MNIST Inference Accelerator on the DE1-SoC — Final Project Report

Ifesi Onubogu (‘io2249‘) · **Colin Paul Jaworowski (‘cpj2118‘)** · CSEE 4840 Spring 2026 · Columbia University

Abstract. We built a custom hardware accelerator on the DE1-SoC that classifies handwritten digits from the MNIST dataset. The accelerator runs a small INT8-quantised multi-layer perceptron ($784 \to 128 \to 10$) entirely inside the FPGA fabric, with 64-way parallel MACs in the first fully-connected layer, and exchanges data with a Linux process on the HPS over the lightweight bridge. A laptop client lets a user either draw a digit interactively (Tkinter + TCP) or run an automated 10 000-image validation pass. On the canonical MNIST test split the hardware achieves **96.53 %** accuracy — essentially identical to the 96.4 % float baseline — with zero protocol-layer failures, in ~**90 µs** of FPGA inference time per digit. The project is on the `Parallel` branch of `https://github.com/Nuk3-W/EmbeddedLab3`.

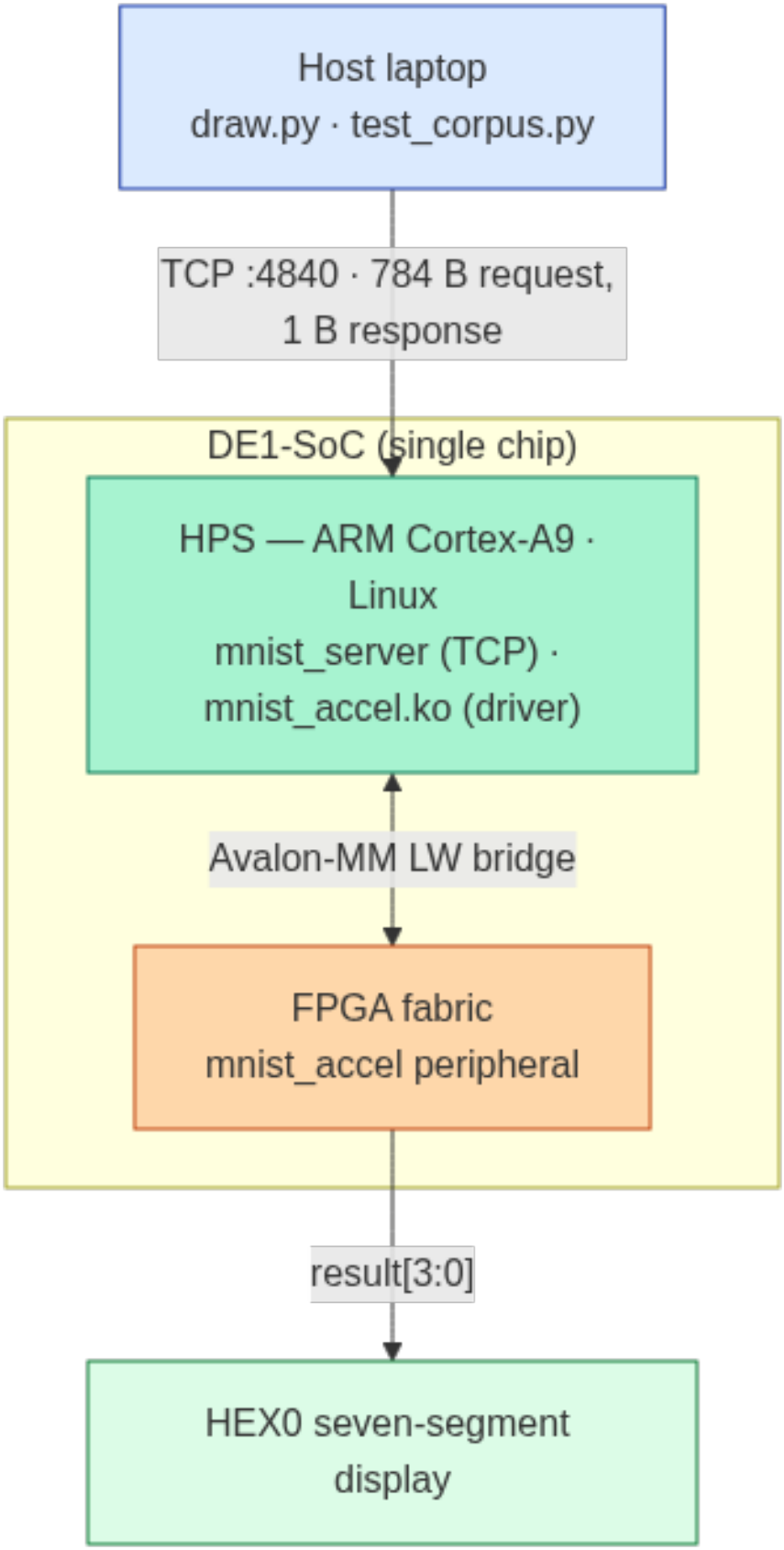
Project overview

What we built

You draw a digit on your laptop. Roughly one millisecond later, the FPGA shows what it thinks the digit is on the on-board seven-segment display, and the laptop receives the prediction over TCP. The "90 ţs inference" headline figure refers to the time the FPGA peripheral spends doing the math — everything else in the round-trip is network and Python overhead.

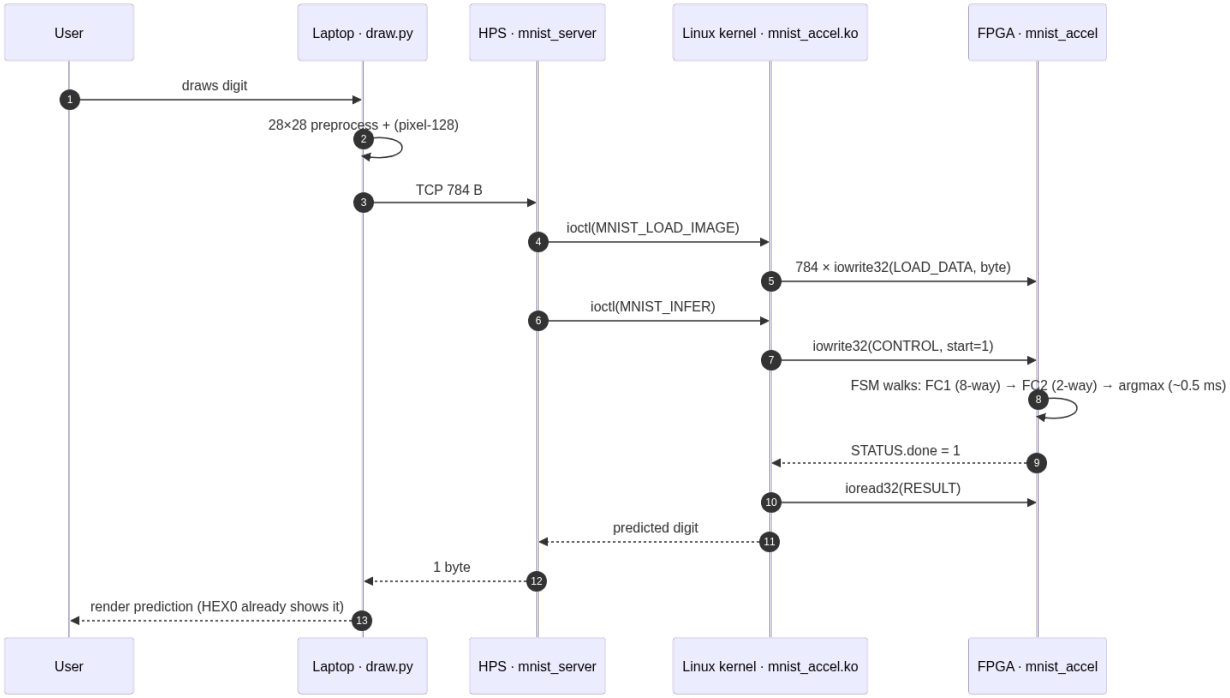
The point of the project is *not* speed (a regular CPU does this faster). The point is taking a trained neural network and turning it into **dedicated digital hardware** — the same exercise that goes into a Google TPU or Tesla FSD chip, just at hobbyist scale and at integer-only precision.

System at a glance



Half	What runs there	Files
Laptop	Tkinter drawing app, batch validation client	`project/sw/draw.py`, `project/sw/test_corpus.py`, `project/sw/plot_eval.py`
HPS	TCP server, Linux kernel driver, interactive picker	`project/sw/mnist_server.c`, `project/sw/mnist_accel.c`, `project/sw/mnist_accel.h`, `project/sw/mnist_pick.c`, `project/sw/mnist_run.c`
FPGA fabric	Custom SystemVerilog peripheral	`project/hw/mnist_accel.sv`, `project/hw/hex7seg.sv`, `project/hw/mnist_accel_hw.tcl`

Per-inference flow



End-to-end interactive latency is about 1.4 ms per digit; ~ 0.5 ms is the FPGA, the rest is TCP and Python.

The algorithm

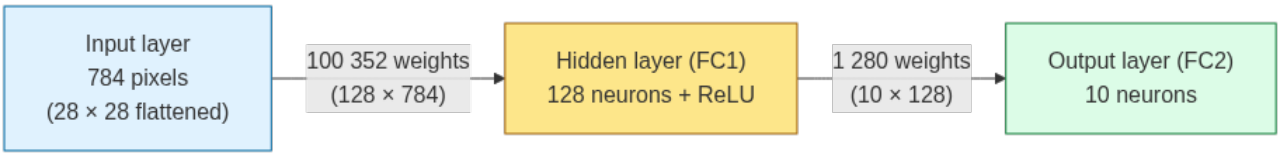
Neural-network primer

The network is just a stack of matrix-vector multiplications interleaved with element-wise non-linearities. One *neuron* with `n` inputs computes

```
output = ReLU( wûx + wûx +  + wûx + bias )
```

That is the entirety of the math: multiply each input by its weight, sum them, add a bias, clamp negatives to zero with `ReLU(x) = max(0, x)`. A *layer* is just many such neurons in parallel, each with its own weight vector.

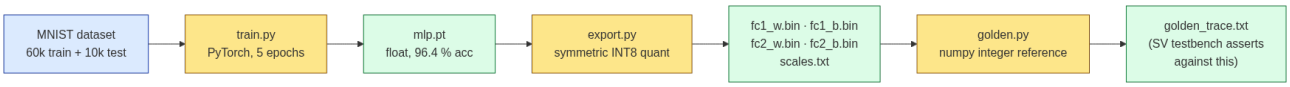
Our network shape



Layer	Inputs	Outputs	Weights	Biases
FC1	784	128	100 352 (INT8)	128 (INT32)
FC2	128	10	1 280 (INT8)	10 (INT32)

Inference is $784 \times 128 = 100\ 352$ multiply-adds in FC1, plus $128 \times 10 = 1\ 280$ in FC2, plus a 10-way argmax. The classifier prediction is `argmax(out, ..., out)`.

