Design Document: MNIST Inference Accelerator

CSEE 4840 Spring 2026 — Final Project

Ifesi Onubogu (io2249)
Colin Paul Jaworowski (cpj2118)
Columbia University

April 17, 2026

Project folder: https://github.com/Nuk3-W/EmbeddedLab3/tree/main/project

---

Abstract

A hardware accelerator on the DE1-SoC that classifies handwritten digits from the MNIST dataset by running a small INT8-quantised multi-layer perceptron ($784 \to 128 \to 10$) inside the FPGA fabric. The network weights and the test images live on the SD card; on boot, a Linux kernel driver on the HPS streams them into on-chip BRAM across the Avalon-MM lightweight bridge; an interactive userspace picker on the host laptop (connected to the board's serial console) chooses an image, triggers one inference, and prints the predicted digit while HEX0 on the board lights up with the same digit.

Contents

0 Plain-English walkthrough **3**

1 Introduction **5**

2 System Block Diagram **6**
- 2.1 Block descriptions . . . 6
- 2.2 Communication pathways . . . 8

3 Algorithms **9**
- 3.1 Integer-exact MLP forward pass . . . 9
- 3.2 Quantisation contract (software side) . . . 9
- 3.3 FSM sequencing . . . 10
- 3.4 End-to-end inference flow . . . 11

4 Resource Budgets **12**
- 4.1 On-chip memory (M10K blocks) . . . 12
- 4.2 Quartus fit report (Cyclone V 5CSEMA5F31) . . . 12
- 4.3 Compute budget per inference . . . 12
- 4.4 Weight / image storage budget . . . 12

5 Hardware/Software Interface 13
5.1 Register map (summary) . . . 13
5.2 CONTROL (`0x00`, W) . . . 13
5.3 STATUS (`0x04`, R) . . . 14
5.4 RESULT (`0x08`, R) and CONFIDENCE (`0x0C`, R) . . . 14
5.5 LOAD streaming interface (`0x10`/`0x14`/`0x18`) . . . 14
5.6 VERSION (`0x1C`, R) . . . 15
5.7 Memory map (address space of each load target) . . . 15
5.8 Datapath detail . . . 15
5.9 Linux driver `ioctl` interface . . . 15
5.10 Device-tree entry . . . 17
5.11 Userspace picker (`mnist_pick`) . . . 17

A File index 18

B Plan B — baked-weights fallback 18

0 Plain-English walkthrough (read this first)

What it does, in one sentence. You boot the DE1-SoC, open a serial terminal on your laptop, type a digit 0–9 at a prompt, and the board's rightmost seven-segment display (HEX0) lights up with the digit our neural network predicts for the corresponding handwritten image — in under 5 ms of compute time, all happening in custom FPGA hardware.

Why an FPGA? A laptop CPU could do this same inference in microseconds. The point of the project is not speed — it's the exercise of taking a trained neural network and implementing it as *custom digital hardware*, the way a commercial AI chip (Google TPU, Tesla FSD, etc.) would be built. An FPGA is the cheapest way to prototype that kind of circuit.

The algorithm, in plain English. We run a small pre-trained network that classifies MNIST digits:

1. Input: 784 numbers (a 28×28 greyscale image, flattened).
2. Layer 1 (*FC1*): 128 neurons, each of which multiplies all 784 pixels by trained weights, adds them up, adds a bias, and clamps negatives to zero (ReLU).
3. Layer 2 (*FC2*): 10 neurons (one per digit), each doing the same kind of multiply-accumulate over the 128 hidden values.
4. *Argmax*: whichever output neuron has the largest value wins, that's the predicted digit.

Total math: 101,632 multiply-accumulates per image. Our FPGA does them sequentially with a single multiplier, so one inference takes $\sim$4 ms.

Where the weights come from. We don't train on the FPGA — training needs floating-point math. We train in PyTorch on a laptop in three steps (Figure 3): `train.py` produces a float `mlp.pt`; `export.py` rounds its weights to 8-bit integers and dumps `*.bin` files; `golden.py` runs the same integer forward pass with numpy to produce a "golden" trace the RTL must match bit-for-bit.

How the weights physically get onto the FPGA. The `.bin` files and the ten sample images all live on the **SD card**, in the same partition Linux boots from. When the board boots, U-Boot configures the FPGA with our `.rbf` bitstream, Linux starts, our kernel module (`mnist_accel.ko`) claims the peripheral, and our userspace picker (`mnist_pick`) reads the weight files and pushes them into the FPGA over the Avalon-MM lightweight bridge. When you type a digit at the prompt, the same path streams the chosen image in, then `ioctl(MNIST_INFER)` kicks the hardware FSM off. A poll-loop in the driver waits for `STATUS.done`, reads `RESULT[3:0]`, and HEX0 shows the answer.

Why "bit-exactness" is such a big deal. *We trained in float. The hardware runs in integer.* Float→integer rounding introduces tiny errors. If any weight rounds differently in PyTorch vs SystemVerilog, one of the 128 hidden values shifts, and the argmax might pick an entirely different digit. When that happens, it's impossible to tell whether the RTL has a bug, the quantisation script has a bug, or they just disagreed by one somewhere. Fix: a Python "golden" that does the exact same integer math the RTL does, and a SystemVerilog testbench that asserts every one of $128+10+1 = 139$ intermediate values per inference. Zero mismatches is the gate before any bitstream gets synthesised.

What you get when the demo runs.

```
# cd /mnt/sdcard/mnist/
# ./run_on_board.sh
[build] compiling driver + userspace
[insmod] sw/mnist_accel.ko
[dmesg] mnist_accel: probed at 0xff200040, VERSION=0x4D4E5301
[run] ./sw/mnist_pick weights
mnist_pick -- loading weights from weights/
  loading fc1_w.bin (100352 bytes)...
  loading fc1_b.bin (512 bytes)...
  loading fc2_w.bin (1280 bytes)...
  loading fc2_b.bin (40 bytes)...
  weights loaded.

  pick an image:
    0..9  run inference on img_N.bin
    a     run all ten in sequence
    q     quit
> 3
  sending weights/img_3.bin ... predicted 3 (confidence 1348271) OK
```

and HEX0 simultaneously shows the digit.

