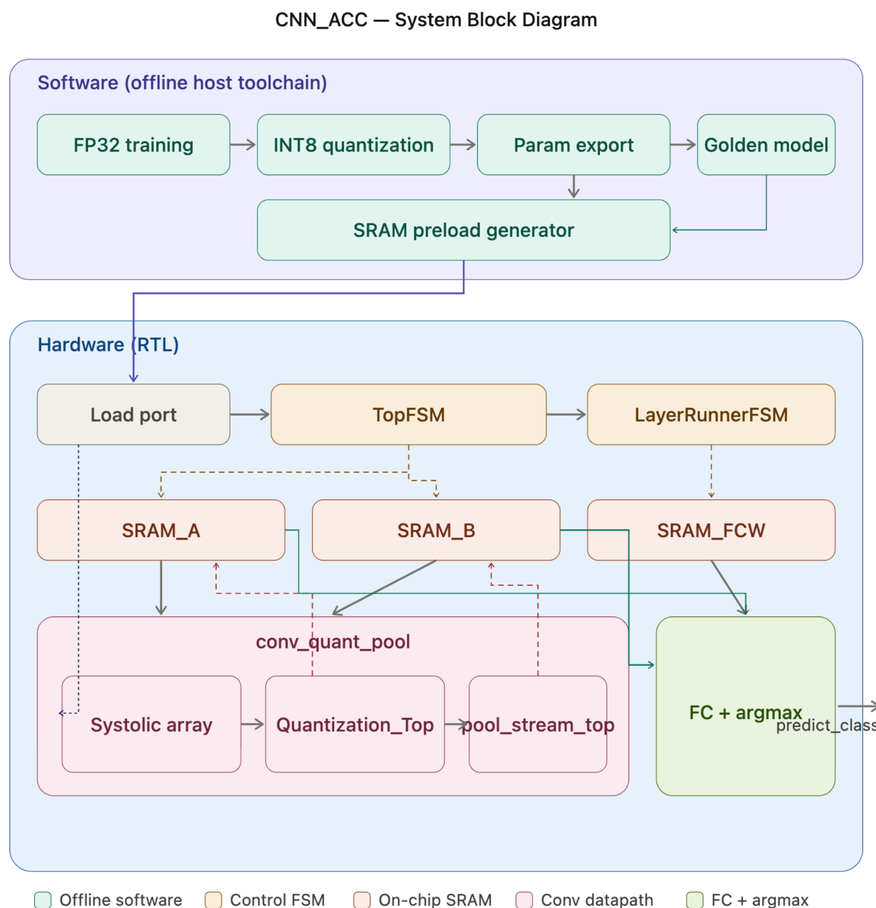

Figure 2: Block Diagram

Figure 2 shows the block diagram of our project, a three-layer INT8 CNN accelerator that performs end-to-end inference from a $64 \times 64 \times 1$ grayscale image to a 0–9 gesture class label. The system is partitioned into an *offline software toolchain* that runs on a development host (top region) and a *synthesizable RTL pipeline* that runs on the FPGA (bottom region). The two domains communicate through a single 32-bit streaming load port on the chip.

3.1 Software Blocks

The software side is a sequential offline pipeline. We first train a floating-point model, quantize it to INT8, export the quantized parameters, run a bit-exact golden reference, and finally pack everything into the binary streams that the hardware expects.

FP32 training. A three-layer CNN, consisting of three convolution/pooling stages followed by a fully-connected classifier, is trained in a standard deep-learning framework on a 0–9 hand-gesture dataset. The output of this stage is a floating-point model checkpoint.

INT8 quantization. The trained floating-point model is converted into a fully INT8 model through post-training quantization. Weights use per-channel symmetric quantization and activations use per-tensor asymmetric quantization. For every layer this stage also produces the requantization multiplier and right-shift value that the hardware will later apply to each accumulator before saturating it back to the INT8 range.

Parameter export. The quantized model is parsed by a set of host scripts that extract the per-channel weights, biases, scales, and zero-points, and emit them as plain numerical parameter files that downstream stages can consume.

Golden reference model. A bit-exact software re-implementation of the quantized network is run on the host to produce per-layer reference tensors. These tensors are compared against the hardware outputs during RTL simulation so that every pipeline stage can be verified independently.