Training pipeline (offline, on the laptop)



The training pipeline lives entirely in `project/train/`:

- **'project/train/train.py'** — 5-epoch PyTorch run on the canonical MNIST split, hits 96.4 % test accuracy. Saves `mlp.pt` (float weights, ~410 kB).
- **'project/train/export.py'** — loads `mlp.pt`, applies symmetric per-tensor INT8 quantisation (formula below), writes the four `.bin` weight blobs plus `weights/scales.txt`.
- **'project/train/golden.py'** — pure-numpy integer forward pass; reads the same `.bin` blobs and produces a per-accumulator trace that the SystemVerilog testbench asserts against (139 internal values per image).
- **'project/train/dump_test_image.py'** — pulls ten sample MNIST test images and writes each as a 784-byte INT8 file (`img_0.bin` ... `img_9.bin`).

How `train.py` actually learns the weights

The model has 101 770 parameters in total — 100 352 weights in `fc1`, 128 biases there, 1 280 weights in `fc2`, 10 biases there. We do **not** pick those numbers ourselves. The training algorithm finds them by trial and error.

Pseudocode of the entire training procedure:

```
# ---- one-time setup ----
initialize all 101,770 parameters to small random values
load 60,000 training images (each 28x28 grayscale + correct-digit label)
choose:   batch_size = 128
          learning_rate = 0.001
          num_epochs = 5

# ---- the training loop ----
for epoch in 1..5:                          # 5 full passes through the data
    shuffle the 60,000 training images
    for each batch of 128 images:           # ~= 469 batches per epoch

        1. RESET    w.grad = 0  for every parameter
        2. FORWARD  scores = model(batch_images)            # 128 x 10
                    loss   = cross_entropy(scores, labels)  # one number
        3. BACKWARD loss.backward()                         # chain rule
                                                            # fills w.grad
                                                            # for every w

        4. UPDATE   w = w - learning_rate * w.grad

    print test-set accuracy

save weights to mlp.pt
```

The five steps explained:

1. **Initialise all 101 770 parameters to random small values.** PyTorch does this automatically when `nn.Linear(...)` constructs each layer. 2. **Forward pass — predict.** Show the model one batch of 128 labelled MNIST images. Every image goes through the multiply-add-ReLU formula, producing 10 output scores per image. Then compute the **cross-entropy loss**: one number that is small when the correct-digit score is much higher than the others, large when the model was confidently wrong. The batch loss is the average of the per-image losses (one scalar summarising 128 images' wrongness). 3. **Backward pass — compute the gradient.** For each of the 101 770 parameters, we want to know "if I increase this weight by an infinitesimal , how much does loss change?" That direction-and-rate-of-change is called the **gradient**. Loss depends on `scores`, which depend on `hidden`, which depend on the weights — by the chain rule, the gradient of loss with respect to any specific weight is the product of the local derivatives along that dependency chain. **Backpropagation** walks the chain *backward* starting from loss, multiplying local derivatives as it goes, and computes all 101 770 gradients in a single sweep. PyTorch tracks the operations performed during the forward pass (the "computation graph") and automatically does the backward sweep when you call `loss.backward()`. After it returns, every parameter `w` has its `w.grad` field filled in. 4. **Update — take a small step downhill.** For each parameter: `w` ← `w` `learning_rate` × `w.grad`. If the gradient is positive, increasing `w` would *increase* loss, so we *decrease* `w`; if negative, vice versa. The `learning_rate = 1e-3` is the step size; tiny on purpose so the optimisation is stable (big steps overshoot and oscillate). PyTorch's `Adam` optimiser refines this with per-parameter adaptive step sizes, but the basic shape is the same: each weight moves a fraction of a percent of its magnitude per batch. 5. **Reset ('w.grad = 0') and repeat** for the next batch. Without the reset, each batch's gradient would accumulate on top of the previous batch's `.grad`, the optimiser would see a corrupt running sum, and training would silently fail.

After all 60 000 training images pass through (= ~469 batches at 128 each), that's one **epoch**. We run **5 epochs**, for a total of about **2 345 weight updates**. Each epoch takes ~30 s on a CPU; the whole training run takes ~2-3 minutes.

The actual loss + accuracy trajectory of our run:

Epoch	Train loss	Test accuracy
1	0.39	92.5 %
2	0.20	94.8 %
3	0.15	95.7 %
4	0.13	96.2 %
5	0.12	**96.4 %**

We chose 5 epochs by *watching* this curve — beyond epoch 5 the test accuracy plateaus, and after about epoch 7 the model starts *overfitting* (test accuracy actually decreases as the model memorises training-set quirks instead of learning general digit patterns). 5 is the smallest number of epochs that hits the plateau, so we stop there.

‘opt.zero_grad()‘ and why it's needed. PyTorch's gradient field on each parameter (`.grad`) is *cumulative* — every call to `loss.backward()` *adds* to it. For ordinary training we want each batch to compute a fresh gradient, so we zero out `.grad` at the start of every iteration. If you forget, the second batch's gradient gets added to the first batch's, then the third gets added to those two, and so on; the network sees a "running sum of all previous batches" instead of just the current one, and training fails silently.

The model does not decide when to stop. *We* hard-code `EPOCHS = 5`. There exist smarter automated stopping rules (early stopping on a validation set, learning-rate scheduling) but we did not use them — at the scale of 101 770 parameters and 60 000 images, training converges fast enough that a fixed 5-epoch budget is sufficient.

Pixel range during training. Every pixel the network sees in `train.py` is a `float32` in `[1.0, +0.99...)`, centred around zero. The transform is `(pixel_uint8 - 128) / 128.0`. This pre-fixes the *training-side input contract* that the FPGA will later mirror as `x_int8 = pixel - 128` — both forms are centred around zero, differing only by a constant scale factor of 128 (see ğ2.6 on quantisation for why that constant doesn't change the final prediction).

Quantization: how the conversion works

Inference on the FPGA is integer-only. **There are two separate operations** that move us from float training to integer inference, and it's worth distinguishing them up front:

- **Weight quantisation**: `W_int8 = round( W_float / s_w1 )` — turns float weights into signed INT8. **Some information is lost** (rounding error per weight, at most `s_w1 / 2` ≈ `0.0024`), but the argument later in this section shows the loss doesn't propagate into wrong predictions.
- **Pixel shift**: `x_int8 = pixel_uint8  128` — turns uint8 pixels into signed INT8. **No information is lost** — it's just a constant integer subtraction with no rounding. The pixel `0` becomes `128`, the pixel `255` becomes `+127`, and every value in between maps one-to-one.

Both operations produce signed INT8 in `[128, +127]`. That's deliberate: the FPGA's MAC unit is a single `signed` × `signed` multiplier (`$signed(fc1_w_rdata[b]) * $signed(img_rdata)`), and feeding it inputs of the same signed type for both operands is the simplest and cheapest hardware. Both operations also re-centre their respective data around zero, matching the distribution PyTorch trained the network on.

The rest of this section walks through the lossy half — weight quantisation — on real weights from our trained model.

The INT8 range

A signed 8-bit integer is one byte that can hold exactly **256 different values**, evenly spaced:

```
-128  -127  -126  ...  -2   -1   0   +1   +2  ...  +125  +126  +127

lowest                                                 highest
```

No decimals. No fractions. 256 whole-number "buckets." Anything we want to store must round to one of these.

The conversion formula

We pick a *scale factor* per weight tensor so that the largest weight in the tensor maps exactly to 127:

```
s_w1  =  max(|W_float|) / 127
      =  0.608 / 127
      =  0.004785351866          (from weights/scales.txt)

W_int8  =  round( W_float / s_w1 )
```

`W_float / s_w1` answers "how many `s_w1`-sized steps fit into this weight?", and rounding picks the nearest INT8 bucket.

Worked example

Eight real weights from our trained `fc1.weight` tensor:

Float weight	ø `0.004785`	rounded	Stored as INT8
0.608 (largest)	127.06	127	the maximum positive bucket
0.300	62.69	63	
0.043562	9.10	9	
0.005217	1.09	1	
0.000843	0.18	0	rounded down to zero
0.018324	3.83	4	
0.231715	48.42	48	
0.608 (smallest)	127.06	127	the maximum negative bucket

That's the conversion in full. Every float weight becomes one of the 256 buckets.

What rounding costs us, in absolute terms

After the round step the original float is gone. We can reconstruct an approximation by multiplying the INT8 back by the scale:

Original float	INT8	Reconstructed float	Error
0.005217	1	0.004785	0.0004
0.000843	0	0.000000	0.0008
0.018324	4	0.019141	0.0008
0.043562	9	0.043068	0.0005
0.608	127	0.607740	0.0003

Every weight is off by **at most ‘s_w1 / 2 ≈ 0.0024‘** — half a bucket, the worst that rounding can ever do.