1 Introduction

This project is a hardware accelerator on the DE1-SoC that classifies a 28×28 handwritten digit from the MNIST dataset by running a small INT8-quantised multi-layer perceptron entirely inside the FPGA fabric. The network architecture is $784 \rightarrow 128 \rightarrow 10$ with ReLU between the two fully-connected layers and an argmax on the output — the smallest shape we could find that still reaches $\geq 96\%$ test accuracy.

The deliverable is a custom SystemVerilog peripheral, `mnist_accel`, that lives in the FPGA fabric and speaks the Avalon-MM lightweight-bridge protocol. On each inference it consumes a 784-pixel signed-INT8 image from on-chip BRAM, runs two sequential integer MAC loops, and latches the predicted digit into a result register that an on-chip seven-segment decoder drives straight onto `HEX0`.

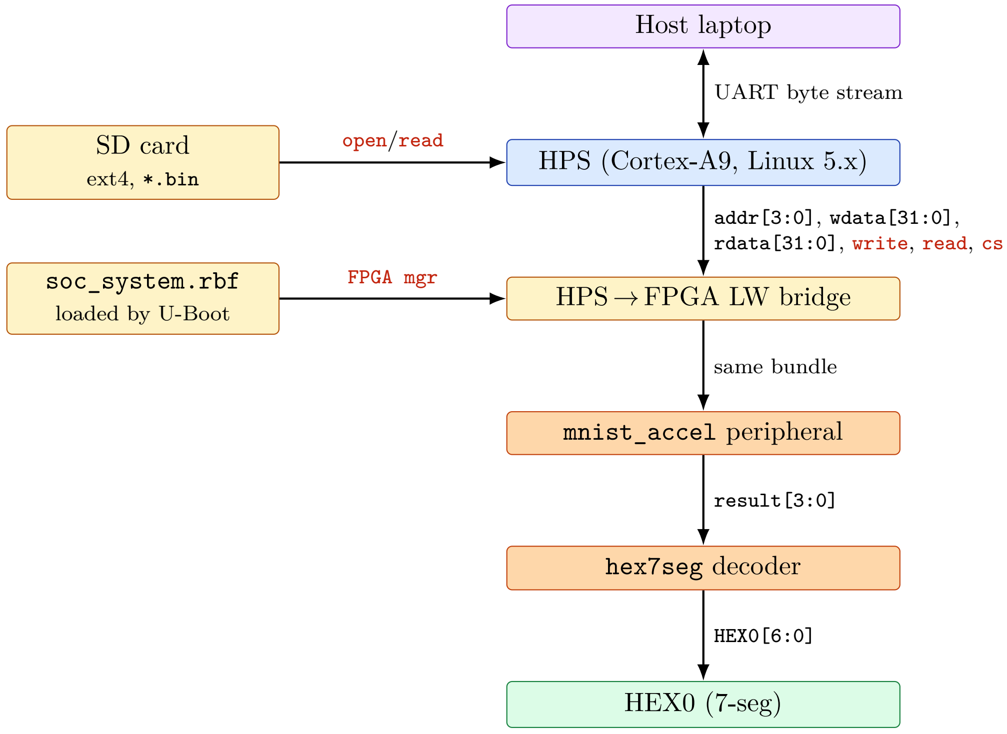
System flow. The *working configuration* is end-to-end software-driven: weights and sample images live on the SD card; a Linux kernel driver on the HPS (`mnist_accel.ko`) exposes `/dev/mnist_accel` with an `ioctl` interface; a userspace picker (`mnist_pick`) streams weights into on-chip BRAM at startup and then loops on an interactive prompt so the user (on a laptop, connected to the board's serial console) can choose which image to classify.

Design principle. The driving principle is *bit-exactness between software and hardware.* A numpy "golden" reference performs the exact same integer forward pass the RTL will run, dumps every accumulator value to a trace file, and the SystemVerilog testbench asserts against that trace byte-for-byte: 128 FC1 hidden values + 10 FC2 outputs + 1 argmax = 139 assertions per image, required to be exact before synthesis.

Deliberate MVP scope cuts. Single sequential MAC (no parallelism); poll-based completion (no interrupts); one image at a time (no DDR3-resident test set); HEX0 only (no VGA, no LEDR). The goal is end-to-end correctness first; parallelism is a follow-up. Appendix B documents an alternate bring-up path (*Plan B*) that skips Linux entirely by baking the weights into the bitstream via `$readmemh`; it was used during early bring-up while the HPS Linux image was being debugged, and remains a valid fallback.

2 System Block Diagram

Figure 1 shows the on-board data path from the host laptop's serial terminal all the way to `HEX0`. Every block that the system comprises is drawn explicitly; every arrow is catalogued in §2.2. Figure 3 shows the offline training pipeline that produces the weight blobs that live on the SD card.



Signal-name convention: `red` = control (directs when / how data moves); black = data. Full Avalon-MM bundle: `address[3:0]`, `writedata[31:0]`, `readdata[31:0]`, `write`, `read`, `chipselect`, `clk`, `reset`.

Figure 1: On-board system block diagram — top-down dataflow on the vertical spine, side inputs on the right (SD-card file reads, initial FPGA configuration). The full Avalon-MM signal bundle carried on the `bridge` → `peripheral` arrow is listed below the diagram. Peripheral internals are expanded in Fig. 2.

2.1 Block descriptions

Host laptop (serial terminal). Any terminal that speaks 115200 8N1 over USB-UART — `minicom`, `picocom`, PuTTY, Windows Terminal. Receives boot messages from U-Boot and Linux, sends keystrokes into `mnist_pick`'s prompt, and receives the predicted-digit printouts.