SRAM preload generator. The final software stage takes the exported parameters and the input images and reformats them for the hardware. Convolution weights are reordered into the column-interleaved layout expected by the systolic array, and the fully-connected weights are packed into the wide slot layout expected by the dedicated FC weight memory. The output is a sequence of 32-bit words that forms the load stream.

Software-internal communication. The software blocks are sequential host programs; they communicate strictly through files. There is no run-time communication between them.

3.2 Hardware Blocks

Systolic Array (SA) structure

The systolic array is organized as a 2-D grid of processing elements (PEs), where each PE performs a local MAC operation on streaming activations and weights. Each PE receives input activations from one direction (from the left) and weights from an orthogonal direction (from the top), multiplies them, accumulates the result into a local register, and then forwards the activation and weight to its neighboring PEs in the next cycle.

For a convolution layer, input feature map tiles and kernel weights are scheduled into the array so that each PE computes partial sums for a subset of output channels and spatial locations over time. This dataflow reuses activations and weights efficiently across rows and columns of the array, which reduces external memory bandwidth and allows high throughput with a simple, regular hardware

$$y_{\text{float}} = s \cdot x, \quad y_{\text{int}} = \text{round}(y_{\text{float}}), \quad y_{\text{q}} = \text{clip}(y_{\text{int}}, -128, 127)$$

layout. After a fixed number of cycles corresponding to the kernel size and mapped channels, the partial sums in the PEs are ready and are forwarded to the quantization and pooling stages.

3.3 Quantization internal computation

Quantization maps high-precision MAC results to low-precision fixed-point integers to save memory and bandwidth. Inside the quantization (QNT) block, each MAC output x (23-bits) is first scaled by a learned or pre-defined factor s, then rounded and saturated into an 8-bit range. Concretely, the computation can be described as:

where yq is the int8 activation written back to memory or passed to the next layer.

In hardware, this is implemented as a shift-and-add or multiplier stage for the scaling factor, followed by a rounding unit and a saturating adder/ comparer block. The QNT unit is fully pipelined so it can accept one MAC result per cycle from the systolic array, ensuring that quantization does not become a performance bottleneck for the accelerator dataflow.

3.4 Pooling pipeline

The pooling pipeline processes the quantized feature maps to reduce spatial resolution and increase invariance. After a convolution layer is completed and its outputs are quantized, the feature map is streamed into the pooling unit, which forms small spatial windows (for example, 2×2) and computes either a max or average value over the window. For max pooling, the pipeline compares the four input values and forwards the maximum; for average pooling, it sums the four values and applies a fixed right shift to divide by the window size.

This pooling operation is also pipelined: each stage of the pipeline performs one step of the reduction (compare or add), so that a new window can enter the pipeline every cycle once it is filled. The pooled outputs are then written back to on-chip memory and reused as inputs to the next convolution layer, forming a continuous conv $\rightarrow$ quantization $\rightarrow$ pooling pipeline that sustains high throughput across the 3-layer CNN.

Load port. The only data entry point of the chip. The host streams 32-bit beats with a `valid/ready/last` handshake. A mode signal selects between model-load traffic (configuration, convolution weights, FC bias, and FC weights) and image-load traffic, which also triggers inference.

Top-level FSM. A network-level sequencer. During model-load it routes the four preload segments to their destinations in the on-chip memories. During inference it drives the fixed layer

schedule of the three convolutional/pooling stages followed by the fully-connected layer, and finally an argmax reduction over the ten output accumulators that produces the predicted class.

Layer-runner FSM. A per-layer transaction controller that loads the current layer's configuration and weights, streams data through the convolution datapath, and waits for the frame to finish. It decouples the network-level schedule from the per-layer streaming protocol and generates the control signals for the on-chip memories.

SRAM_A. A 32-bit memory that stores the convolution configuration words, the convolution weights, and the fully-connected bias vector. It also doubles as the writeback buffer for the second convolution layer, using a stride-2 interleaved address region.

SRAM_B. A 32-bit memory that holds the pooled feature maps between layers. The output of the first pooling stage and the output of the last pooling stage share the same physical memory; the later write simply overwrites the earlier data once it is no longer needed.