Why this doesn't ruin the network's accuracy

The per-weight error is on the order of 5 % of a typical weight (typical weights sit around 0.05 in absolute value; ś0.0024 of error is roughly 5 % of that). Sounds bad in isolation. But three things rescue us:

1. **The decision is "which output is biggest"**, not "what's the exact value of the biggest." Small wobbles in each score don't flip the winner. 2. **Each output is a sum of 784 such weighted products** (or 128 for FC2). When we add 784 roughly-random ś5 % errors, statistics says the *combined* error grows like `784 ≈ 28`, not 784×. So the output noise is about `5 % × 28 / 784 ≈ 0.18 %` of the signal. 3. **The decision margin** — the gap between the winning class score and the runner-up — is typically 20-50 % of the signal magnitude on a well-trained MNIST classifier. **The quantisation noise is two orders of magnitude smaller than the decision margin.**

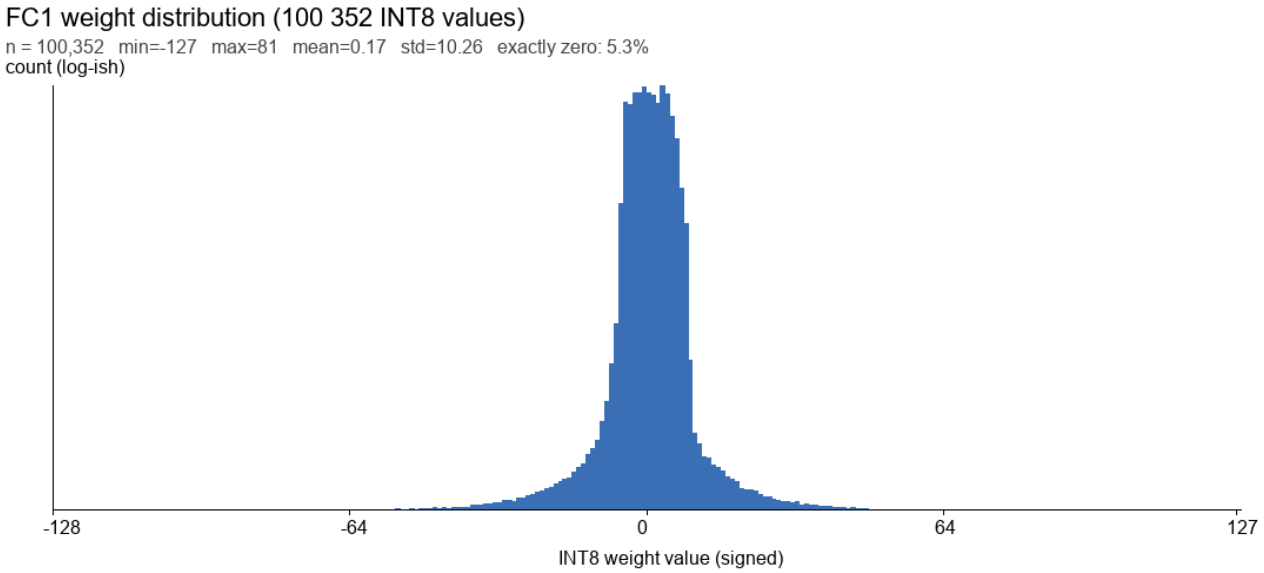
End result: the float model and the INT8 hardware classifier agree on virtually every image. Float test accuracy was 96.4 %; the same model running on the FPGA scored 96.53 % on the 10 000-image test set (ğ5.2). Quantisation cost us nothing measurable.

The full set of scale factors used in the design

From `project/weights/scales.txt`:

Tensor	Scale (float per INT8 step)	Max float value	Decimal precision per step
input pixel	`s_x = 1/128 = 0.0078125 (exact)`	± 1.0	≈ 0.008
FC1 weights	`s_w1 = 0.004785351866`	$127 \cdot s_{w1} \approx 0.608$	≈ 0.0048
FC1 hidden (implicit)	`s_h = s_x \{}cdot s_{w1} = 3.74 \{}times 10^{-5}`	INT32-bounded	$\approx 3.7 \times 10^{-5}$
FC2 weights	`s_w2 = 0.004158100744`	$127 \cdot s_{w2} \approx 0.528$	≈ 0.0042

And the actual FC1 weight distribution after the conversion:

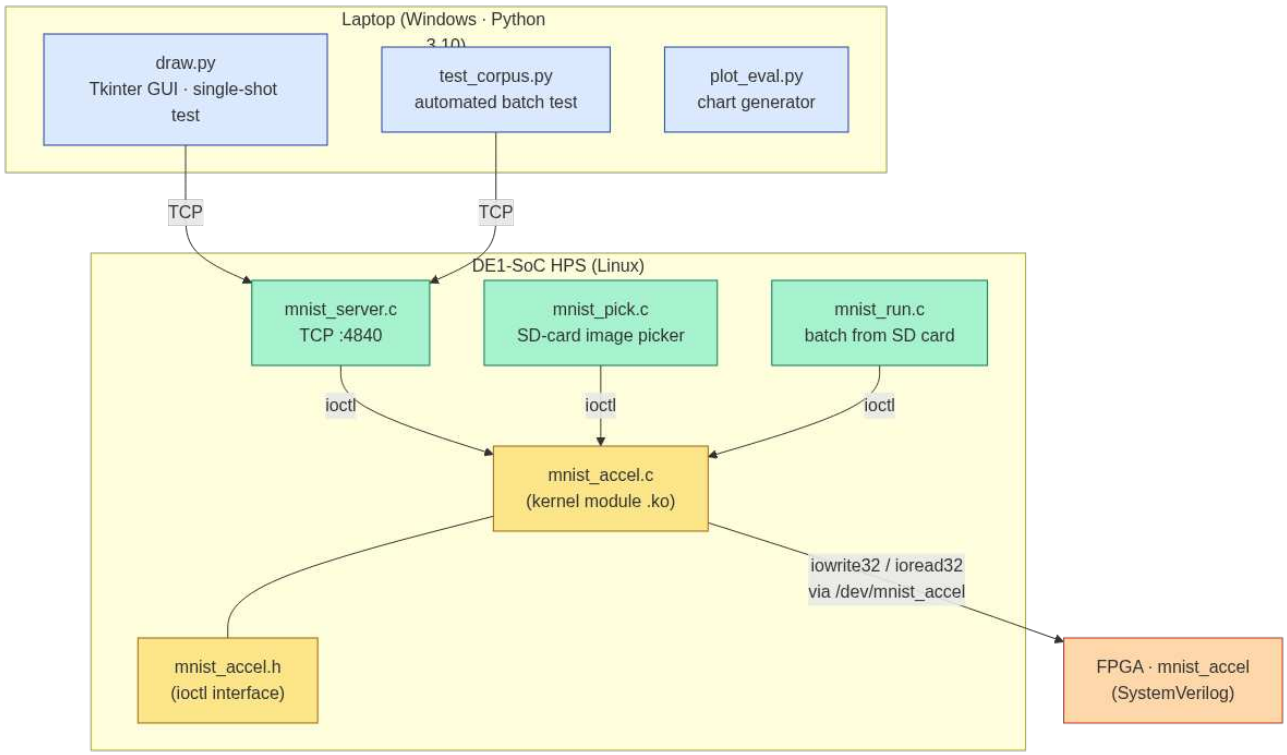


5.3 % of weights round to exactly zero — connections the network learned to ignore. Most live in $[-30, +30]$ in INT8 units. Only the heavy tail near ± 127 matters for the scale choice.

Software architecture

The software lives on two machines — a Windows laptop running Python clients, and the DE1-SoC's ARM Cortex-A9 (HPS) running Linux with our kernel module + userspace daemon.

Software stack



The kernel driver `mnist_accel.c` is the single bottleneck through which every byte that touches the FPGA must pass. All three userspace programs (`mnist_server`, `mnist_pick`, `mnist_run`) speak to it through the same seven `ioctl()` calls defined in `mnist_accel.h`.

Laptop · `project/sw/draw.py`

A Tkinter window with a hi-res Pillow canvas. The user draws a digit with the mouse; the script does MNIST-style preprocessing and pushes 784 bytes to the server over TCP.

Why preprocessing is necessary

A raw mouse drawing is *not* an MNIST-distribution image. It's:

- 280×280 pixels, not 28×28.
- Possibly off-centre — drawn wherever the user clicked.
- Possibly too large (filling the canvas edge-to-edge) or too small (5 pixels of ink in the middle).
- Blocky-edged at the cell boundaries of the Tkinter grid.

The trained network only understands inputs that look like the MNIST distribution — centred on a 28×28 black canvas with the digit occupying at most a 20×20 area, with anti-aliased grey edges. We have to reshape every fresh drawing into that exact format before sending it.

Why 28×28 if the digit is 20×20

The MNIST format defines two different things:

- The **image** is 28×28. Every byte in the file is one of those 784 pixels.
- The **digit inside the image** has a longest dimension of at most 20 pixels, with a 4-pixel black margin on every side.

The 4-pixel margin is intentional. The network was trained with this margin always present — its weights expect that outer rows and columns are usually black. If a digit ever ran to the edge of the 28×28 frame, the model would see ink in pixel positions it has never been trained on, and predictions would degrade.

The five preprocessing steps in `draw.py:_to_model_bytes()`

We use the `Pillow` library (`PIL`) to do these steps in pure Python — we do *not* link against any MNIST-authors library. The original LeCun 1998 dataset was preprocessed once by its authors and shipped to us already-baked; we are reproducing those same operations on fresh user-drawn images.

Step	Code	What it does
1	`self.img.getbbox()`	Find the smallest rectangle that contains all the white ink on the 280×280 canvas. If you drew a small digit in the corner, this rectangle is small and in the corner.
2	`digit = self.img.crop(bbox)`	Throw away the black space outside that rectangle. Now we have just the digit, no surrounding emptiness.
3	`digit.resize((new_w, new_h), Image.LANCZOS)`	Shrink (or stretch) the digit so its longest side becomes exactly 20 pixels, preserving its aspect ratio. `LANCZOS` is Pillow's high-quality anti-aliased resize filter, so the edges come out smooth grey rather than blocky.
4	`canvas = Image.new("L", (28, 28), 0); canvas.paste(digit, (px, py))`	Make a fresh 28×28 black canvas. Compute the digit's centre of mass and choose `(px, py)` so the centre of mass lands on pixel `(14, 14)` — the centre of the 28×28 canvas. Paste it there.
5	`bytes((p - 128) & 0xff for p in canvas.tobytes())`	Shift every pixel from `uint8 [0, 255]` to signed `int8 [-128, +127]` and pack as a single byte each. This is exactly the format the FPGA's `LOAD_DATA` register expects.

After Step 4, the image is **structurally identical to an MNIST test image** — 28×28, centred by centre of mass, longest side 20 pixels, anti-aliased edges. Step 5 then converts the bytes from the file-storage format (`uint8`) to the wire format (`int8`) that the rest of the pipeline (server → driver → FPGA) consumes.

The hi-res 280×280 drawing buffer (`HI_RES = 280`, code constant in `draw.py`) means strokes look smooth and marker-like to the user, but only the post-resize 28×28 image is ever sent to the FPGA. The preprocessing matches what `dump_test_image.py` does on real MNIST images, and what `golden.py` does on its integer reference pass — so the FPGA receives bytes formatted *identically* to what it was trained on.

Laptop · `project/sw/test_corpus.py`

Automated hardware-in-the-loop validation. Iterates the canonical MNIST test set (10 000 labelled images, never seen during training); for each one:

```
payload = bytes((p - 128) & 0xff for p in pil_img.tobytes())
with socket.socket(AF_INET, SOCK_STREAM) as s:
    s.connect((FPGA_IP, 4840))
    s.sendall(payload)
    pred = s.recv(1)[0]
confusion[true_label][pred] += 1
```

Writes three deliverables under `project/eval/`:

- `test10k_2026-05-11_report.txt` — human-readable summary
- `test10k_2026-05-11_results.csv` — per-image rows: `idx, split, true, predicted, correct`
- `misclassified/test_idxNNNNN_trueX_predY.png` — every wrong image saved as a PNG for analysis

Section 5 below reports the full results from the 10 000-image run.

