FPGA-Based LeNet-5 CNN Accelerator for Handwritten Digit Recognition

Design Document for CSEE 4840 Embedded System Design

Spring 2026

Yizheng Tang, yt2992

Chenxi Shen, cs4634

Tian Li, tl3468

Weiwei Wu, ww2766

Sirui Chen, sc5746

1. Introduction

This project focuses on the design and implementation of a hardware accelerator for the LeNet-5 convolutional neural network on an embedded FPGA platform, targeting handwritten digit recognition using the MNIST dataset.

LeNet-5 is a classical CNN architecture with a well-structured and computationally efficient pipeline, making it suitable for hardware acceleration and system-level implementation.

In this project, the network parameters, including kernel weights and biases, are trained offline using Python. The trained model is then quantized and deployed onto a Verilog-based hardware accelerator that performs forward inference on the FPGA. The system integrates hardware and software components, enabling efficient execution and verification of CNN inference.

The objective is to build a complete end-to-end handwritten digit recognition system and to gain practical experience in hardware/software co-design, including model deployment, data representation, and system integration.

2. System Overview and Block Diagram

2.1 System Architecture Overview

This system architecture utilizes a hardware-software co-design on the DE1-SoC to accelerate CNN inference. The HPS (ARM Processor) handles high-level tasks, including loading int8

weights and image data from the SD card and managing the Avalon-MM Bus communication via a custom driver. The Avalon Interconnect serves as the bridge, mapping FPGA resources into the HPS memory space to facilitate control signals and data streaming. On the FPGA Hardware side, dedicated M10K BRAM blocks provide low-latency local storage for weights and input pixels, ensuring the Conv + Pool Core can perform parallelized computations without bus contention. Once the hardware engine completes the inference, results are stored in an Output Buffer, allowing the HPS to retrieve the final classification scores through the status registers.

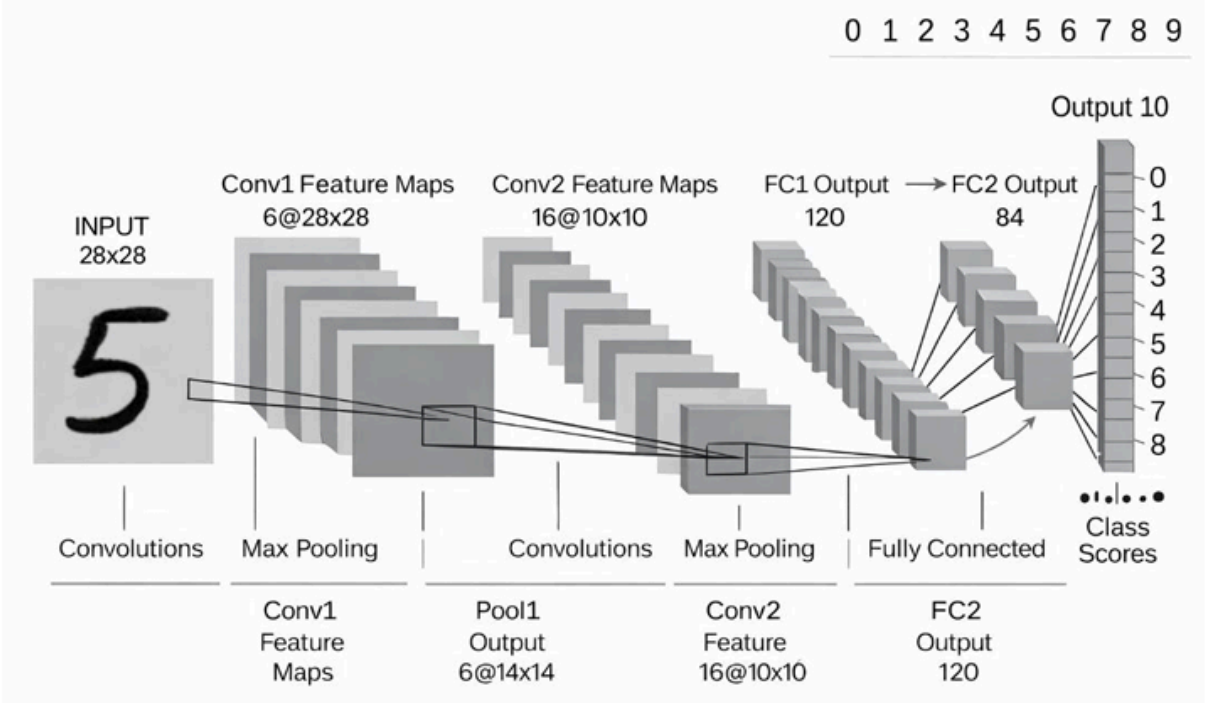


Fig 1. End-to-end LeNet-style inference dataflow

Figure 1 summarizes the complete inference flow implemented by our CNN accelerator project. On the software side, the PC preprocesses the input image and prepares data in the format expected by the hardware pipeline. On the FPGA side, convolutional and pooling layers extract spatial features, followed by fully connected layers that generate class scores. The dimensional annotations in each stage are used to derive buffer sizes, compute workloads, and determine transfer volumes. Therefore, this dataflow directly guides our RTL module decomposition, on-chip memory allocation, and interface protocol design.