SRAM_FCW. A dedicated wide memory for the fully-connected weights. Each slot packs the per-output-channel weights for one flattened input position, so that all ten output channels can be fed in parallel in a single read. On the FPGA the internal storage can be mapped onto a block-memory IP.

All three memories receive control signals from the FSMs and expose streaming interfaces for data access.

3.5 Memory Resource Allocation

The memory subsystem is designed to support both model parameter storage and intermediate activation buffering while enabling efficient data reuse across layers. The design adopts three logical on-chip memories: `SRAM_A`, `SRAM_B`, and `SRAM_FCW`. Although these modules follow SRAM-style interfaces, in the FPGA implementation they are realized using Verilog register arrays rather than physical SRAM macros.

SRAM_A (parameter + partial activation storage). `SRAM_A` is a 32-bit $\times$ 1024-word memory (4 KB) primarily used to store model-related data. During model preload, the host streams configuration and weight data through the 32-bit load interface. The memory is logically partitioned into regions for convolution configuration parameters, convolution weights, and fully-connected bias values.

In addition to parameter storage, `SRAM_A` is reused as an intermediate feature-map buffer. After the second convolution layer, the pooled output ($14 \times 14 \times 8 = 1568$ bytes) is written back into

SRAM_A and later read as the input to the third layer. This reuse avoids allocating a dedicated buffer for each layer and reduces total on-chip memory usage.

SRAM_B (activation buffer). SRAM_B is also a 32-bit $\times$ 1024-word memory (4 KB), used primarily as a runtime activation buffer between layers. The output of the first pooling stage ($31 \times 31 \times 4 = 3844$ bytes) is written into SRAM_B and subsequently read as the input to the second convolution layer.

Later in the pipeline, the output of the third pooling stage ($6 \times 6 \times 8 = 288$ bytes) is also written into SRAM_B. This data is then reused as the flattened input vector for the fully-connected layer. By reusing the same physical memory across multiple stages, the design minimizes memory footprint while maintaining correct dataflow.

SRAM_FCW (fully-connected weight memory). SRAM_FCW is a dedicated wide memory for storing fully-connected weights. Each entry is 80 bits wide and contains ten INT8 weights corresponding to all output classes for a single input activation.

The host provides FC weights as 32-bit words, which are packed into 80-bit entries before being written into memory. During execution, one 80-bit word is read per cycle, allowing a single input activation to be broadcast and multiplied with all ten class weights in parallel. This design improves computational parallelism and reduces memory access overhead in the FC stage.

Streaming input and dataflow. The input image is not stored in on-chip memory prior to computation. Instead, during the first convolution layer, pixel data is streamed directly from the external load interface into the convolution datapath. This eliminates the need for a large input buffer and reduces memory bandwidth requirements.

Across layers, the feature map sizes are:

- Input: $64 \times 64 \times 1 = 4096$ bytes
- Layer 1 output: $31 \times 31 \times 4 = 3844$ bytes
- Layer 2 output: $14 \times 14 \times 8 = 1568$ bytes
- Layer 3 output: $6 \times 6 \times 8 = 288$ bytes

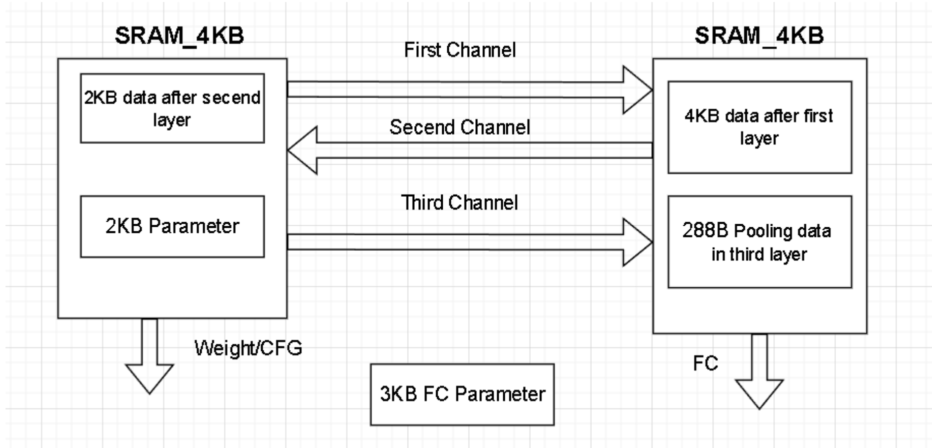

Figure 3: Memory organization and data reuse across layers

The overall dataflow follows a memory reuse pattern:

```
Input -> L1 -> SRAM_B -> L2 -> SRAM_A -> L3 -> SRAM_B -> FC
```