Laptop · `project/sw/plot_eval.py`

Pure-Pillow chart generator (no matplotlib dependency). Reads `results.csv` and `weights/*.bin`, emits four PNGs into `project/eval/`:

- `accuracy_bars.png` — per-digit accuracy histogram
- `confusion_heatmap.png` — 10 × 10 confusion matrix colour-coded by intensity
- `weights_hist_fc1.png`, `weights_hist_fc2.png` — INT8 weight distribution histograms

HPS · `project/sw/mnist_server.c`

A small TCP daemon. On startup it opens `/dev/mnist_accel`, streams the four weight blobs through `ioctl(MNIST_LOAD_FC1_W`, then listens on port 4840. For each accepted connection, it loops:

```
recv(clntsock, image_buffer, 784, ...);              // 784 B from network
ioctl(fpga_fd, MNIST_LOAD_IMAGE, ...);                // push to FPGA's img_mem
ioctl(fpga_fd, MNIST_INFER, &result);                 // start + poll + read result
send(clntsock, &result.digit, 1, 0);                  // 1-byte response
```

The 100 ms timeout in the driver's `MNIST_INFER` ioctl (vs the 0.5 ms typical inference time) means any wedged FSM or unresponsive bus is caught quickly rather than hanging the connection.

HPS · `project/sw/mnist_pick.c`, `mnist_run.c` (alternatives)

Two siblings of `mnist_server` that don't talk to the network at all — useful for local-only testing on the board:

- **‘mnist_pick.c‘** — interactive picker. Prompts at the console for a digit 0–9 (or `'a'` for all ten, `'q'` to quit), loads the corresponding `weights/img_N.bin` from the SD card, runs one inference, prints the result. Used during initial bring-up before the TCP path existed.
- **‘mnist_run.c‘** — non-interactive batch tester that runs all ten sample images and reports per-image accuracy. Useful for smoke-testing the SD card and driver path with no network involved.

Both of these were the workhorses for early hardware bring-up; `mnist_server` is the production path used for the live drawing demo.

HPS · `project/sw/mnist_accel.c` (kernel driver)

A Linux platform-driver/miscdevice pair (~360 lines). The platform-driver side matches the device-tree node and obtains the bus base address via `of_iomap`; the miscdevice side exposes `/dev/mnist_accel` and dispatches seven `ioctl()` commands defined in `project/sw/mnist_accel.h`:

```
MNIST_LOAD_FC1_W   _IOW('m', 1, mnist_buf_t)
MNIST_LOAD_FC1_B   _IOW('m', 2, mnist_buf_t)
MNIST_LOAD_FC2_W   _IOW('m', 3, mnist_buf_t)
MNIST_LOAD_FC2_B   _IOW('m', 4, mnist_buf_t)
MNIST_LOAD_IMAGE   _IOW('m', 5, mnist_buf_t)
MNIST_INFER        _IOR('m', 6, mnist_result_t)
MNIST_GET_RESULT   _IOR('m', 7, mnist_result_t)
```

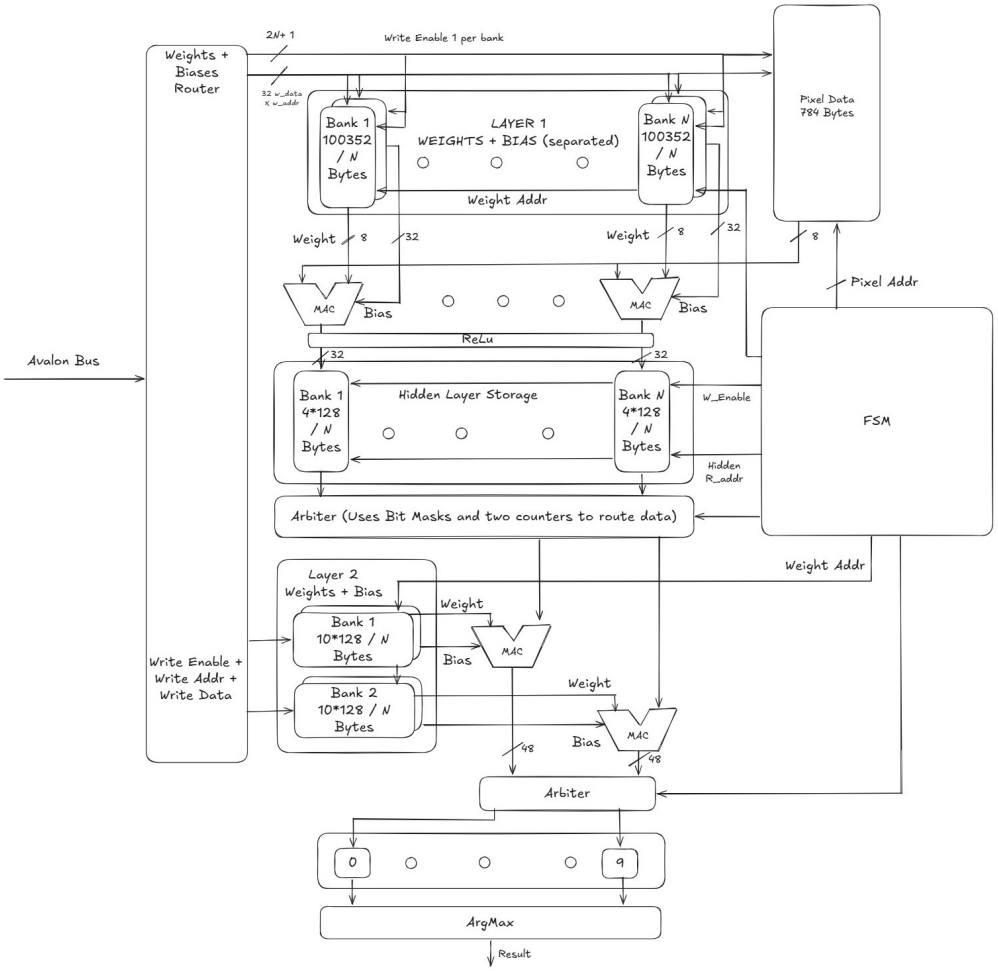
Each `MNIST_LOAD_*` ioctl copies the buffer in from user-space, then loops `iowrite32()`-ing one byte per write into the FPGA's `LOAD_DATA` register; the hardware demux (Section 4) figures out which BRAM bank each byte belongs to.

`MNIST_INFER` writes `CONTROL = 0` (clears load_mode), writes `CONTROL = 1` (start), then polls `STATUS.done` with `udelay(100)` between reads, timing out at 100 ms if the FSM doesn't finish. On success it reads `RESULT[3:0]` and `CONFIDENCE[31:0]` and returns them to user-space as `mnist_result_t`.

Hardware architecture

This is the heart of the project. The entire FPGA logic is in **'project/hw/mnist_accel.sv'** (~380 lines); the only other RTL file is **'project/hw/hex7seg.sv'** (a 16-line 4-to-7-segment ROM from Lab 1, used unchanged).

Top-level hand-drawn block diagram



Editable source: <https://excalidraw.com/#json=b0_C1xVcVGlzJ1NlJmrzn,VjF53zlqBiEnRJ79-cxWzQ>

Reading the diagram. Avalon bus enters on the left → splits into a **Weights+Biases router** (top) and a **Write Enable / Write Addr / Write Data fan-out** (bottom-left). The router populates **Layer 1's $N=8$ weight+bias banks** (centre-left) and the **pixel-data BRAM** (top-right). During inference the **FSM** (right) drives addresses into the BRAMs; each bank's MAC produces an INT32 partial that goes through **ReLU** and is stored in the **hidden-layer banks**. An **arbiter** (centre) gathers hidden values via the bit-mask + counter trick so that **Layer 2's 2 banks** see them in flat order. Two more MACs feed **out_regs[0..9]**, an **ArgMax** picks the winner, the result is latched and drives HEX0.

The seven subsystems are detailed individually below, in roughly top-to-bottom order around the diagram.

Module parameters

From `project/hw/mnist_accel.sv:33-58`:

```
localparam int FC1_IN  = 784;   localparam int FC1_OUT = 128;
localparam int FC2_IN  = 128;   localparam int FC2_OUT = 10;
parameter  int FC1_BANKS = 64;  parameter  int FC2_BANKS = 2;

localparam int FC1_B_PER_BANK = FC1_OUT / FC1_BANKS;       //   2
localparam int FC1_W_PER_BANK = FC1_B_PER_BANK * FC1_IN;   // 1568
localparam int FC2_B_PER_BANK = FC2_OUT / FC2_BANKS;       //   5
localparam int FC2_W_PER_BANK = FC2_B_PER_BANK * FC2_IN;   //  640
```

64-way parallelism in FC1 and 2-way in FC2 are the design's central trade-off: FC1 finishes in two outer iterations (128 / 64) instead of 16, dropping the FC1 phase from ~25 000 cycles to ~3 000 cycles (~60 µs at 50 MHz). The bank count is a parameter — we also synthesised a 128-way variant which finishes FC1 in ~30 µs.

Avalon-MM slave + register file

The peripheral exposes itself to the HPS as eight 32-bit registers mapped at `0xFF200040`:

Offset	Name	Access	Bit layout
`0x00`	CONTROL	W	`[2] clear_done` · `[1] load_mode` · `[0] start`
`0x04`	STATUS	R	`[1] done (sticky)` · `[0] busy`
`0x08`	RESULT	R	`[3:0] predicted digit`
`0x0C`	CONFIDENCE	R	`best_val[31:0]` (low 32 bits of winning INT48 accumulator)
`0x10`	LOAD_ADDR	W	Writing any value resets the router's `load_j` / `load_i` counters
`0x14`	LOAD_DATA	W	One write = one byte streamed into the currently selected BRAM target
`0x18`	LOAD_TARGET	W	`[2:0] target` (0=FC1_W, 1=FC1_B, 2=FC2_W, 3=FC2_B, 4=IMG)
`0x1C`	VERSION	R	constant `0x4D4E5301` ("MNS\{}x01") — sanity-read at driver probe

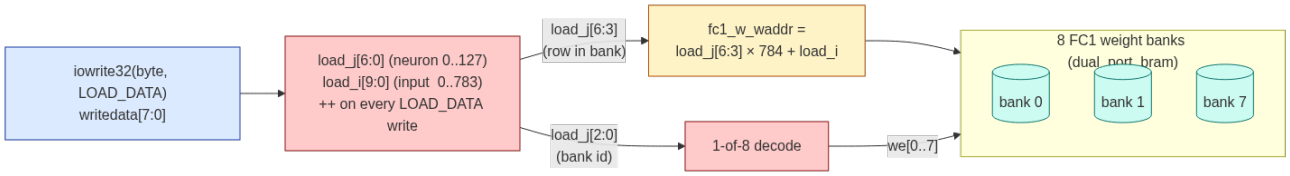
The Avalon frontend registers all bus inputs on the rising clock edge (`mnist_accel.sv:108-112`):

```
always_ff @(posedge clk) begin
    avalon_write_q <= chipselect && write;
    avalon_addr_q  <= address;
    avalon_wdata_q <= writedata;
end
```

This single flop stage gives the downstream logic a stable, glitch-free view of bus traffic, and is also why `load_data_write = avalon_write_q && (avalon_addr_q == 4'd5)` lands cleanly on the next cycle.