2.2 System Block Diagram

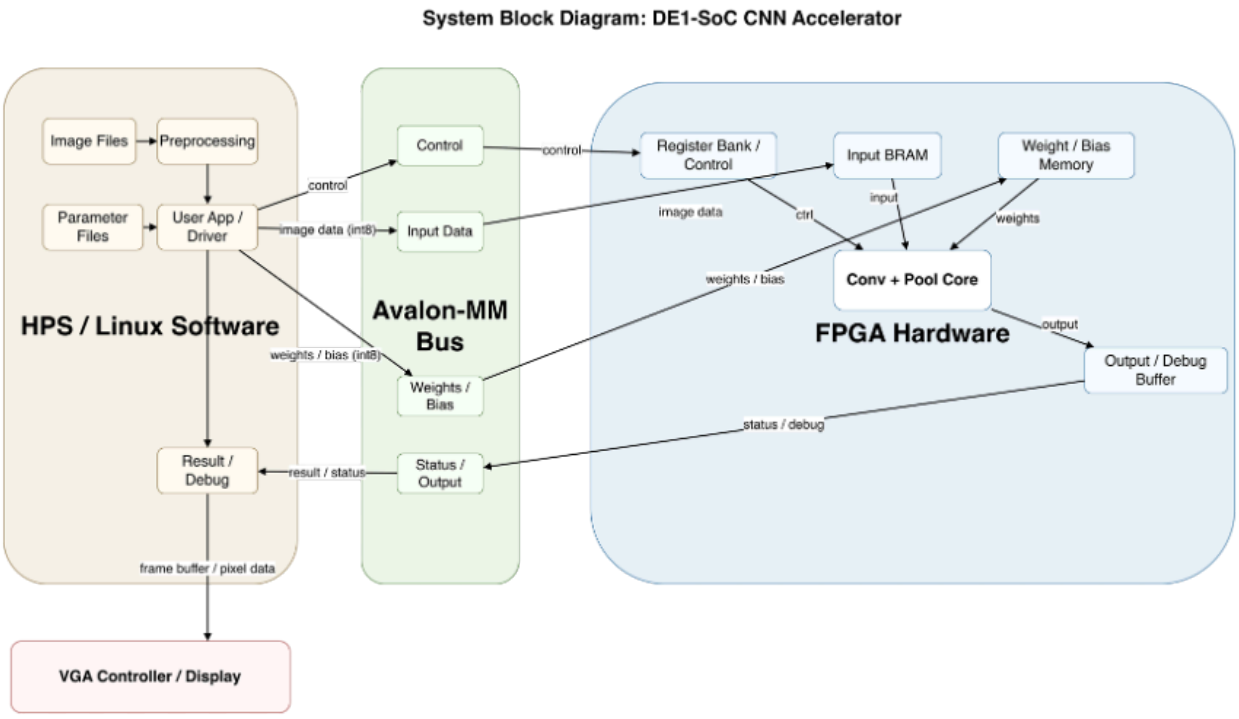


Fig 2. Block Diagram for the Project

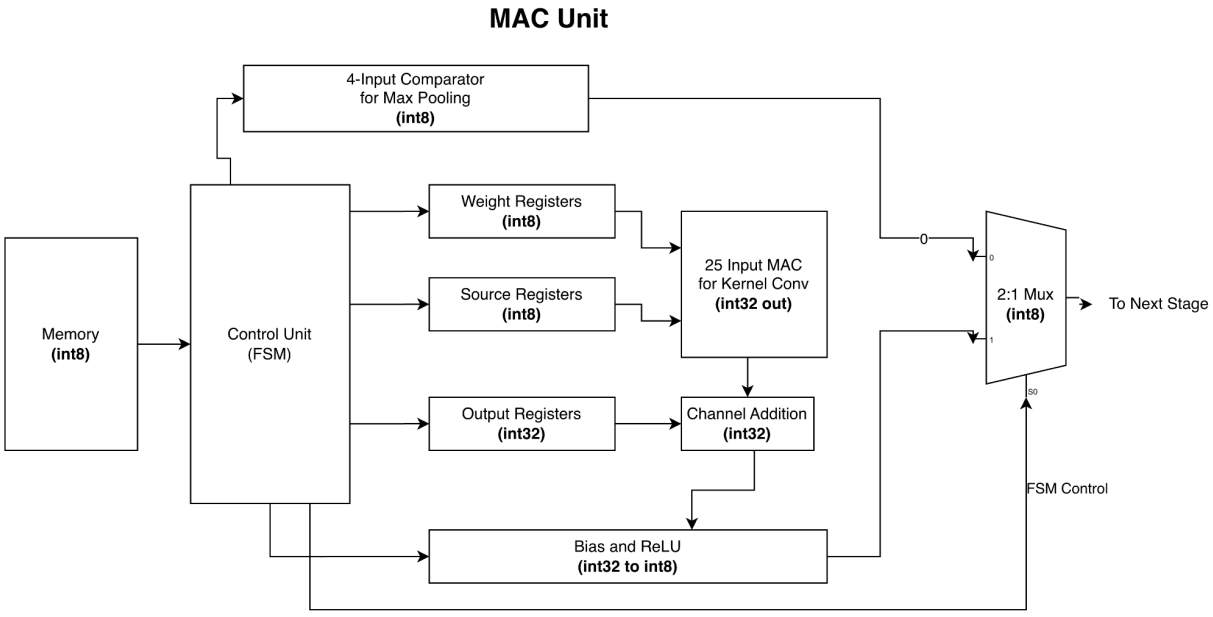


Fig 3. MAC Unit for FPGA

The MAC unit is the core computation block of the accelerator. For each 5 x 5 convolution window, it multiplies the input pixel values with the corresponding kernel weights and

accumulates the results to produce a partial convolution sum. The control FSM coordinates data loading, register updates, and operation sequencing, while the surrounding blocks support pooling comparison, channel accumulation, bias addition, ReLU activation, and output selection. This design allows the MAC unit to serve as the main arithmetic engine in the CNN hardware pipeline.

2.3 Module Description

2.3.1 Hardware Modules (FPGA)

Implemented:

- BRAM.sv: Generic on-chip memory block.
- conv1_top.sv, conv1_fsm.sv: Conv1 + bias/ReLU + pooling pipeline and control.
- conv2_top.sv, conv2_fsm.sv: Conv2 pipeline and control (under active verification refinement).
- pooling.sv, pooling_fsm.sv: Reused pooling datapath/control.
- bias_relu.sv: Bias addition and ReLU activation.
- board_top.sv, uart_rx.sv, uart_tx.sv: Board wrapper and UART debug/bring-up path.

In Progress:

- Conv2 full regression stability and writeback corner-case verification.
- Cross-layer integration checks for downstream FC-stage readiness.

Planned:

- Final top-level integration with complete LeNet-5 inference chain and runtime control interface.

2.3.2 Software Modules (HPS)

The HPS software in this project consists of two lightweight Linux user-space programs for reading FPGA results through MMIO. Their main role is not to control the accelerator, but to observe and verify the output data generated by the hardware.

Software Module	File	Main Function	Access Method	Output

Debug Register Reader	read_pool_regs.c	Reads status and captured debug values	/dev/mem + mmap() + MMIO	4 captured pooling values for each of 6 channels
Full Pool Output Reader	read_pool_full.c	Reads the complete pooled feature maps	/dev/mem + mmap() + MMIO	Full 14 x 14 output for each of 6 channels

Both programs map the lightweight HPS-to-FPGA bridge at base address 0xFF200000 and access the FPGA register space as a 32-bit word array. Since the hardware output values are stored as signed 21-bit numbers, both programs include a sign-extension function before printing the results.