SD card. Standard DE1-SoC Linux image partition; we add `mnist/weights/*.bin`, `mnist/weights/img_*.bin` and `mnist/sw/mnist_accel.ko` alongside. U-Boot additionally loads `soc_system.rbf` and `soc_system.dtb` from the same partition before starting the kernel.

HPS (ARM Cortex-A9, Linux 5.x). Runs the standard lab-3 Linux image, our kernel module `mnist_accel.ko`, and our userspace `mnist_pick`. The kernel module's probe matches the device-

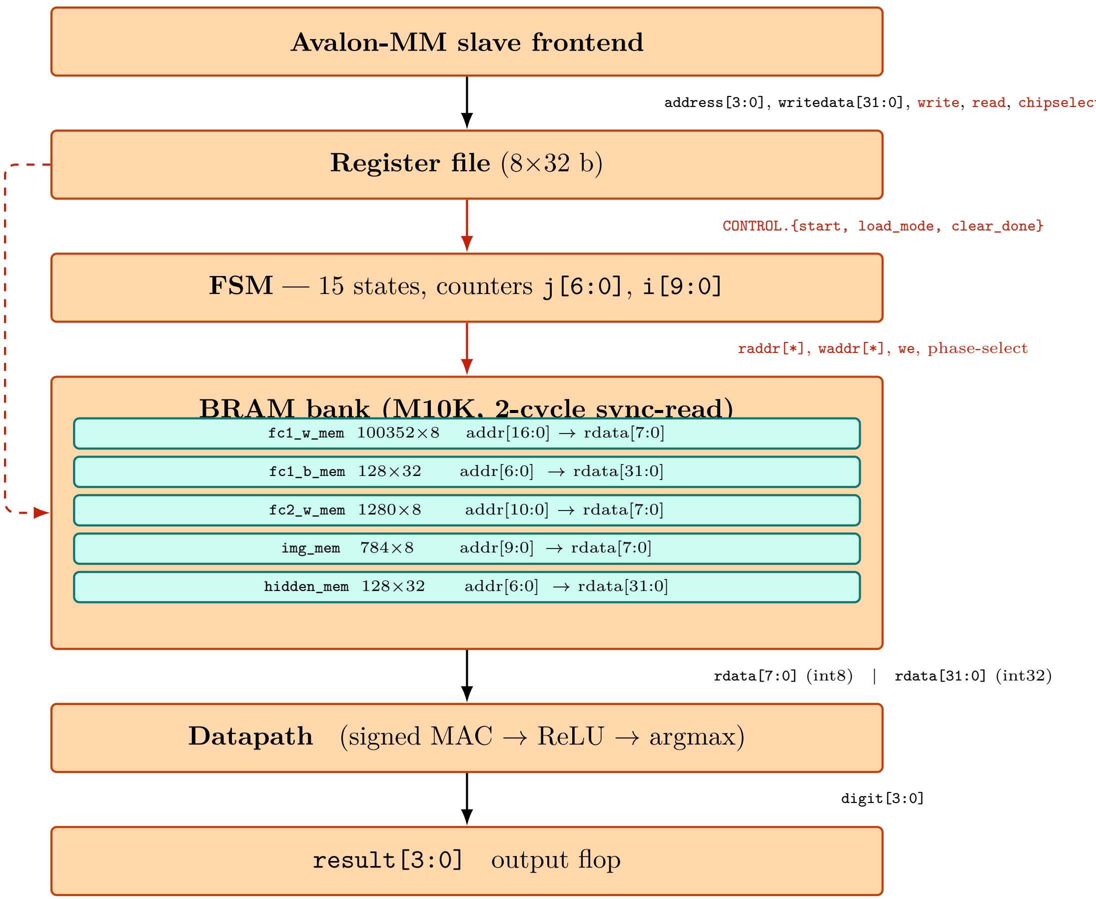


Figure 2: `mnist_accel` peripheral internals, rendered as a strict vertical stack — every primary arrow is a straight-down line, and each arrow's label is the actual signal bundle crossing that boundary (net names match `project/hw/mnist_accel.sv`). The *dashed* arrow on the left is the load path: when `CONTROL.load_mode` is asserted, the register file's `LOAD_DATA[7:0]`, `LOAD_ADDR[16:0]` and `LOAD_TARGET[2:0]` fields bypass the FSM and drive the *write* ports of the selected BRAM directly. The register file's `readdata[31:0]` port returns to the Avalon frontend (not drawn). `result[3:0]` leaves the bottom of the figure and continues into `hex7seg` in Fig. 1.

tree node `csee4840,mnist_accel-1.0` and calls `of_iomap` to map the peripheral into kernel virtual memory; all `iowrite32` accesses go through the HPS-to-FPGA lightweight AXI/Avalon bridge.

HPS-to-FPGA lightweight bridge. A 32-bit Avalon-MM master on the HPS side, mapped at `0xFF200000` in HPS physical memory. We place `mnist_accel` at `0xFF200040` (byte offset `0x40` into the bridge window); see §5.

`mnist_accel` peripheral. Custom SystemVerilog module (`project/hw/mnist_accel.sv`, ∼610 lines). Contains six BRAMs, an FSM, one 8×8 signed multiplier, one 32×8 signed multiplier, an argmax comparator, and six 7-seg decoders for the HEX pins. The Avalon-MM slave front-end exposes eight 32-bit registers (§5.1).