Smart router (load path)

When the driver streams weight bytes through `LOAD_DATA`, the router has to decide **which of the N banks** receives the byte and **what address within that bank** to write it to. The trick is to use the low bits of the neuron counter `load_j` as a 1-of-8 bank select, and the high bits as the row index inside each bank.



Target	Bank-select bits	Row-select bits	# banks
FC1 weights	`load_j[5:0]` → `we[0..63]`	`load_j[6]` → row in `waddr`	64
FC1 biases	`load_i[5:0]` → `we[0..63]`	`load_i[6]` → row in `waddr`	64
FC2 weights	`load_j[0]` → `we[0..1]`	`load_j[3:1]` → row in `waddr`	2
FC2 biases	`load_i[0]` → `we[0..1]`	`load_i[3:1]` → row in `waddr`	2
Image	(single BRAM — no banking)	`load_i[9:0]` is `img_waddr`	1

The entire router is a `case` statement plus a 1-hot decoder (`mnist_accel.sv:188-213`), costing ~50 ALMs. Software stays trivial — just a tight `for` loop pushing one byte per iteration into `LOAD_DATA`.

BRAM banks (the `dual_port_bram` primitive)

Every memory in the design is an instance of a small wrapper module that forces Quartus to pack it into M10K block memory:

```
module dual_port_bram #(int DATA_WIDTH, int ADDR_WIDTH, int DEPTH) (
    input  logic clk,
    input  logic we,
    input  logic [ADDR_WIDTH-1:0] waddr,
    input  logic [ADDR_WIDTH-1:0] raddr,
    input  logic [DATA_WIDTH-1:0] wdata,
    output logic [DATA_WIDTH-1:0] rdata
);
    (* ramstyle = "M10K" *) logic [DATA_WIDTH-1:0] mem [0:DEPTH-1];
    always_ff @(posedge clk) begin
        if (we) mem[waddr] <= wdata;
        rdata <= mem[raddr];      // registered output: 1-cycle read latency
    end
endmodule
```

The registered output means a read takes **1 cycle from ‘raddr‘ change to valid ‘rdata‘** — the FSM hides this by inserting `*_MAC_WAIT` states that issue the next address while the datapath is consuming the current one. Without the `ramstyle` attribute Quartus occasionally inferred ALMs instead of M10K blocks, which blew up the logic count.

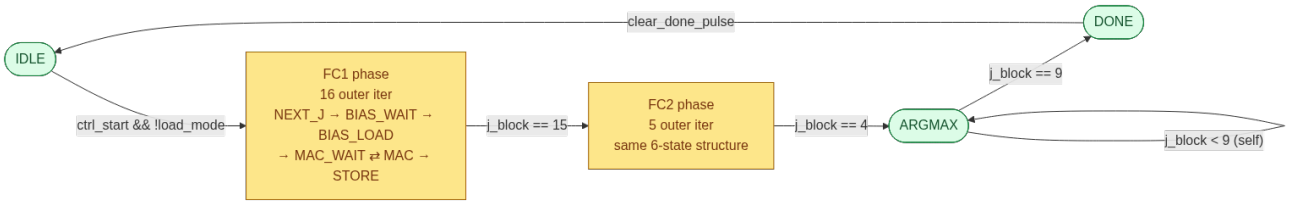
Use	DATA_WIDTH	ADDR_WIDTH	DEPTH	Instances
`fc1_w_mem[b]` (Layer 1 weights)	8	11	1 568	64
`fc1_b_mem[b]` (Layer 1 biases)	32	1	2	64
`h_ram[b]` (hidden activations)	32	1	2	64
`fc2_w_mem[b]` (Layer 2 weights)	8	10	640	2
`fc2_b_mem[b]` (Layer 2 biases)	32	3	5	2
`img_ram` (input image)	8	10	784	1

The FC1 weight memory is interleaved by neuron: bank `b` holds neurons whose index is congruent to `b` mod 64 (so bank 0 holds neurons 0 and 64; bank 1 holds 1 and 65; and so on). When the FSM holds `j_block = r`, all 64 banks simultaneously expose neurons `64r, 64r+1, ... 64r+63`.

FSM

The 15-state inference controller. Counters used: `j_block[7:0]` (outer loop, 0..15 for FC1, 0..4 for FC2, 0..9 for argmax), `i_idx[9:0]` (inner loop, 0..783 for FC1, 0..127 for FC2). The FSM is suspended whenever `load_mode == 1` (`mnist_accel.sv:261`).

Phase-level view (each FC phase hides a 6-state inner loop; the inner states are listed in the table that follows):



Inner-loop transitions (15 states total — these are the conditions hidden inside the FC1 and FC2 phase boxes above):

From state	To state	Condition
FC1_NEXT_J → FC1_BIAS_WAIT → FC1_BIAS_LOAD → FC1_MAC_WAIT → FC1_MAC	FC1_MAC_WAIT	`i_idx < 783`
FC1_MAC	FC1_STORE	`i_idx == 783`
FC1_STORE	FC1_NEXT_J	`j_block < 15`
FC1_STORE	FC2_NEXT_J	`j_block == 15`
FC2_NEXT_J → FC2_BIAS_WAIT → FC2_BIAS_LOAD → FC2_MAC_WAIT → FC2_MAC	FC2_MAC_WAIT	`i_idx < 127`
FC2_MAC	FC2_STORE	`i_idx == 127`
FC2_STORE	FC2_NEXT_J	`j_block < 4`
FC2_STORE	ARGMAX	`j_block == 4`

Cycle accounting:

Phase	Outer iterations	Inner cycles	Total cycles
FC1 (64-way parallel)	2	784 × 2 cyc/MAC	~3 000
FC2 (2-way parallel)	5	128 × 2 cyc/MAC	~1 300
Argmax (linear scan)	10	1	10
Misc transitions	—	—	~140
Total per inference			**~4 500 cyc ≈ 90 µs @ 50 MHz** (FC1 phase alone ≈ 60 µs)

Datapath — FC1 MACs (64-way parallel)

During `S_FC1_MAC` (`mnist_accel.sv:282-294`):

```
S_FC1_MAC: begin
    for (int b = 0; b < FC1_BANKS; b++) begin
        acc_fc1[b] <= acc_fc1[b]
                    + ($signed(fc1_w_rdata[b]) * $signed(img_rdata));
    end
    ...
end
```

64 signed 8 × 8 multiplies happen on every `S_FC1_MAC` cycle; each result is sign-extended to INT32 and added to the bank's own 32-bit accumulator. The image pixel is **shared** across all 64 banks (read once from `img_ram`, fanned out to 64 multipliers); the weights differ per bank.

Quartus packs the FC1 MAC array onto a small number of Cyclone V DSP slices (the fit report shows 8 DSPs total for the whole peripheral) plus ALM-based multipliers for the remaining lanes — the DSP block count never explodes because most lanes fit in logic at this bit width.

After the last `S_FC1_MAC` of an outer iteration, **'S__FC1__STORE'** writes the ReLU'd accumulator into the hidden BRAM:

```
for (int b = 0; b < FC1_BANKS; b++)
    hidden_wdata[b] <= acc_fc1[b][31] ? 32'd0 : acc_fc1[b];
hidden_we    <= 1'b1;
hidden_waddr <= j_block[0];
```

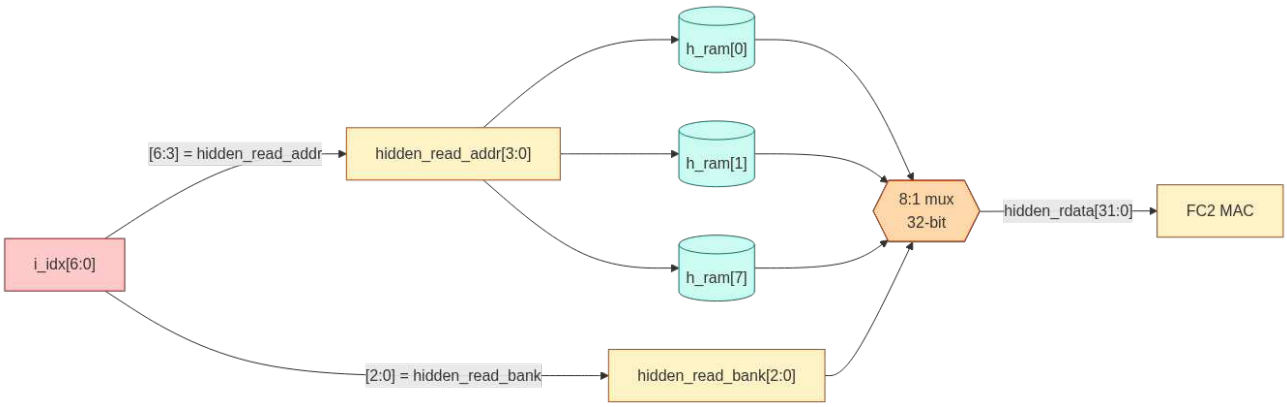
`acc_fc1[b][31]` is the sign bit of the INT32 accumulator — if it's set the value was negative, so ReLU outputs zero. The ReLU circuit is just a 32-bit 2:1 mux per bank.

Hidden-layer scatter / gather

FC1 *wrote* hidden values in interleaved order (bank `b` holds neuron `r·64 + b`). FC2 *reads* them in flat sequential order 0, 1, 2, … 127.

The 7-bit FC2 inner counter `i_idx[6:0]` is the same physical wires, but the FSM splits it into two fields:

```
i_idx[6:0] =     [6]          [5:0]
                 row        bank id
                 (1 bit,    (6 bits,
                  0..1)      0..63)
```



In SystemVerilog this is one line (`mnist_accel.sv:91`):

```
assign hidden_rdata = hidden_bank_rdata[hidden_read_bank];
```

A 32-bit 64:1 mux costs ~250 LUTs. All 64 hidden BRAMs read the same row in parallel; the mux at the output picks the correct bank's word.