In early design, the HPS does not issue a start command to the accelerator. Instead, the hardware automatically generates a one-cycle start_pulse after reset in soc_system_top.sv. Therefore, the HPS software serves mainly as a monitoring and validation interface for better debugging. And it will be updated to have a control function in the final design.

2.3.3 Communication Interfaces

Communication between HPS and FPGA is implemented through the lightweight HPS-to-FPGA AXI bridge. On the FPGA side, the custom module hps_pool_regs.sv acts as an AXI-accessible register block. On the HPS side, Linux software accesses these registers through MMIO using /dev/mem and mmap().

Interface Summary

Item	Description
Communication Type	Memory-Mapped I/O (MMIO)
Bridge Used	Lightweight HPS-to-FPGA AXI Bridge
HPS Access Method	/dev/mem + mmap()
Base Address	0xFF200000
Map Size	0x1000 bytes

Data Width	32-bit word access
Main Purpose	Read debug values and pooled CNN outputs from FPGA

MMIO Register Map

Word Offset	Address	Description	Source in RTL
0	0xFF200000	Status register, bit 0 indicates valid data	regs_valid in hps_pool_regs.sv
1 to 24	0xFF200004 onward	Debug capture registers, 6 channels x 4 samples	debug_pool_regs from conv1_top.sv
64 onward	0xFF200100 onward	Full pooled feature map data, 6 channels x 14 x 14	Pool BRAM dump path in conv1_top.sv

HPS-FPGA Communication Flow

Step	HPS Side	FPGA Side
1	Open /dev/mem	AXI slave interface is exposed by hps_pool_regs
2	Map address 0xFF200000 with mmap()	Lightweight AXI bridge forwards transactions
3	Read regs[0]	FPGA returns regs_valid status
4	Read debug register words	FPGA returns flattened debug capture values
5	Read pool output words	FPGA decodes channel/address and returns BRAM data

This design uses MMIO as a simple and effective debug interface. Instead of transferring data through a complex DMA or driver-based mechanism, the HPS can directly read hardware-generated results by accessing mapped

2.4 Data Flow Description

The system data flow initially utilized BRAM preloading (via .memh files) for standalone module verification. To process full datasets and overcome on-chip memory capacity constraints, the final integration employs the HPS to dynamically load weights and images over the Avalon-MM bus. If time permits, a DMA controller will be implemented to stream images directly from DDR RAM, maximizing acceleration efficiency by eliminating CPU overhead. Finally, inference results and performance metrics are exposed to the HPS through the MMIO interface for verification.