Design rationale. The memory architecture is optimized for three key objectives: (1) minimizing on-chip memory usage through buffer reuse, (2) reducing external memory bandwidth by streaming inputs and reusing feature maps locally, and (3) improving parallelism in the fully-connected stage through wide memory access. This balance enables efficient execution of the CNN pipeline within the limited FPGA memory resources.

Convolution datapath (conv-quant-pool). A three-stage streaming pipeline that forms the compute core of the chip:

- **Systolic array** — a multi-column array with a 3×3 sliding window. It performs several parallel INT8×INT8 multiply-accumulate operations per cycle and produces wide partial sums, one per output channel.
- **Quantization stage** — several parallel requantization units that rescale each partial sum with the per-channel multiplier, shift, and effective bias, and then saturate the result back to INT8.
- **Pooling stage** — a 2×2 stride-2 max-pooling unit that also fuses the ReLU activation, so that the full convolution → activation → pooling path is completed in a single streaming pass.

The three stages are connected internally through streaming handshakes, with no intermediate buffering in external memory.

FC and argmax. A ten-way parallel multiply-accumulate array that computes all ten class scores simultaneously, reading the broadcast pixel from the feature-map memory, the ten per-channel weights from the wide FC weight memory, and the bias vector from the configuration memory. The ten final accumulators are reduced by the top-level FSM into the 4-bit predicted class, together with a one-cycle valid pulse on the output pins.

Memory	Width	Depth	Size	Function
`SRAM_A`	32-bit	1024	4 KB	Stores convolution configuration, convolution weights, FC bias, and layer-2 output feature maps
`SRAM_B`	32-bit	1024	4 KB	Stores intermediate feature maps (layer-1 and layer-3 outputs) and serves as FC input buffer
`SRAM_FCW`	80-bit	288 (used)	2.88 KB	Stores fully-connected weights; each entry contains 10 INT8 weights for parallel computation

Table 3: On-chip memory allocation and functionality

3.6 Communication Pathways

- **Host to load port:** a 32-bit streaming handshake carrying the concatenated preload files produced by the offline toolchain.
- **Top-level FSM to layer-runner FSM:** single-cycle start pulses and level-sensitive done acknowledgments.
- **FSMs to on-chip memories:** a control bus carrying start, layer-select, data-select, pass-id, and done signals.
- **Memories to convolution datapath:** three independent streaming channels for configuration, weights, and pixels.
- **Convolution datapath back to memories:** the pooled outputs of the first and third layers are streamed back into one of the on-chip memories, while the pooled output of the second layer is streamed into the other memory with a stride-2 interleaved address pattern, so that it can be read back in the correct order as the third layer's input.
- **Memories to FC and argmax:** pixel broadcast, wide weight unpacking, and one-hot bias distribution, all synchronized by the layer-runner FSM.
- **FC and argmax to output pins:** the packed accumulator vector is reduced by combinational argmax logic in the top-level FSM and driven to the predict-class and predict-valid output pins.

- **First-layer pixel bypass:** during the first convolution layer the load stream is routed directly into the convolution datapath, bypassing the on-chip memories entirely for the raw image pixels.

4 Resource Utilization

Table 4 summarizes the current post-synthesis resource usage of the `soc_system` design on the Cyclone V `5CSEMA5F31C6`. These values are taken from the Quartus Analysis & Synthesis and Partition Merge summaries. Since the final Fitter report is not yet available, this table reports the current pre-fit utilization.

Resource	Used	Available	Utilization
Registers	25,538	–	–
Pins	73	457	16.0%
Virtual pins	0	–	0%
Block memory bits	237,568	4,065,280	5.8%
DSP blocks	86	87	98.9%
PLLs	0	6	0%
DLLs	1	4	25.0%

Table 4: Current post-synthesis resource utilization for the `soc_system` design.