Datapath — FC2 MACs (2-way parallel)

During `S_FC2_MAC` (`mnist_accel.sv:319-329`):

```
S_FC2_MAC: begin
    for (int b = 0; b < FC2_BANKS; b++)
        acc_fc2[b] <= acc_fc2[b]
                    + ($signed(hidden_rdata) * $signed(fc2_w_rdata[b]));
    ...
end
```

Two signed 32 × 8 multiplies (one per bank). Each multiplier maps to ~4 Cyclone V DSP slices because of the 32-bit operand — ~8 DSPs total for FC2, ~16 with the cascade structure. The hidden value `hidden_rdata` is shared between the two banks; the weights differ.

Bias-load for FC2 has to sign-extend the INT32 bias up to INT48 before placing it in the accumulator (`mnist_accel.sv:314`):

```
acc_fc2[b] <= {{16{fc2_b_rdata[b][31]}}, fc2_b_rdata[b]};
//             16 copies of MSB
```

Argmax + output flop

After all 10 FC2 outputs are stored in the `out_regs[0..9]` flip-flop array, `S_ARGMAX` spends 10 cycles scanning them with a strict-> comparator:

```
S_ARGMAX: begin
    if (out_regs[j_block[3:0]] > best_val) begin
        best_val <= out_regs[j_block[3:0]];   // signed INT48
        best_idx <= j_block[3:0];             // winning index, 0..9
    end
    if (j_block == FC2_OUT - 1) state <= S_DONE;
    else                        j_block <= j_block + 8'd1;
end
```

`best_val` is initialised to `48'sh800000000000` (most-negative INT48) so the first compared value always wins. On entering `S_DONE` the winning index is latched into `result[3:0]`, which is wired directly to the `hex7seg` decoder, which combinationally drives `HEX0[6:0]`. The HEX0 pin updates within combinational delay of `S_DONE` (a few ns) — far before software sees `STATUS.done`.

Validation

Three layers of validation

We caught bugs at three increasingly-realistic scales, each a prerequisite for the next:

Layer	Tool	What it proves	Where it runs
1	`train/golden.py`	The integer math (quantisation contract) is internally consistent and still classifies correctly	Laptop, pure NumPy
2	`tb/tb_mnist_accel.sv`	The RTL produces **bit-exact** the same INT32 / INT48 numbers as golden.py	Questa simulator
3	`sw/test_corpus.py`	End-to-end pipeline (preprocessing, TCP, driver, FPGA, FSM, argmax) classifies real MNIST at near-float accuracy	DE1-SoC + laptop, 10 000 images

Layers 1 and 2 are *upstream gates*: layer 3 is meaningless unless they are both green first.

Layer 1 — `golden.py` (laptop algorithmic floor)

`train/golden.py` is a re-implementation of the integer forward pass in pure NumPy. It loads the same `.bin` weight blobs that the FPGA loads, multiplies them out using `np.int32` / `np.int64`, applies ReLU, runs argmax, and prints the predicted digit. Every accumulator value is also dumped to `golden_trace.txt` — 139 internal values per image (128 hidden-layer INT32 sums + 10 output INT48 sums + the final argmax).

What it proves: that the quantisation recipe described in ğ2.4 — `s_x = 1/128`, `s_w = max|W|/127`, bias prescale, INT48 accumulator, no inter-layer requantisation — is internally consistent and still classifies MNIST correctly. If golden.py drops more than ~1 % accuracy versus the float PyTorch model, the math we wrote on paper is already wrong and there is no point continuing to RTL.

* *What it does* not *prove:** anything about the SystemVerilog. It only validates the algorithm.

Layer 2 — Questa testbench (RTL bit-exact)

`tb/tb_mnist_accel.sv` is a SystemVerilog testbench that drives the Avalon-MM slave through a small BFM, loads the same four weight blobs and a sample image, runs an inference, and at every FC1/FC2 accumulator snapshot diffs the value against the matching line in `golden_trace.txt`. **Zero mismatches required** — any non-zero diff is treated as a hard failure.

What it proves: the RTL produces the *exact same* INT32 and INT48 numbers as the numpy golden on a single image. This is the only layer that can isolate:

- sign-extension bugs (signed × signed casts on the multipliers),
- off-by-one MAC indexing (e.g. address `i` vs `i+1` on the first cycle of a new bank),
- BRAM read-latency drain bugs (the 2-cycle pipeline between `raddr` and `q`),
- argmax tie-break direction (we follow numpy's `argmax` rule — earliest index wins on a tie).

Each of these would show up as a non-zero accumulator diff at the exact cycle where the error happens, with the offending neuron index in the failure message.

* *What it does* not *prove:** anything about real timing, real I/O, real Linux, or whether the design fits the chip. It is a microscope, not a survey.

Layer 3 — `test_corpus.py` (hardware-in-the-loop end-to-end)

Once the bitstream is on the board and `mnist_server` is running, we validate end-to-end against real labelled data using `project/sw/test_corpus.py`:

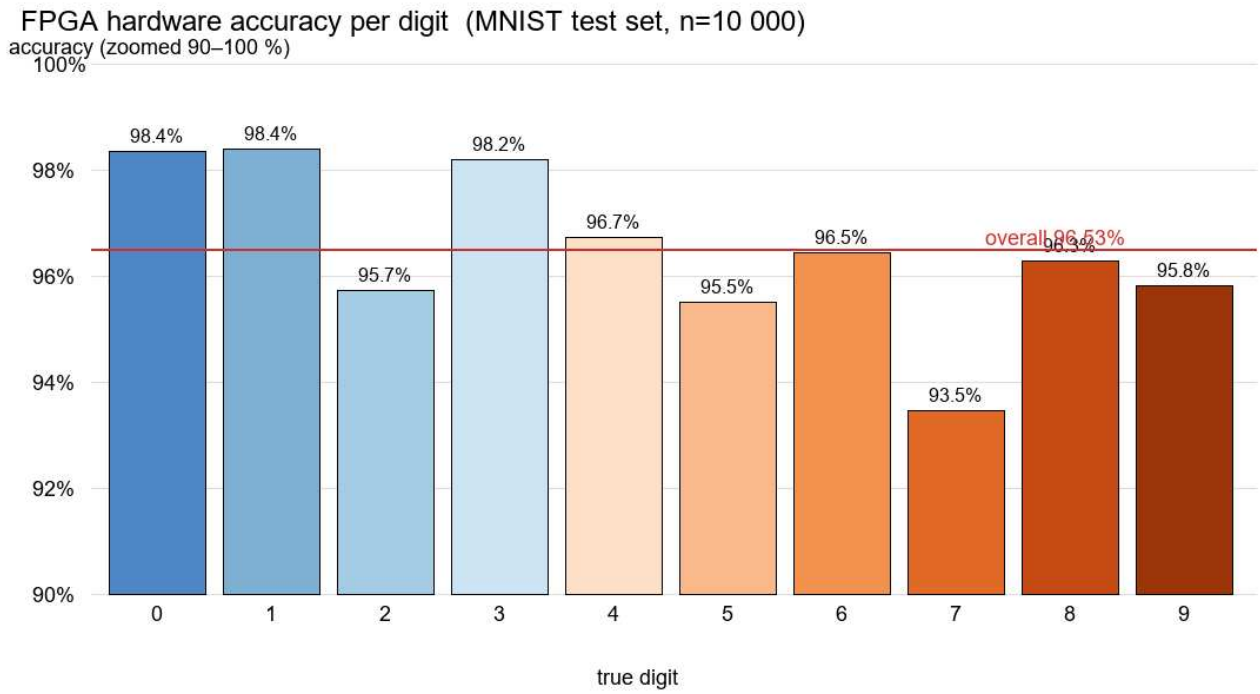
```
laptop (test_corpus.py)             DE1-SoC

for each of 10 000 MNIST            mnist_server
test images:                         |
  1. pixel - 128  INT8 byte      ioctl(LOAD_IMAGE)
  2. open TCP, send 784 bytes         ioctl(INFER)
  3. recv 1-byte prediction       STATUS.done ú RESULT[3:0]
  4. compare against ground-truth label
  5. record in CSV, update confusion matrix
```

The MNIST test set is the canonical "did we generalise?" benchmark — 10 000 images the network **never saw during training**. The 96.4 % figure quoted from PyTorch is measured on exactly this set, so it is the right reference number to compare hardware accuracy against.

The test is **end-to-end**: hardware integer math, the streaming-load path, the FSM, the polling, the TCP plumbing, the Linux kernel module, the Python client — all in the loop. A silent miscompute anywhere along the way would manifest as accuracy loss. This is the test whose result we lead with — the headline figure is its output.

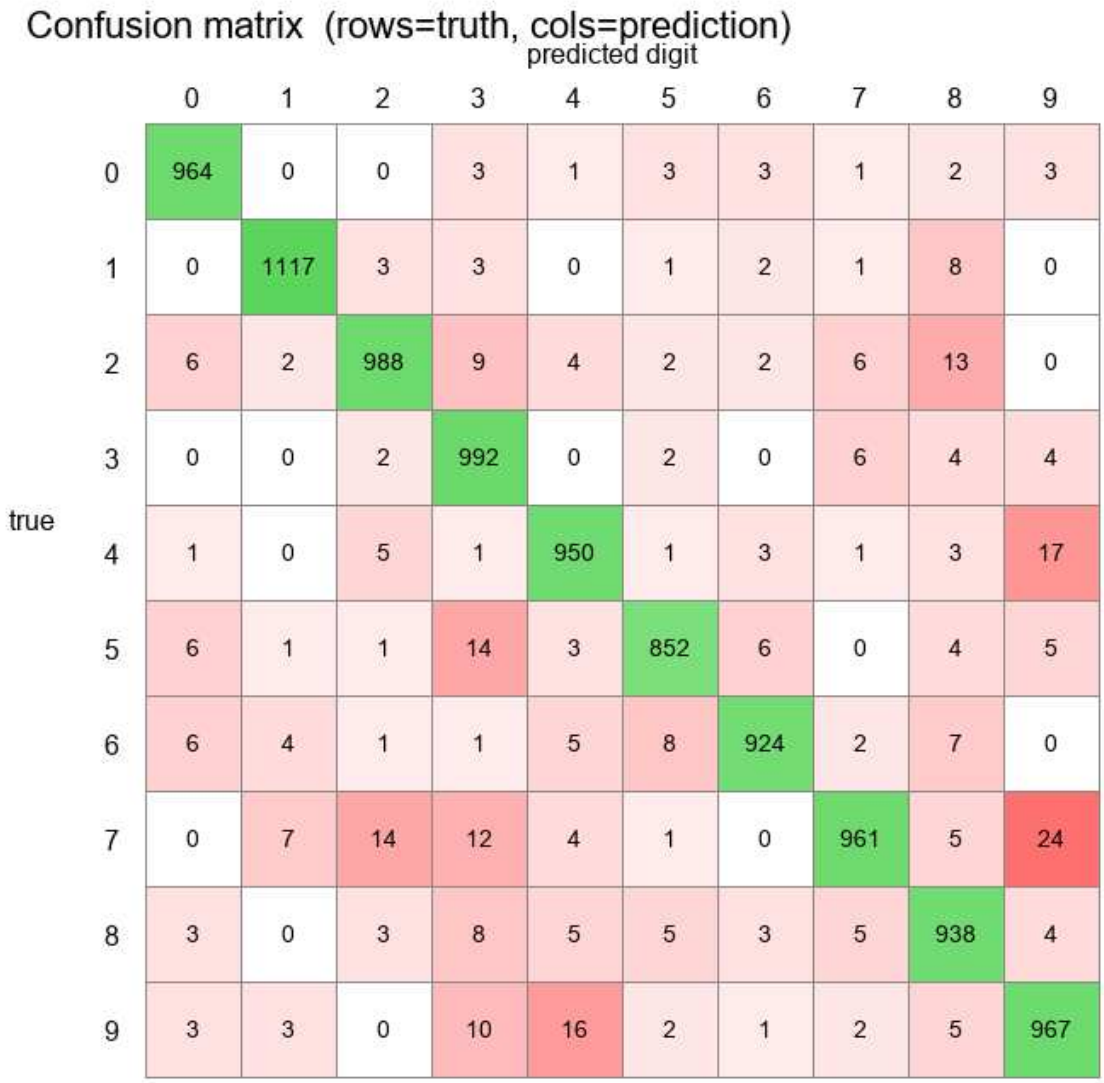
Headline result — 96.53 % on 10 000 images