System Data Flow

Stage	Module / File	Function	Output
Input Initialization	conv1_top.sv + memory init files	Load image, kernels, and bias into BRAM	Input image and CNN parameters ready
Convolution	conv1_top.sv, conv1_fsm.sv, mac, bias_relu	Perform convolution, accumulation, bias addition, and ReLU	28 x 28 feature maps for 6 channels
Pooling	pooling_fsm.sv, pooling.sv	Perform 2 x 2 max pooling	14 x 14 pooled outputs for 6 channels
Debug Capture	conv1_top.sv	Store first 4 pooled values per channel into debug registers	Compact debug data
MMIO Exposure	hps_pool_regs.sv	Expose status, debug data, and pooled BRAM data to HPS	AXI-readable register space
HPS Readout	read_pool_regs.c, read_pool_full.c	Read and print results through MMIO	

In this project, the most important role of MMIO is at the final stage of the data path. It provides a bridge between FPGA computation results and HPS software observation. Through this mechanism, the HPS can verify whether the pooling output and debug values match the expected CNN behavior.

3. Algorithm Description

3.1 Model Architecture

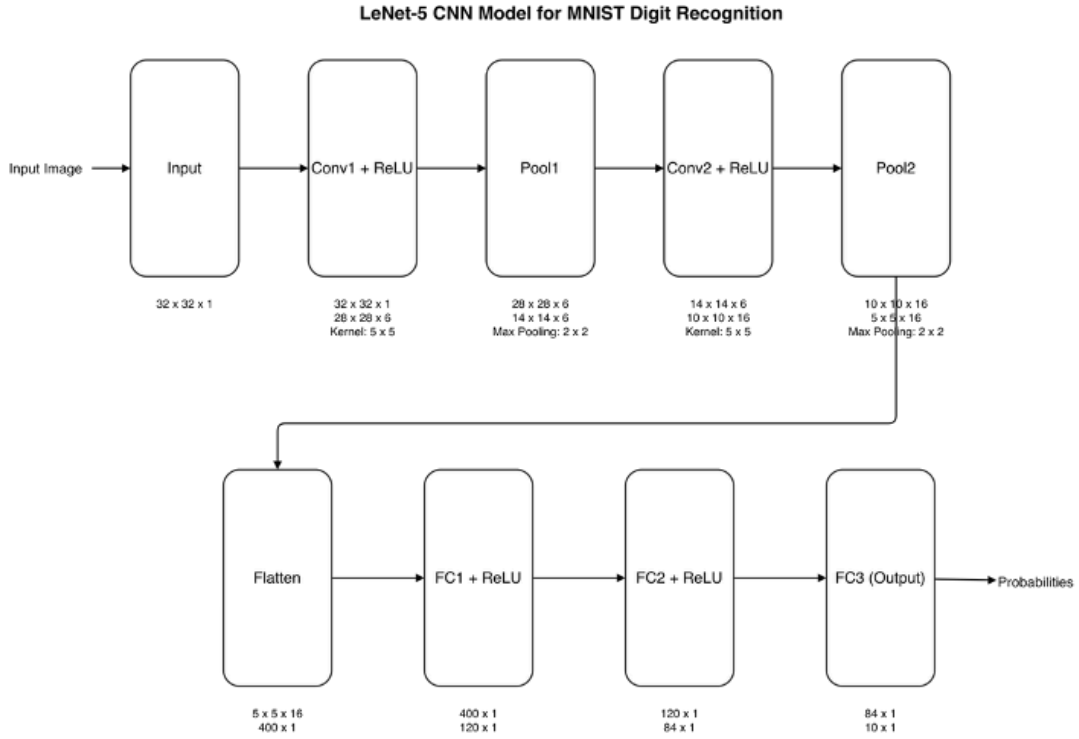


Fig 4. LetNet-5 CNN Architecture

The LeNet-5 architecture is a hierarchical model designed to extract and refine visual features for digit recognition. The process begins with Conv1 and Conv2, which use 5 x 5 kernels to identify spatial patterns. Conv1 extracts 6 basic feature maps, while Conv2 expands this to 16 maps to represent complex shapes. Each convolution is paired with a ReLU activation function to introduce non-linearity, allowing the model to learn sophisticated data mappings.

To ensure computational efficiency, the model uses Pool1 and Pool2 for 2 x 2 Max-Pooling. This halves the spatial resolution after each convolution, preserving prominent features while condensing the data. The resulting 5 x 5 16 output is then flattened into a 400-element vector, transitioning the data from spatial feature maps to a format suitable for classification.