The current synthesis results show that the design uses 25,538 registers, 237,568 block memory bits, and 86 DSP blocks. Among the resources with known device capacity, DSP usage is the tightest at 98.9%, while block memory usage remains modest at 5.8%.

5 The Hardware/Software Interface

The CNN accelerator is exposed to the HPS as a 16-bit memory-mapped peripheral. The HPS software stages model parameters and input data in a scratchpad memory, programs the control registers, and polls the status registers until execution completes. The MMIO wrapper then replays the staged data into the original `system_top` streaming datapath.

The interface uses two address regions:

- **Memory space** (`address[19] = 0`): 16-bit scratchpad storage
- **Configuration space** (`address[19] = 1`): 16-bit control and status registers

5.1 Register Map

All registers are 16 bits wide. Table 5 summarizes the register interface exposed by the MMIO wrapper.

Register	Index	Access	Description
CONTROL	0	WO	start model load, start inference, clear status
STATUS	1	RO	busy flag, model-loaded flag, done flag, predicted class
CONV_CFG_BASE	2	RW	base halfword address of convolution configuration segment
CONV_CFG_LEN	3	RW	number of 32-bit words in convolution configuration segment
CONV_WT_BASE	4	RW	base halfword address of convolution weight segment
CONV_WT_LEN	5	RW	number of 32-bit words in convolution weight segment
FC_BIAS_BASE	6	RW	base halfword address of FC bias segment
FC_BIAS_LEN	7	RW	number of 32-bit words in FC bias segment
FCW_BASE	8	RW	base halfword address of FC weight segment
FCW_LEN	9	RW	number of 32-bit words in FC weight segment
IMAGE_BASE	10	RW	base halfword address of input image segment
IMAGE_LEN	11	RW	number of 32-bit words in input image segment
PREDICT	12	RO	latched predicted class
IF_ERROR	13	RO	interface error flags

Table 5: MMIO register map.

5.2 Register Fields

Table 6 lists the meaning of each implemented control and status bit.

Register	Bits	Meaning
CONTROL	bit 0	start model preload replay
CONTROL	bit 1	start inference replay
CONTROL	bit 2	clear predict_done and IF_ERROR
CONTROL	bits 15:3	reserved, write zero
STATUS	bit 0	reserved
STATUS	bit 1	replay engine busy
STATUS	bit 2	model preload complete
STATUS	bit 3	inference complete
STATUS	bits 7:4	predicted class
STATUS	bits 15:8	reserved
PREDICT	bits 3:0	predicted class
PREDICT	bits 15:4	reserved
IF_ERROR	bit 0	model load requested while busy
IF_ERROR	bit 1	inference requested while busy
IF_ERROR	bit 2	inference requested before model load completed
IF_ERROR	bit 3	low halfword scratchpad read out of range
IF_ERROR	bit 4	high halfword scratchpad read out of range
IF_ERROR	bits 15:5	reserved

Table 6: Implemented control and status bit definitions.

5.3 Scratchpad Address Map

The software stores all staged data in the scratchpad memory before writing the CONTROL register. The scratchpad is 16-bit addressed, so each logical 32-bit word occupies two consecutive halfword locations. The default layout is shown in Table 7.

Segment	Base (halfwords)	Length (32-bit words)	Content
conv_cfg	0	45	convolution configuration
conv_wt	90	225	convolution weights
fc_bias	540	10	fully-connected bias
fcw	560	864	fully-connected weights
image	2288	1024	input image

Table 7: Default scratchpad layout.

The scratchpad address map is organized as follows:

conv_cfg_base	conv_cfg
conv_wt_base	conv_wt
fc_bias_base	fc_bias
fcw_base	fcw
image_base	image

Figure 4: Scratchpad memory organization.

5.4 Transaction Sequence

Each logical 32-bit word is written into the scratchpad as two consecutive 16-bit halfwords. The MMIO wrapper reconstructs the original 32-bit word before forwarding it to the accelerator input stream.

One inference transaction proceeds as follows:

1. Write preload data and input image into the scratchpad memory.
2. Program the base and length registers.
3. Write `CONTROL[0]` to start model loading.
4. Poll `STATUS[2]` until model loading completes.
5. Write `CONTROL[1]` to start inference.
6. Poll `STATUS[3]` until inference completes.
7. Read `PREDICT` and `IF_ERROR`.