Metric	Value
Overall accuracy	**9 653 / 10 000 = 96.53 %**
Wall-clock for full run	215.4 s (~46 inferences / s end-to-end)
Best digit	0 and 1 at 98.4 % each
Worst digit	7 at 93.5 %
Network failures	0 / 10 000

The hardware accuracy is essentially identical to the float baseline of 96.4 % — the small over-shoot is normal statistical noise for n = 10 000. The INT8 quantisation cost us nothing measurable.

Failure modes — confusion matrix



The biggest off-diagonal cells:

- `7` → `9` (24 cases) and `7` → `2` (14 cases): slanted 7s look like 9s, hooked 7s look like 2s.
- `9` → `4` (16 cases): tight loops on 9s confused with the diagonal stroke of 4.
- `5` → `3` (14 cases): round 5s vs round 3s.

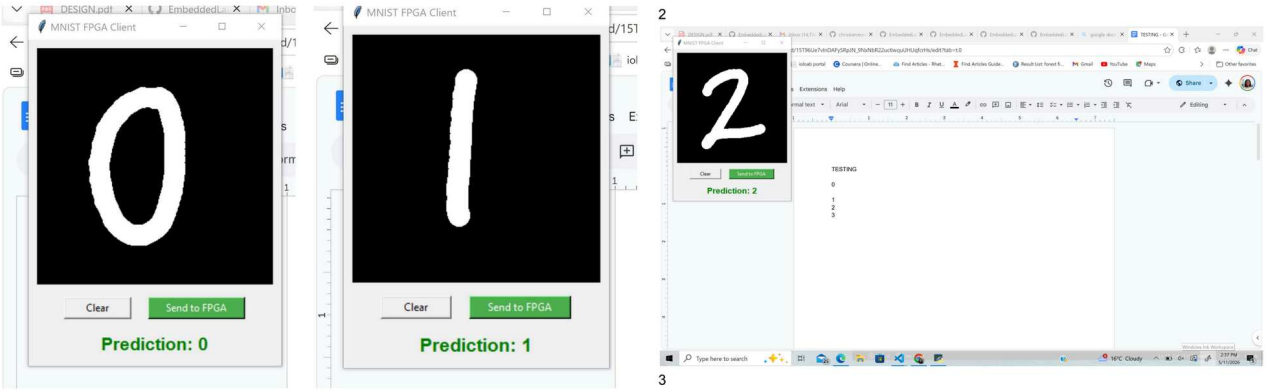
These are the same confusions a 96 %-accurate float MLP would make — capacity limits of the 128-hidden-neuron architecture, not artefacts of the hardware. The hardware faithfully reproduces the float network's decisions.

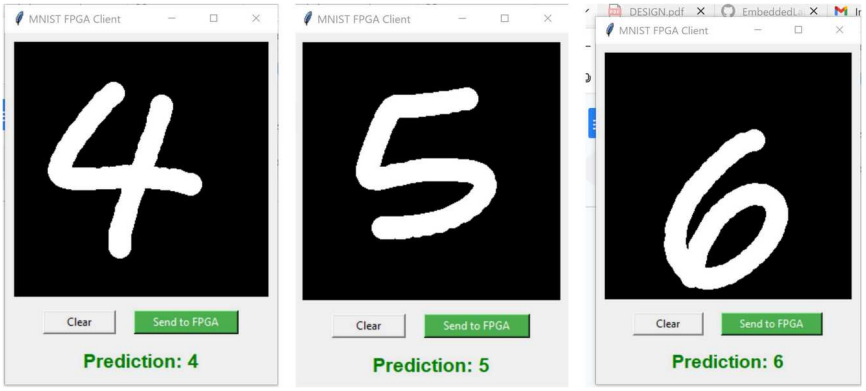
The 347 individual misclassified images are preserved as PNGs in `project/eval/misclassified/`, with filenames encoding the dataset index, true label, and predicted label.

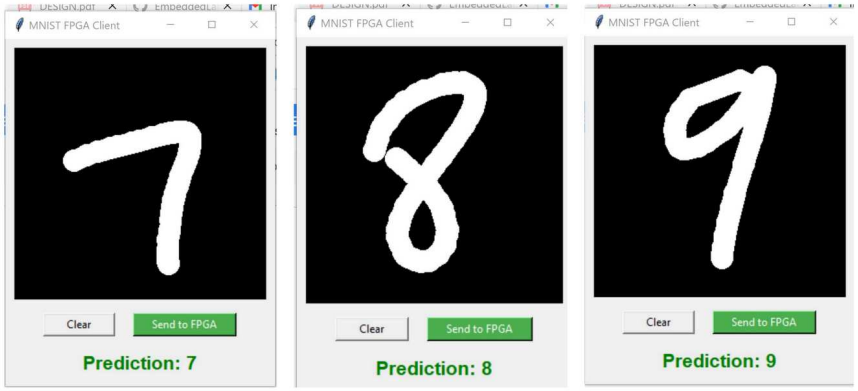
Manual testing with hand-drawn digits

The automated 10 000-image test answers "does the hardware match the float model on the canonical benchmark." The second question we wanted answered was "does it work on *our* handwriting" — drawings produced live in `draw.py` and routed through the same TCP path. We captured screenshots throughout testing; the most informative ones are reproduced below.

Success cases (one example per digit)



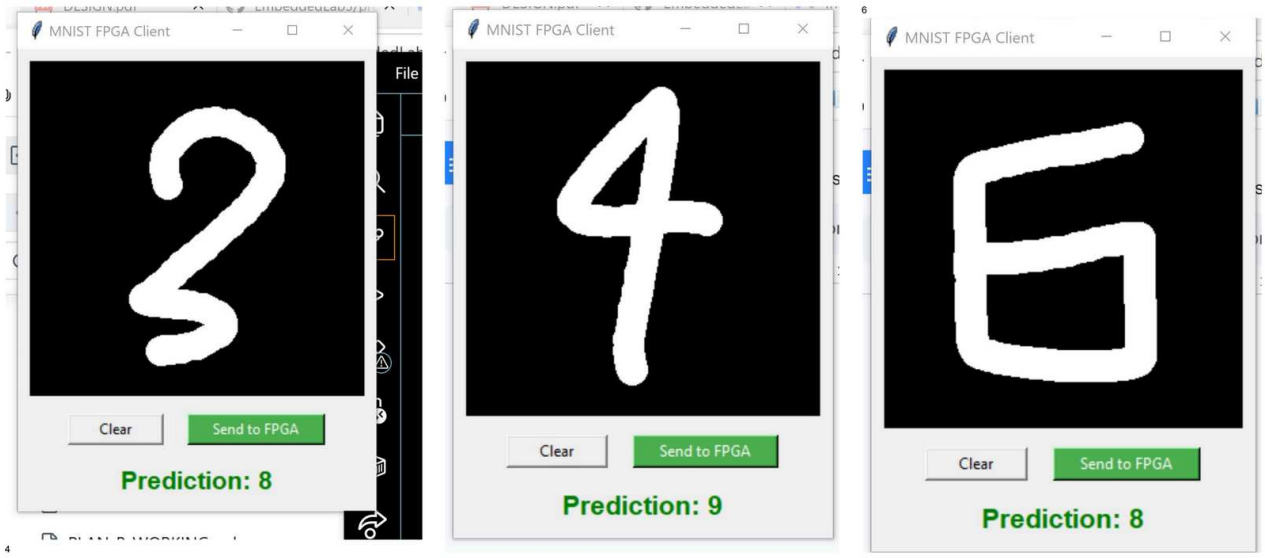




Each panel is a single `Send to FPGA` click in `draw.py`: the user drew the digit shown, hit the green button, the laptop preprocessed it (MNIST-style centring + 20×20 fit + signed-INT8 shift) and sent 784 bytes to `mnist_server`. The big green "Prediction: N" text at the bottom is the byte that came back from the FPGA after the FSM finished. All nine digits were classified correctly on the first attempt with a reasonable drawing.