The final Fully Connected Stage (FC1, FC2, and FC3) serves as the decision-making engine. The 400 features are compressed through 120 and 84 nodes respectively, before reaching the 10-node output layer. Each output node represents a digit from 0 to 9, with the highest activation value determining the model's final classification result.

Table 1. Verilog Module Interface Definitions

Module	File	Main Input Ports
`soc_system_top`	soc_system_top.sv	Board-level clock, reset, switches, keys, HPS DDR, USB, UART, I2C, Ethernet, an
`conv1_top`	conv1_top.sv	`clk` , `rst_n` , `start` , `dump_rd_en` , `dump_ch` , `dump_addr`
`hps_pool_regs`	hps_pool_regs.sv	`clk` , `rst_n` , `regs_valid` , `regs_flat` , `dump_data` , AXI-LW slave inputs
`conv_fsm`	conv1_fsm.sv	`clk` , `rst_n` , `start` , `mul_done`
`pooling_fsm`	pooling_fsm.sv	`clk` , `rst_n` , `conv_done`
`mac`	mac.sv	`clk` , `rst_n` , `clr` , `mul_en` , `pixel` , `kernel`
`bias_relu`	bias_relu.sv	`acc_in` , `bias_in`
`pooling`	pooling.sv	`clk` , `clr` , `pool_in` , `pool_en`
`BRAM`	BRAM.sv	`clk` , `w_en` , `w_addr` , `r_en` , `r_addr` , `w_data`

Table 2. Instantiated Submodules in Each Verilog Module

Parent Module	Instantiated Submodules
`soc_system_top`	`conv1_top dut` , `hps_pool_regs u_regs` , `soc_system_hps_0 hps_0`
`conv1_top`	`BRAM u_input_mem` ; for each channel: `BRAM u_kernel_mem` , `BRAM u_fm_mem` , `BRAM u_pool_m`
`hps_pool_regs`	None
`conv_fsm`	None
`pooling_fsm`	None
`mac`	None
`bias_relu`	None
`pooling`	None
`BRAM`	None

Table 1 defines the external interface contract of each RTL module, while Table 2 captures the internal structural composition through instantiation relationships. Together, these two views connect algorithm-level dataflow to implementation-level hierarchy. Based on this module interface and hierarchy foundation, Section 3.2 describes the layer-by-layer execution algorithm and control flow.

3.2 Algorithm for each layer

The following algorithms describe the basic operations used in the CNN: convolution, ReLU, max pooling, and fully connected computation. These operations are repeatedly applied

throughout the network to extract features, reduce spatial dimensions, and produce the final classification result.

Figure4 illustrates how a 5 x 5 kernel slides across the input feature map and generates one output value at each valid position. This helps explain why the output size becomes smaller when zero padding is used and stride is fixed at 1. After convolution, ReLU removes negative values, max pooling keeps the strongest response in each 2 x 2 region, and the fully connected layers map the extracted features to the final output scores for the ten-digit classes

```
Algorithm 1. Convolution
for each output channel do
    for each output position do
        sum = 0
        for each kernel element do
            sum = sum + input * weight
        end for
        output = sum + bias
    end for
end for
```

```
Algorithm 2. ReLU
for each element do
    if value < 0 then
        value = 0
    end if
end for
```

```
Algorithm 3. Max Pooling
for each channel do
    for each 2 x 2 window do
        output = max(window)
    end for
end for
```

```
Algorithm 4. Fully Connected Layer
for each output node do
    sum = 0
    for each input element do
        sum = sum + input * weight
    end for
    output = sum + bias
end for
```

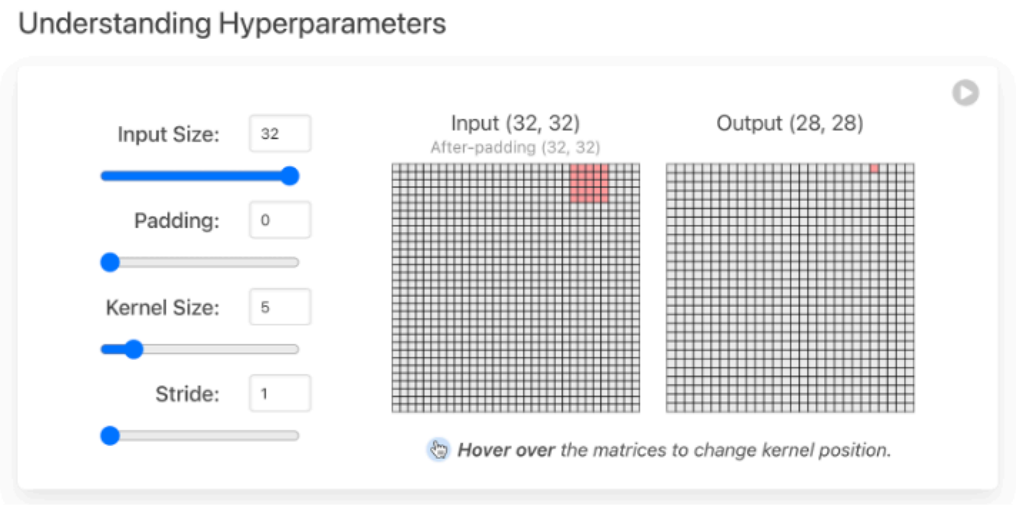


Fig 5. visualization of kernel convolution for first layer [1]