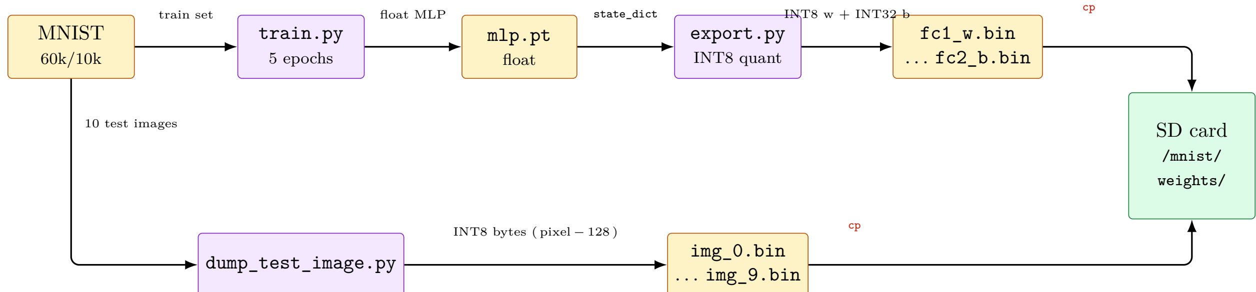


Figure 3: Offline training + image-dump pipelines — runs on a development laptop, not on the board. Top row is the training chain: PyTorch trains a float MLP, `export.py` quantises it to INT8 weights and INT32 biases. Bottom row is the image-dump chain: `dump_test_image.py` pulls ten sample digits from the MNIST test set and writes each as pre-shifted signed-INT8 bytes. Both chains' outputs are copied onto the SD card. *Not shown in this figure:* `golden.py` — a separate numpy-integer forward pass that reads the same four weight blobs and one image blob and writes `golden_trace.txt`, the per-neuron accumulator log the SystemVerilog testbench asserts against. The golden is a verification tool, not part of the deployment flow.

HEX0 (7-seg display). Active-low; combinationally decoded from `result[3:0]` via the Lab-1 `hex7seg` module. HEX1–HEX5 are tied off to `7'h7f` (blank) in this MVP.

2.2 Communication pathways

Every arrow in Figure 1 corresponds to a single communication pathway; Table 1 catalogues them. The system has exactly these nine pathways — there is no other data that crosses block boundaries during the working demo.

#	Endpoints	Direction	Mechanism	Width / size
1	Dev PC → SD card	one-way	`cp` over card reader	∼300 kB (weights + kernel modul
2	SD card → FPGA config	one-way	U-Boot loads `.rbf` via FPGA manager	7 MB
3	SD card → HPS Linux	read	ext4 filesystem reads	as needed
4	Laptop ↔ HPS	full-duplex	USB-UART serial, 115200 8N1	byte stream
5	HPS → FPGA peripheral	write-driven	Avalon-MM, `iowrite32`	32-bit words
6	FPGA peripheral → HPS	read-driven	Avalon-MM, `ioread32`	32-bit words
7	FSM → BRAM	internal	registered address bus per BRAM	17 b / 11 b / 10 b / 7 b / 4 b
8	BRAM → datapath	internal	registered `rdata` bus (2-cycle lat.)	8 b or 32 b
9	Datapath → HEX0	internal	4 b → `hex7seg` → 7 b (active-low)	7 b

Table 1: Communication pathways. Rows 1–2 are one-time per power-up; rows 3–6 drive one inference; rows 7–9 are internal peripheral wiring exercised on every MAC cycle.

3 Algorithms

3.1 Integer-exact MLP forward pass

The joint specification for the Python golden and the SystemVerilog RTL.

Notation. $x \in \mathbb{Z}^{784}$: pixels pre-shifted by -128 to signed INT8. $W_{1q} \in \mathbb{Z}^{128\times784}$: FC1 weights, signed INT8. $b_{1q} \in \mathbb{Z}^{128}$: FC1 biases, pre-scaled INT32. $W_{2q} \in \mathbb{Z}^{10\times128}$: FC2 weights, signed INT8. $b_{2q} \in \mathbb{Z}^{10}$: FC2 biases, pre-scaled INT32.

```
Algorithm 1: Integer MLP forward pass (RTL + golden must agree bit-for-bit)

for j in 0..127:
    acc <- b1_q[j]                          # int32
    for i in 0..783:
        acc <- acc + W1_q[j,i] * x_q[i]     # int8 * int8 = int16; acc stays int32
    hidden[j] <- max(0, acc)                # ReLU; int32

for j in 0..9:
    acc <- b2_q[j]                          # int48
    for i in 0..127:
        acc <- acc + W2_q[j,i] * hidden[i]  # int8 * int32 = int40; acc stays int48
    out[j] <- acc

result <- argmax_j out[j]                   # strict >, as in numpy
```

Invariants the RTL enforces by construction (§5.8):

- FC1 accumulator is signed INT32; worst case $784 \times 127^2 \approx 1.26 \times 10^7$, well inside range.
- Hidden activations are INT32 — no inter-layer requantisation.
- FC2 accumulator is signed INT48 (32+8=40-bit partial, 128-deep sum gains 7 bits, 48 bits has one bit of headroom).
- Argmax is scale-invariant, so the predicted digit does not depend on the residual scale.

3.2 Quantisation contract (software side)

The single most failure-prone interface in the project. An off-by-one in the rounding rule produces a bit mismatch that Python cannot see but the RTL will. The contract:

- Input scale is *fixed*: $s_x = 1/128$, so $x_q = \text{pixel} - 128$ with no runtime scale computation.
- Per-tensor symmetric weight quantisation: $s_w = \max(|W|)/127$, $W_q = \text{round}(W/s_w).\text{clip}(-128, 127).\text{astype}($
- FC1 bias pre-scaled into accumulator units: $b_{1q} = \text{round}(b_1/(s_{w1} \cdot s_x)).\text{astype}(\texttt{int32})$.
- FC2 bias pre-scaled: $b_{2q} = \text{round}(b_2/(s_{w2} \cdot s_h)).\text{astype}(\texttt{int32})$, where $s_h = s_{w1} \cdot s_x$ is the implicit scale of the INT32 hidden activations.
- The entire golden forward pass runs in `np.int32` / `np.int64` — never in float.
- The RTL produces the same int32/int48 values exactly.

3.3 FSM sequencing

The BRAMs have a two-cycle synchronous read latency (address flopped in one `always_ff`, `rdata` flopped in another), which would produce a drain bug in a pipelined MAC. We chose the deliberately simple two-cycles-per-MAC schedule shown in Figure 4; the total is ∼4 ms per inference at 50 MHz, far under any interactive threshold.