3.3 Model Pre-trained

The model parameters are pretrained offline in PyTorch using the MNIST training set, so the FPGA only performs inference and does not update weights on board. In the current software flow, the input images are resized to 32 x 32, converted to tensors, and binarized before training. The network is trained using the Adam optimizer with learning rate 0.001, batch size 64, cross-entropy loss, and 2 training epochs. Under this setup, the software reference model achieves about 98% accuracy on the MNIST test set in theory.

After training, the learned floating-point weights and biases are quantized to signed 8-bit integers and exported as .hex files for hardware deployment. This allows the trained model to be loaded into FPGA memory and used directly for inference. Since the FPGA implementation uses quantized parameters and hardware arithmetic rather than floating-point computation, the final on-board accuracy may differ slightly from the software test accuracy.

Layer Name	Input Dim	Operation	Output Dim	Weights (Stored as int8)
Input	32×32×1	Raw Pixel Data	32×32×1	0
Conv1	32×32×1	5×5 Kernel, 6 Filters	28×28×6	150 weights + 6 biases
Pool1	28×28×6	2×2 Max Pooling	14×14×6	0
Conv2	14×14×6	5×5 Kernel, 16 Filters	10×10×16	2,400 weights + 16 biases
Pool2	10×10×16	2×2 Max Pooling	5×5×16	0
Flatten	5×5×16	Unspool to 1D Vector	400×1	0
FC1	400×1	Matrix Mult (120 nodes)	120×1	48,000 weights + 120 biases
FC2	120×1	Matrix Mult (84 nodes)	84×1	10,080 weights + 84 biases
FC3 (Out)	84×1	Matrix Mult (10 nodes)	10×1	840 weights + 10 biases

Fig 6. weights for each layer

3.3 Data Representation

3.4 Hardware/Software Partitioning

4. Resource Budget

4.1 On-Chip Memory Usage

Register map:

Offset Address	Register / Block	Access	Size	Precise Hardware Behavior
0x00000000	REG_CTRL	Write	4 Bytes	Bit 0 (Start): Write 1 to launch one inference pulse.Bit 1 (Reset): Write 1 to trigger a soft reset of the FSM.
0x00000004	REG_STATUS	Read	4 Bytes	Bit 0 (Busy): Returns 1 while inference is running.Bit 1 (Done): Returns 1 when inference completes.
0x00000008	REG_RESULT	Read	4 Bytes	Returns the final classified digit (0-9).
0x0000000C	REG_CYCLES	Read	4 Bytes	Returns total hardware clock cycles taken for inference.
0x00010000	MEM_INPUT	R/W	1,024 Bytes	Stores 32x32 pixels. Stored as 1 byte per pixel (int8).
0x00020000	MEM_CONV1	R/W	156 Bytes	Stores 150 kernel weights + 6 biases (int8).
0x00030000	MEM_CONV2	R/W	2,416 Bytes	Stores 2,400 kernel weights + 16 biases (int8).
0x00040000	MEM_FC1	R/W	48,120 Bytes	Stores 48,000 dense weights + 120 biases (int8).
0x00050000	MEM_FC2	R/W	10,164 Bytes	Stores 10,080 dense weights + 84 biases (int8).
0x00060000	MEM_FC3	R/W	850 Bytes	Stores 840 dense weights + 10 biases (int8).
0x00070000	DBG_CONV1_C	Read	4,704 Bytes	(Debug) Returns 28x28x6 feature maps. Data is the int8 truncated output after ReLU.
0x00080000	DBG_FC1_OUT	Read	120 Bytes	(Debug) Returns the 120-node vector. Data is the int8 truncated output after ReLU.

- **Total Physical Memory: 67,554 Bytes (~66 KB)**

The register map serves as the primary interface between the HPS (software) and the FPGA (hardware), employing a sparse addressing strategy with 64 KB offsets. This safe-spacing strategy simplifies the hardware's internal address decoding logic and allows individual layers to scale in size during development without requiring a full remapping of subsequent memory blocks.

The first four registers act as the Command and Control centre, enabling the ARM processor to trigger inference via a self-clearing pulse, poll for completion using sticky status bits, and retrieve metadata such as the final digit classification and cycle-accurate performance benchmarks.

To maximize throughput, the system can utilize DMA to stream input images directly into the MEM_INPUT block from DDR RAM. By offloading high-volume data movement from the CPU to a dedicated hardware controller, the ARM processor is freed to handle high-level application logic or pre-process the next batch of images. This hardware-driven streaming ensures the CNN accelerator operates at peak efficiency, effectively eliminating the bottleneck where the fast calculation core sits idle waiting for the next 32 x 32 pixel graph to be loaded manually.

4.2 Computational Resources (DSP, ALU, etc.)

The computational resources of the design are mainly consumed by the convolution datapath in conv1_top.sv. The system instantiates six parallel MAC units, one for each output channel, and each MAC performs signed multiplication and accumulation. According to the Quartus compilation report, the final implementation uses 6 DSP blocks, which matches the six `mac` instances in the RTL. Therefore, the DSP resources are primarily used for the convolution operation.

Besides DSP usage, the design also uses LUT/ALM-based logic for control and non-multiplication arithmetic. Modules such as bias_relu.sv, pooling.sv, conv1_fsm.sv, and pooling_fsm.sv are implemented mainly with ALMs and registers. The complete fitted design uses 601 ALMs, 1273 registers, 14 RAM blocks, and 137,216 block memory bits on the Cyclone V device.

Resource Type	Used	Device Total	Utilization
ALMs	601	32,070	2%
Registers	1273	64,140	about 2%
Block Memory Bits	137,216	4,065,280	3%
RAM Blocks	14	397	4%
DSP Blocks	6	87	7%

4.3 Bandwidth Requirements

The bandwidth requirements of this design are moderate. While initial testing utilized BRAM preloading to minimize bus traffic , the final system employs HPS dynamic loading to overcome on-chip memory capacity limits and support a larger volume of input images. The main runtime data movement, including

streaming pixels, loading kernel weights, and reading pooled outputs, which iis managed through local buffering to ensure the hardware core is not starved for data. Consequently, the current design remains compute-oriented rather than bandwidth-limited.

4.4 Scalability Considerations

The current design is scalable, but resource usage will increase with larger CNN dimensions. In particular, increasing the number of output channels will almost linearly increase DSP usage, because each channel currently corresponds to one MAC unit in mac.sv. Similarly, larger feature maps or deeper layers will require more on-chip memory for intermediate storage and more control logic for address generation.

Another scalability consideration is bandwidth and data movement. While the current first-layer implementation works well with on-chip BRAM, larger CNN models may exceed the available internal memory and require external DDR access or a more efficient buffering strategy. In that case, the lightweight MMIO interface used for debugging would still be suitable for control and monitoring, but not for transferring large amounts of feature-map data. Overall, the present architecture is suitable for a small CNN prototype, but deeper and wider networks would require more DSP parallelism, more memory capacity, and a more efficient communication mechanism.

Scaling Factor	Main Impact
More channels	More DSP blocks
Larger feature maps	More BRAM usage
Deeper network	More control logic and storage
More HPS-FPGA data exchange	Higher bandwidth requirement

5. Hardware/Software Interface

5.1 System Interface Overview

The system employs a master-slave architecture for communication between the HPS and the accelerator.

The processor on the HPS acts as the master. It initiates all transactions, including configuring the accelerator, loading input data, starting the computation, and retrieving the results.

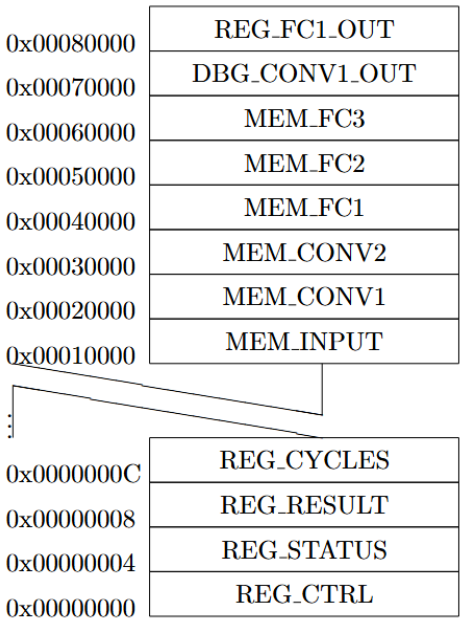
The hardware accelerator, implemented on the FPGA, acts as a slave peripheral. It responds to read and write requests from the HPS to its assigned memory-mapped address range.

The connection is physically realized through the AXI-Light bridge, which maps a portion of the HPS's memory address space to the accelerator's control registers and internal memories.

5.2 Register Map and Description

We use a memory-mapped I/O (MMIO) approach for early debugging and use the Linux kernel module in the final system. All control registers, status registers, and data memories within the accelerator are exposed to the HPS as a contiguous block of memory addresses.

The following figure shows the overall architecture of memory.



These registers are used for controlling the accelerator's operation and querying its status. They are 32-bits wide.

Offset Address	Register Name	Type	Description
0x00	REG_CTRL	R/W	Bit 0 (Start): write 1 to launch one inference pulse. Bit 1 (Reset): write 1 to trigger a soft reset. Bit 31:2 (Reserved).
0x04	REG_STATUS	R	Bit 0 (Busy): return 1 when inference is running. Bit 1 (Done): return 1 when inference completes.