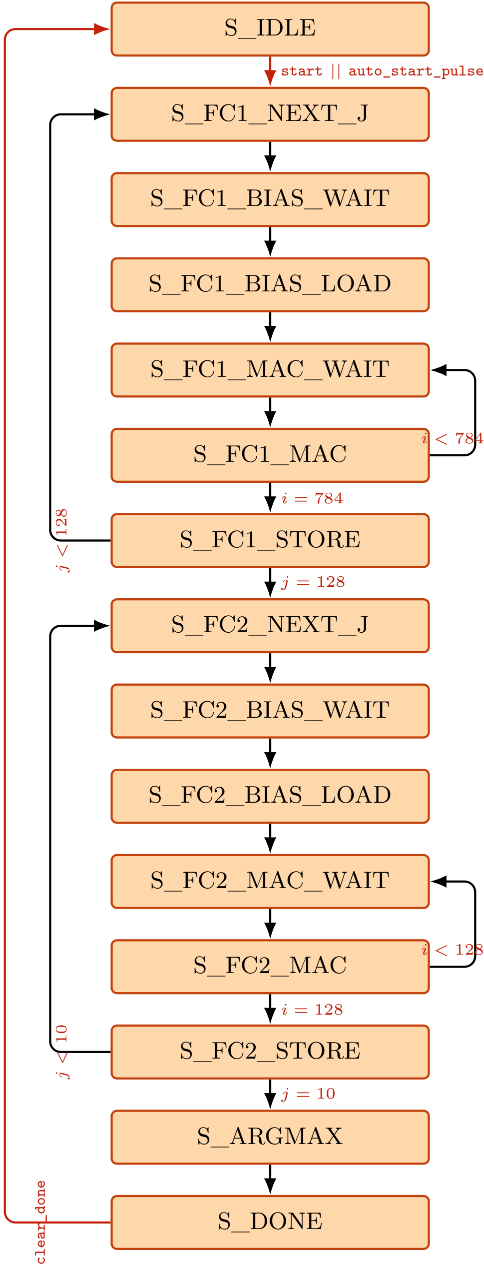


Figure 4: FSM state graph — one state per row, top-to-bottom. Inner *MAC loops* arc on the right side: `FC1_MAC` loops back to `FC1_MAC_WAIT` while $i < 784$, same for FC2 with $i < 128$. Outer *per-neuron loops* arc on the left: `FC1_STORE` returns to `FC1_NEXT_J` while $j < 128$, FC2 while $j < 10$. The outermost `clear_done` arc returns `S_DONE` to `S_IDLE` (the FSM then waits for the next `start` pulse).

Per-state actions.

```
S_FC1_NEXT_J      schedule fc1_b_raddr <- j
S_FC1_BIAS_WAIT   (BRAM latency cycle)
S_FC1_BIAS_LOAD   capture bias -> acc_fc1; pre-issue first MAC read (i=0)
S_FC1_MAC_WAIT    (BRAM latency cycle for w and x)
S_FC1_MAC         acc_fc1 += w_q * x_q; i++; pre-issue next read
S_FC1_STORE       hidden_mem[j] <- max(0, acc_fc1); j++; if j<128 goto NEXT_J
S_FC2_*           same structure, with hidden_mem instead of img_mem
S_ARGMAX          10-cycle linear scan; strict > tie-break
S_DONE            latch result, assert STATUS.done (sticky), return to IDLE
```

Cycle count. FC1: $128 \times (3+784\times 2) \approx 201{,}000$ cycles ≈ 4.0 ms. FC2: $10\times(3+128\times 2) \approx 2{,}600$ cycles ≈ 0.05 ms. Argmax + plumbing: $\sim$30 cycles. **Total** $\sim$4.0 **ms per inference.**

3.4 End-to-end inference flow

Ties the hardware FSM and the software side together:

1. User boots the board; U-Boot configures the FPGA from `soc_system.rbf`, boots Linux from the SD card.
2. Over the serial console, user runs `./run_on_board.sh`; this insmods `mnist_accel.ko` and launches `mnist_pick weights`.
3. `mnist_pick` opens `/dev/mnist_accel`, reads the four `.bin` weight files from the SD card, and pushes each through a dedicated `ioctl` (`MNIST_LOAD_FC1_W`, ...). The driver translates each ioctl into a sequence of `iowrite32` bursts into `LOAD_DATA` (§5.5).
4. The user types `3` at the prompt. `mnist_pick` reads `weights/img_3.bin` (784 signed-INT8 bytes), calls `ioctl(MNIST_LOAD_IMAGE)` to stream it into `img_mem`, then `ioctl(MNIST_INFER)`.
5. The driver writes `CONTROL = 0` (clears load_mode) then `CONTROL = 1` (sets start). The FSM walks the state graph of Figure 4 and asserts `STATUS.done` $\sim$4 ms later.
6. Driver polls `STATUS.done`, reads `RESULT[3:0]`, writes `CONTROL.clear_done`, returns the result struct to userspace.
7. `mnist_pick` prints `predicted 3 (confidence ...)`. `HEX0` was already showing the digit $\sim$4 ms earlier (it's combinationally driven by `result[3:0]`, which latched at `S_DONE`).
8. Control returns to the prompt; the user picks another image.

4 Resource Budgets

4.1 On-chip memory (M10K blocks)

BRAM	Depth × Width	Bits	Purpose
`fc1_w_mem`	100,352 × 8	802,816	FC1 weights, addressed $j \cdot 784 + i$
`fc1_b_mem`	128 × 32	4,096	FC1 biases (INT32, pre-scaled)
`fc2_w_mem`	1,280 × 8	10,240	FC2 weights, addressed $j \cdot 128 + i$
`fc2_b_mem`	10 × 32	320	FC2 biases (INT32, pre-scaled)
`img_mem`	784 × 8	6,272	Input image (INT8 pre-shifted)
`hidden_mem`	128 × 32	4,096	Post-ReLU FC1 activations
Total		**827,840**	≈ 101 kB

`out_regs` (10 × 48 b = 480 b) lives in flip-flops so argmax can read all ten in one cycle.