			Bit 31:2 (Reserved).
0x08	REG_RESULT	R	Result Register. Stores the final inference result (digit 0-9).
0x0C	REG_CYCLES	R	Cycle Count Register. Stores the total hardware clock cycles for the last inference.

These regions are for transferring data between the HPS and the accelerator, such as model parameters (weights, biases) and the input image.

Offset Address	Memory Block	Description
0x00010000	MEM_INPUT	Stores 32x32 pixels. Stored as 1 byte per pixel (int 8).
0x00020000	MEM_CONV1	Memory for CONV1 layer weights and biases.
0x00030000	MEM_CONV2	Memory for CONV2 layer weights and biases.
0x00040000	MEM_FC1	Memory for FC1 layer weights and biases.
0x00050000	MEM_FC2	Memory for FC2 layer weights and biases.
0x00060000	MEM_FC3	Memory for FC3 layer weights and biases.
0x00070000	DBG_CONV1_OUT	Debug memory region to read the output of the CONV1 layer.
0x00080000	REG_FC1_OUT	Debug memory region to read the output of the FC1 layer.

5.3 Data Access Protocol

1. Initialization:
    a. Reset the accelerator: writing a 1 to the Reset bit of REG_CTRL (0x00).

b. Load model: load the trained LeNet-5 model parameters (weights and biases) into the corresponding memory regions (MEM_CONV1, MEM_CONV2, etc.) starting at offset 0x00020000.
c. Load input: load the input image data into the MEM_INPUT region at offset 0x00010000.

2. Single Inference Cycle:
   a. Start Computation: Initiates the inference process by writing a 1 to the Start bit of REG_CTRL. This action also clears the Done flag from the previous run.
   b. Poll for Completion: The software enters a loop, periodically reading the REG_STATUS register at offset 0x04. It waits until the Done bit (bit 1) is set to 1. Alternatively, it can check that the Busy bit (bit 0) has returned to 0.
   c. Retrieve Result: Once the Done bit is 1, the computation is finished. The software can read the 8-bit classification result from the REG_RESULT register at offset 0x08.
   d. (Optional) Read Performance Metrics: Read the 32-bit cycle count from the REG_CYCLES register at offset 0x0C for profiling.

5.4 C Header for Data Access

This C header file defines the necessary constants and data structures for interacting with the FPGA from the Linux user space. In the final system, a dedicated Linux kernel module (device driver) will be used to replace the direct Memory-Mapped I/O (MMIO) approach used in early debugging.

```C/C++
#include <linux/ioctl.h>

// ****************************************************************
// **                  Register Map and Constants                **
// ****************************************************************

// Base address for these offsets is determined by the platform device tree
#define REG_CTRL_OFS      0x00 // Control Register (R/W)
#define REG_STATUS_OFS    0x04 // Status Register (R)
#define REG_RESULT_OFS    0x08 // Final Result Register (R)
#define REG_CYCLE_OFS     0x0C // Performance Cycle Counter (R)

// Memory block offsets
#define MEM_INPUT_IMG_OFS 0x10000 // 32x32 Input Image Memory
#define MEM_WEIGHTS_OFS   0x20000 // All layer weights and biases

// --- REG_CTRL Bit Fields ---
#define CTRL_START (1 << 0) // Write 1 to start inference
```

```
#define CTRL_RESET (1 << 1) // Write 1 to reset the accelerator

// --- REG_STATUS Bit Fields ---
#define STATUS_BUSY (1 << 0) // 1 = Accelerator is busy
#define STATUS_DONE (1 << 1) // 1 = Inference complete, result is ready

// ##########################################################################
// ##                      IOCTL Command Definitions                       ##
// ##########################################################################

#define ACCEL_MAGIC 'L'

// IOCTL command to run a full inference cycle
// Takes a pointer to an accelerator_inference_t object
#define ACCEL_RUN_INFERENCE _IOWR(ACCEL_MAGIC, 1, accelerator_inference_t)

// IOCTL command to load weights into the accelerator's memory
// Takes a pointer to an accelerator_weights_t object
#define ACCEL_LOAD_WEIGHTS _IOW(ACCEL_MAGIC, 2, accelerator_weights_t)

// IOCTL command to reset the accelerator
#define ACCEL_RESET _IO(ACCEL_MAGIC, 3)

// ##########################################################################
// ##                   Data Structures for IOCTL                          ##
// ##########################################################################

// Structure for the inference command
typedef struct {
    unsigned char image[1024]; // Input: 32x32 image data
    unsigned char result;      // Output: The classified digit (0-9)
    unsigned int  cycles;      // Output: Clock cycles taken for inference
} inference;

// Structure for loading weights
typedef struct {
    void *weights_user_addr;  // Pointer to the weights data in user-space memory
    unsigned int  size;       // Total size of weights data in bytes
} accelerator_weights_t;
```

Reference:

[1] https://poloclub.github.io/cnn-explainer/