4.2 Quartus fit report (Cyclone V 5CSEMA5F31)

Resource	Used	Available	Utilisation
ALMs	957	32,070	3%
Registers	1,617	—	—
Block memory bits	827,840	4,065,280	20%
M10K blocks	104	397	26%
DSP blocks (27×27)	8	87	9%
Pins	362	457	79%

The design uses 20% of on-chip BRAM and 9% of the DSP slices; ALMs are essentially free at 3%. **Headroom is more than sufficient to add multi-MAC parallelism as a follow-up** — a 16-way parallel MAC bank would still fit with room to spare.

4.3 Compute budget per inference

Phase	Ops	Cycles	Wall time (50 MHz)
FC1 (784 MACs × 128 neurons, 2 cy/MAC)	100,352 MACs	∼201,000	∼4.0 ms
ReLU + store (128 writes)	—	128	∼2.6 µs
FC2 (128 MACs × 10 neurons, 2 cy/MAC)	1,280 MACs	∼2,600	∼0.05 ms
Argmax (10 neurons)	—	10	0.2 µs
Total per inference		**∼203,700**	**∼4.1 ms**

4.4 Weight / image storage budget

Everything fits trivially on the SD card and, once loaded, in on-chip BRAM — no DDR3 access required during inference.

Item	Count	Bytes per item	Total bytes
FC1 weights	100,352 × INT8	1	100,352
FC1 biases	128 × INT32	4	512
FC2 weights	1,280 × INT8	1	1,280
FC2 biases	10 × INT32	4	40
Image	784 × INT8	1	784
One run (weights + one image)			**102,968 B (∼101 kB)**
Ten sample images on SD	10 × 784 B		7,840 B
All weights + 10 images on SD			∼110 kB

5 Hardware/Software Interface

This is the heart of the system. The peripheral is an Avalon-MM slave with eight 32-bit registers at word addresses 0–7 (byte offsets `0x00`–`0x1C`), mapped at `0xFF200040` in HPS physical memory. Every register's bit layout is specified below; the Linux driver builds its `iowrite32`/`ioread32` accesses directly against these diagrams.

5.1 Register map (summary)

Offset	Word	Name	Access	HPS phys addr	Description
`0x00`	0	CONTROL	W	`0xFF200040`	start / load_mode / clear_done
`0x04`	1	STATUS	R	`0xFF200044`	busy / done (sticky)
`0x08`	2	RESULT	R	`0xFF200048`	predicted digit
`0x0C`	3	CONFIDENCE	R	`0xFF20004C`	winning FC2 accumulator (low 32)
`0x10`	4	LOAD_ADDR	R/W	`0xFF200050`	target-local write index
`0x14`	5	LOAD_DATA	W	`0xFF200054`	streaming data port
`0x18`	6	LOAD_TARGET	W	`0xFF200058`	selects destination memory
`0x1C`	7	VERSION	R	`0xFF20005C`	constant `0x4D4E5301`

5.2 CONTROL (`0x00`, W)

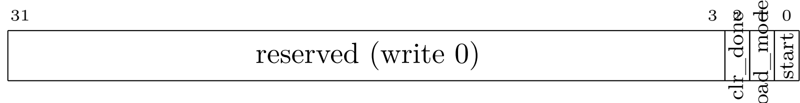


Figure 5: `CONTROL` register bit layout. All three used bits are self-documenting: `start` self-clears one cycle after assertion (it pulses the FSM); `load_mode` is sticky until software clears it; `clear_done` self-clears after pulsing `STATUS.done` low.

Software rules:

- Set `load_mode = 1` before any `LOAD_DATA` sequence.
- Clear `load_mode = 0` before asserting `start`.
- `start` is *ignored* if `load_mode = 1` or if the FSM is already busy.
- `clear_done` is edge-sensitive: write 1, self-clears.

5.3 STATUS (`0x04`, R)

Figure 6: `STATUS` register bit layout. `busy` is high while the FSM is in any state other than `S_IDLE` or `S_DONE`; `done` is *sticky* — set on entry to `S_DONE`, cleared only by `CONTROL.clear_done`.

Driver poll loop:

```
while ((ioread32(base + STATUS) & STATUS_DONE) == 0) {
    if (waited_us >= INFER_TIMEOUT_US)   return -ETIMEDOUT;
    udelay(100); waited_us += 100;
}
digit     = ioread32(base + RESULT) & 0xF;
confidence = (s32) ioread32(base + CONFIDENCE);
iowrite32(CTRL_CLEAR_DONE, base + CONTROL);
```

5.4 RESULT (`0x08`, R) and CONFIDENCE (`0x0C`, R)

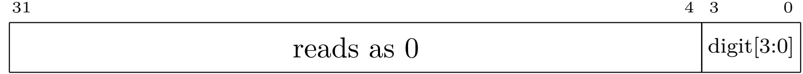


Figure 7: `RESULT` register bit layout. Low 4 b: predicted digit 0–9. Latched into flip-flops on entry to `S_DONE`; remains stable until the next `S_DONE`.

`CONFIDENCE` is the low 32 bits of the winning FC2 accumulator, truncated from the 48-bit internal value. The sign bit of the INT48 may or may not appear in `bit 31` depending on magnitude; this register is a debug view, not a calibrated probability.

5.5 LOAD streaming interface (`0x10`/`0x14`/`0x18`)

Figure 8: `LOAD_TARGET` register bit layout. `target[2:0]` selects one of five memories; see Table 2.

Load sequence (driver side). For each weight/image blob, the kernel driver executes:

```
iowrite32(CTRL_LOAD_MODE, base + CONTROL);       // enter load mode
iowrite32(target_id,      base + LOAD_TARGET);   // auto-clears LOAD_ADDR to 0
for (i = 0; i < len; i++)
    iowrite32((u32)buf[i], base + LOAD_DATA);    // writes[7:0] into BRAM[LOAD_ADDR++]
// stays in load mode; next load may follow, or infer clears it
```

Timing diagram of a load burst.

target[2:0]	Name	Destination BRAM	Element type	Total bytes	Writes to LOAD_DATA
0	FC1_W	fc1_w_mem	int8	100,352	100,352
1	FC1_B	fc1_b_mem	int32	512	128
2	FC2_W	fc2_w_mem	int8	1,280	1,280
3	FC2_B	fc2_b_mem	int32	40	10
4	IMAGE	img_mem	int8	784	784
5..7	(reserved)				

Table 2: LOAD_TARGET encoding. For INT8 targets (FC1_W, FC2_W, IMAGE) the driver writes one byte at a time in the low 8 bits of LOAD_DATA; LOAD_ADDR auto-increments by 1 per write regardless of target. Packing four bytes per 32-bit write was considered but deliberately skipped to eliminate a whole class of byte-ordering bugs — 100,000 single-byte bus writes at ∼1 µs each is still ∼100 ms boot-time, well within budget.

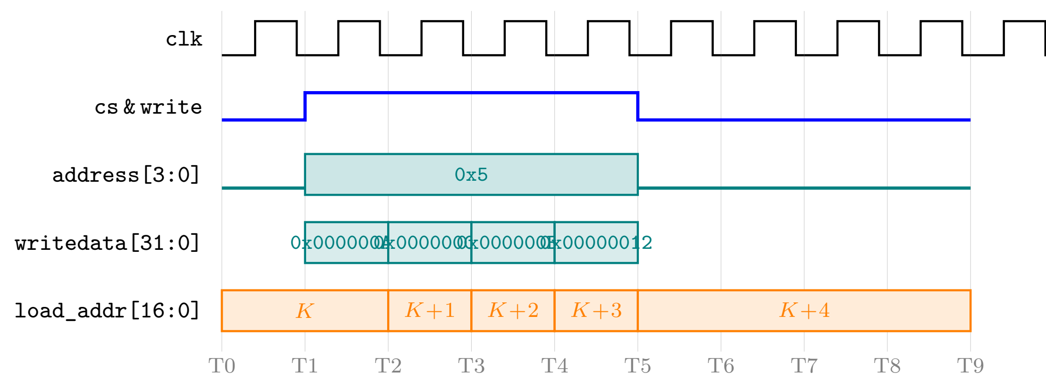


Figure 9: Four consecutive LOAD_DATA writes. Each Avalon write captures writedata[7:0] into the selected BRAM at the current load_addr, then increments load_addr. Writes are back-to-back at 50 MHz — one byte every clock — so a full 100 kB weight blob streams in 2 ms of bus time, with no software-side stall in the tight for loop.

5.6 VERSION (0x1C, R)

Returns the constant 0x4D4E5301 (ASCII "MNS" + 0x01). The driver's probe() reads this first as a sanity check that the device tree mapping worked.

5.7 Memory map (address space of each load target)

5.8 Datapath detail

Figure 11 shows the signed-integer widths at every stage. The one 8×8 multiplier (FC1 phase) and one 32×8 multiplier (FC2 phase) are time-shared by the FSM; synthesised as two DSP slices each, total 8 DSPs (since 32×8 splits across four 27×27 DSPs in Cyclone V).

5.9 Linux driver ioctl interface

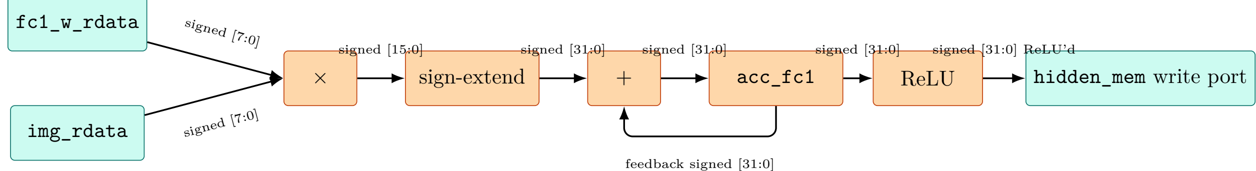
Seven ioctls on /dev/mnist_accel:

```
#define MNIST_LOAD_FC1_W  _IOW('m', 1, mnist_buf_t)   // 100352 bytes
#define MNIST_LOAD_FC1_B  _IOW('m', 2, mnist_buf_t)   //    512 bytes
#define MNIST_LOAD_FC2_W  _IOW('m', 3, mnist_buf_t)   //   1280 bytes
```

LOAD_TARGET	Name	Depth (indicative bar)	LOAD_ADDR range
3’d0	FC1_W		[0, 100352) int8
3’d1	FC1_B		[0, 128) int32
3’d2	FC2_W		[0, 1280) int8
3’d3	FC2_B		[0, 10) int32
3’d4	IMAGE		[0, 784) int8

Figure 10: Target-local address spaces. Bar widths are indicative, not to scale — FC1_W is actually 78× larger than FC2_W. On every `LOAD_DATA` write the hardware places `writedata[7:0]` (INT8 targets) or `writedata[31:0]` (INT32 targets) into the selected BRAM at position `LOAD_ADDR`, then increments `LOAD_ADDR` by 1. Writing `LOAD_TARGET` automatically resets `LOAD_ADDR` to 0. Row-major ordering for weight matrices: for `fc1_w`, address $j \cdot 784 + i$ holds $W_1[j, i]$.

(a) FC1 phase — signed 8×8 MAC, accumulated into `acc_fc1[31:0]`



(b) FC2 phase — signed 32×8 MAC, accumulated into `acc_fc2[47:0]`

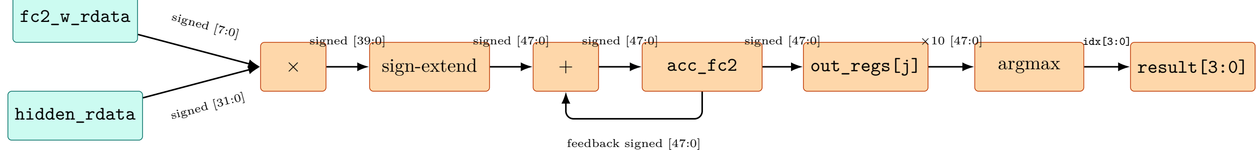


Figure 11: Datapath, with explicit signed bit widths on every wire. **(a)** FC1 phase runs 784 iterations of a signed 8×8 MAC per output neuron, accumulating into a 32-bit register; the bottom feedback arrow is the self-accumulate path, asserted every `S_FC1_MAC` cycle. ReLU is a combinational $\max(0, \cdot)$ that feeds the write port of `hidden_mem` at `S_FC1_STORE`. **(b)** FC2 phase runs 128 iterations of a signed 32×8 MAC per output neuron, accumulating into a 48-bit register; after all 10 neurons finish, argmax linearly scans `out_regs[0..9]` with a strict-> comparator (matches numpy’s tie-break). The two multipliers are time-shared by the FSM and are never active in the same cycle. Sign-extension is declared explicitly in the SystemVerilog via `$signed` casts.

```
#define MNIST_LOAD_FC2_B   _IOW('m', 4, mnist_buf_t)   //     40 bytes
#define MNIST_LOAD_IMAGE   _IOW('m', 5, mnist_buf_t)   //    784 bytes
#define MNIST_INFER        _IOR('m', 6, mnist_result_t)
#define MNIST_GET_RESULT   _IOR('m', 7, mnist_result_t)
```

`mnist_buf_t` = {`const void *buf, unsigned int len`}; the kernel `copy_from_users` the whole buffer, verifies the length matches the expected constant, and streams through `LOAD_DATA` as in §5.5. `mnist_result_t` = {`unsigned int digit, int confidence`}.

5.10 Device-tree entry

Generated by the Qsys flow and loaded by U-Boot alongside the bitstream:

```
mnist_accel_0: mnist_accel@40 {
    compatible = "csee4840,mnist_accel-1.0";
    reg = <0x40 0x20>;                     /* 32 bytes at 0xFF200040 */
};
```

The `of_match_table` in the driver matches `csee4840,mnist_accel-1.0`; `of_iomap` returns a kernel virtual address into the lightweight-bridge window.

5.11 Userspace picker (`mnist_pick`)

Tiny loop:

```
int fd = open("/dev/mnist_accel", O_RDWR);
push_weights(fd, "weights/");          // once, ~300ms over the bus
while (fgets(line, stdin)) {
    char c = first_non_space(line);
    if (c == 'q') break;
    if (c >= '0' && c <= '9') {
        snprintf(path, "%s/img_%c.bin", dir, c);
        push_blob(fd, MNIST_LOAD_IMAGE, path, 784);
        ioctl(fd, MNIST_INFER, &result);
        printf("predicted %u (confidence %d)\n", result.digit, result.confidence);
    }
}
```

A File index

Path	Role
project/hw/mnist_accel.sv	Accelerator RTL (Avalon slave, BRAMs, FSM, MAC, argmax, hex)
project/hw/hex7seg.sv	Active-low 7-seg decoder (from Lab 1)
project/hw/mnist_accel_hw.tcl	Qsys component descriptor
project/hw/soc_overlay/*	Patches to soc_system.qsys / .dts / _top.sv
project/tb/tb_mnist_accel.sv	Self-checking Questa testbench (139 asserts)
project/tb/run.do	vsim driver script
project/train/train.py	PyTorch training
project/train/export.py	INT8 quantisation
project/train/golden.py	Numpy integer reference
project/train/dump_test_image.py	MNIST sample images
project/sw/mnist_accel.c	Linux kernel module
project/sw/mnist_accel.h	Shared kernel/user ioctl header
project/sw/mnist_pick.c	Interactive userspace picker
project/sw/mnist_run.c	Batch userspace (runs one image from argv)
project/sw/Makefile	Builds .ko + userspace
project/sdcard_stage/	Stage directory ready to copy onto the SD card
project/sdcard_stage/run_on_board.sh	Top-level launcher (insmod + pick)
project/scripts/apply_soc_overlay.sh	Copies hw/ into lab3-hw/
project/scripts/regen_qsys.sh	Regenerates soc_system.v after Qsys edits

B Plan B — baked-weights fallback

During early bring-up, the HPS Linux image was not yet usable on our Windows/WSL development environment, so we prototyped a *software-free* path we call *Plan B*. The same RTL is used with two small additions that disappear when the software path is active:

1. **`$readmemh` initial blocks** inside each `initial` block of the five input-side BRAMs (`fc1_w`, `fc1_b`, `fc2_w`, `fc2_b`, `img`). These cause Quartus to embed the contents of the corresponding `.hex` files directly into the bitstream at synthesis time, so the on-chip BRAMs come up populated after configuration.

2. **A 16-bit auto-start counter** that pulses a one-shot signal ~1.3 ms after reset, equivalent to `CONTROL.start`. This causes the FSM to run one inference with no software involvement at all. Gated on `!load_mode` so the pulse is harmlessly eaten whenever the Avalon slave is actually being used.

Plan B is loaded over USB Blaster (Quartus Programmer) instead of the SD card; HEX0 shows the predicted digit ~1.3 ms after configuration finishes. To classify a different sample digit, the script `project/scripts/swap_image.sh` re-bakes a different `img_N.hex` into the synthesis inputs, after which the user re-compiles in Quartus and re-flashes. The accuracy on hardware matches the golden: 9/10 sample digits classify correctly (the 5→6 confusion is a property of the trained float model, not of the quantisation).

Plan B is retained as a demo fallback; the default demo is the SD-card path described in the body of this document.