Style-sensitivity failures

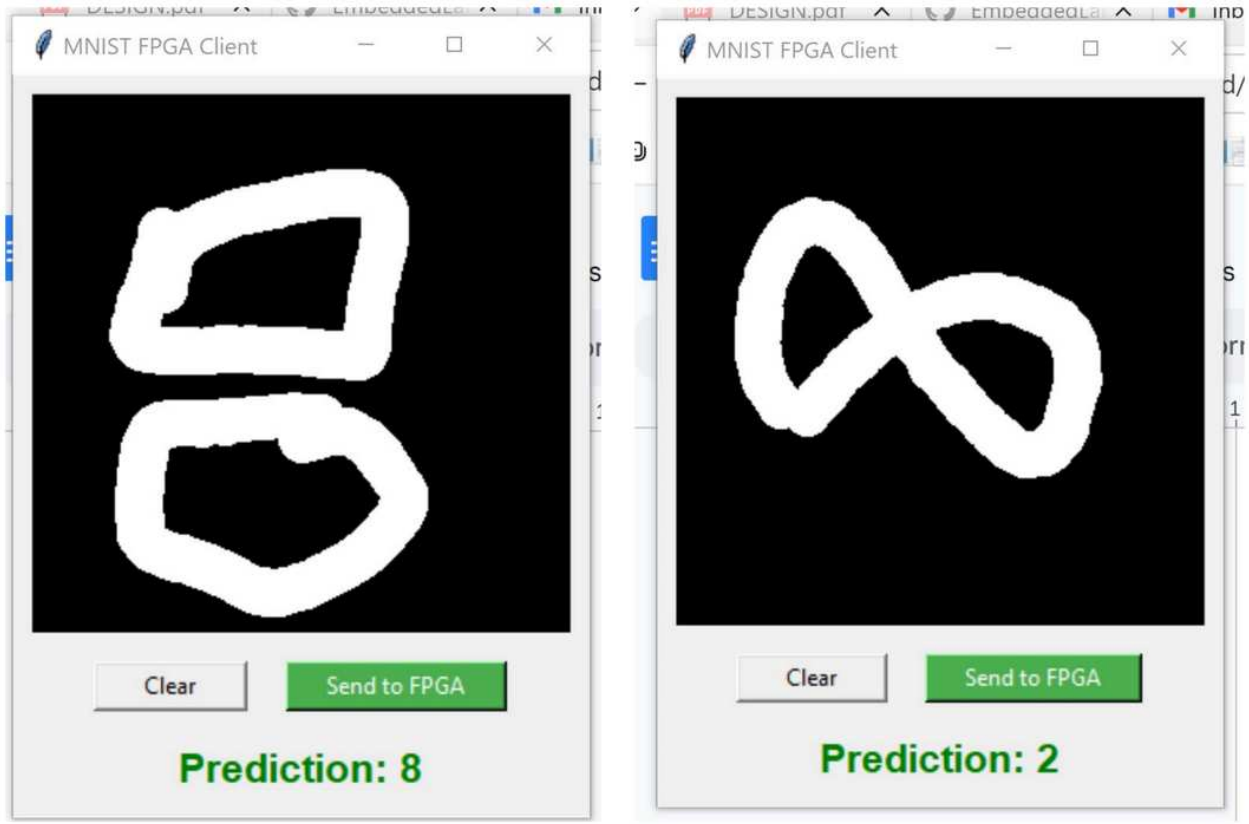
Not every drawing succeeds — some strokes look enough like a different digit (to a 128-hidden-neuron model) that the prediction flips:



Left: an open-curve "2" with a short tail gets classified as 8. Centre: a very tall, thin "4" with a barely-closed loop gets classified as 9. Right: a "6" drawn with a square upper half gets classified as 8. These match the off-diagonal cells in the confusion matrix from ğ5.3 — the model has these same blind spots on the official MNIST test set; the hardware reproduces them faithfully.

Non-digit shapes

Two cases where the user drew something that isn't a digit at all. The model has no "reject" output — it always returns one of 0..9 — so it picks the closest class:



The "glasses" (two adjacent circles) get classified as 8 (which is also two stacked circles). The infinity-style figure gets classified as 2 (which has a similar swooping curve). Both are sensible interpretations given the model has only 10 output neurons.

What these tests demonstrate

- The end-to-end path **works on real handwriting**, not just the MNIST distribution — important for the live demo.
- The Pillow preprocessing in `draw.py` (bounding-box + LANCZOS resize + centre-of-mass) is doing its job; without it, accuracy on these drawings drops dramatically.
- Style-sensitivity failures match the model's known confusion-matrix structure; they are not hardware bugs.
- The model has no out-of-distribution detection, but its mistakes on non-digits are at least visually defensible.

Test artefacts on disk

Path	What it is
`project/eval/test10k_2026-05-11_report.txt`	human-readable summary + confusion matrix
`project/eval/test10k_2026-05-11_results.csv`	one row per image: idx, split, true, predicted, correct
`project/eval/accuracy_bars.png`	per-digit accuracy bar chart
`project/eval/confusion_heatmap.png`	10×10 confusion matrix heatmap
`project/eval/weights_hist_fc1.png`	FC1 INT8 weight distribution
`project/eval/weights_hist_fc2.png`	FC2 INT8 weight distribution

Resource budget

Synthesised for Cyclone V 5CSEMA5F31 (on the DE1-SoC) at a 50 MHz target.

Resource	Used	Available	Utilisation
ALMs (logic cells)	957	32 070	3 %
Registers	1 617	—	—
DSP slices (27 × 27)	8	87	9 %
M10K block-RAM	104	397	26 %
On-chip memory bits	827 840	4 065 280	20 %
Pins	362	457	79 % (DDR3, HEX, switches)
f_{max} target	50 MHz	—	met with margin

Numbers are taken straight from `lab3-hw/output_files/soc_system.fit.summary`. Plenty of headroom — we ran an experiment with FC1 doubled to **128-way parallel** (one bank per output neuron) and FC1 dropped from ~60 ţs to ~30 ţs without exhausting any resource. We kept the production design at 64-way because the inference time is already an order of magnitude below interactive speed and the doubling buys nothing the user can perceive.

Reproduction guide

Build from source

From the repo root on a Windows host with Quartus Prime Lite 21.1 and WSL Ubuntu installed:

```
# Apply our overlay (mnist_accel.sv  lab3-hw/, Qsys descriptor, etc.)
bash project/scripts/apply_soc_overlay.sh

# Train + export + dump samples (~5 min, requires Python 3.10 + PyTorch)
cd project/train
python train.py
python export.py
python dump_test_image.py
python golden.py ../weights/img_7.bin

# Simulate; must pass bit-exact before building bitstream
cd ../tb
vsim -c -do "run -all; quit -f" work.tb_mnist_accel

# Regenerate Qsys + compile Quartus (~10-15 min)
cd ../../lab3-hw
bash ../project/scripts/regen_qsys.sh
quartus_sh --flow compile soc_system.qpf

# Load via Quartus Programmer (USB Blaster)
#    HEX0 lights up about 1.3 ms after configuration completes.
```

Run the live demo

On the board (HPS), after the bitstream is loaded:

```
cd /mnt/sdcard/mnist
sudo insmod sw/mnist_accel.ko
./sw/mnist_server weights        # listens on TCP :4840
```

On the laptop:

```
pip install --user --no-cache-dir Pillow torchvision torch  # one-time
# Edit project/sw/draw.py and set FPGA_IP to the board's IP
python project/sw/draw.py
```

A Tkinter window opens; draw a digit with the mouse, click **Send to FPGA**, the prediction appears on the laptop and the same digit lights up on HEX0.

Run the 10 000-image validation

```
python project/sw/test_corpus.py 10000
python project/sw/plot_eval.py          # regenerates the charts
```

Results land in `project/sw/.test_corpus/`; published copies of our run are checked into `project/eval/`.

File index

```
project/
 DESIGN.md                        earlier-version design document (Markdown)
 DESIGN.tex                       design document source (LaTeX)
 DESIGN.pdf                       design document, compiled
 PRESENTATION.md                  narrative slide deck (Markdown)
 PRESENTATION.pdf                 slide deck, compiled
 FINAL_REPORT.md                  this report (Markdown)
 FINAL_REPORT.pdf                 this report, compiled
 hardware.jpg                     hand-drawn architecture diagram

 hw/
    mnist_accel.sv              accelerator RTL (~380 lines)
    hex7seg.sv                  7-seg decoder (from Lab 1)
    mnist_accel_hw.tcl          Qsys component descriptor
    soc_overlay/                patches to lab3-hw (soc_system.qsys etc.)

 sw/
    mnist_accel.h               shared kernel/user ioctl header
    mnist_accel.c               Linux kernel module
    mnist_server.c              HPS TCP server  (the production path)
    mnist_pick.c                HPS interactive picker
    mnist_run.c                 HPS batch tester
    draw.py                     laptop Tkinter drawing client
    test_corpus.py              laptop automated MNIST validator
    plot_eval.py                laptop chart generator
    Makefile                    builds .ko + userspace

 train/
    train.py                    PyTorch training
    export.py                   INT8 quantisation
    golden.py                   numpy integer reference forward pass
    dump_test_image.py          sample-image extractor

 tb/
    tb_mnist_accel.sv           SystemVerilog testbench (139 asserts)
    run.do                      Questa/ModelSim driver script

 weights/                         output of training pipeline
    fc1_w.bin ů fc1_b.bin       INT8 / INT32 weight blobs
    fc2_w.bin ů fc2_b.bin
    img_0.bin .. img_9.bin      sample test images (INT8)
    labels.txt                  ground-truth labels for samples
    scales.txt                  per-tensor float scales from quantisation

 eval/                            validation artefacts (committed)
    test10k_2026-05-11_report.txt
    test10k_2026-05-11_results.csv
    accuracy_bars.png
    confusion_heatmap.png
    weights_hist_fc1.png
    weights_hist_fc2.png

 scripts/
     apply_soc_overlay.sh        copies hw/ into lab3-hw/
     regen_qsys.sh               regenerates Qsys after edits
     swap_image.sh               re-bake a different sample digit
     render_slides.py            builds {PRESENTATION,FINAL_REPORT}.pdf
     pdf_render.py               PDF  PNG for QA
     plot_eval.py                see sw/ (also lives here)
```

Acknowledgements

Course and dataset

The DE1-SoC hardware, Quartus tooling, lab template, and HPS Linux image are courtesy of CSEE 4840 (Prof. Stephen A. Edwards). The MNIST dataset of handwritten digits is from Yann LeCun, Corinna Cortes, and Christopher J.C. Burges, *The MNIST database of handwritten digits* (1998), <http://yann.lecun.com/exdb/mnist/>. The `hex7seg.sv` decoder module is reused unchanged from Lab 1 of this course.

AI assistance — Anthropic Claude

Substantial portions of this project's *non-RTL* surface — the Python training pipeline (`train.py`, `export.py`, `golden.py`, `dump_test_image.py`), the SystemVerilog testbench (`tb/tb_mnist_accel.sv`), the Linux kernel driver and userspace clients (`sw/mnist_accel.c`, `sw/mnist_server.c`, `sw/draw.py`, `sw/test_corpus.py`, `sw/plot_eval.py`), the build scripts under `scripts/`, this report, the presentation deck, and the design document — were drafted, iterated, and debugged in collaboration with **Anthropic Claude (Claude Opus 4 / 4.6 / 4.7, claude.ai/code)**. All AI-generated artefacts were read, edited, validated against the bit-exact testbench and the 10 000-image hardware-in-the-loop test, and integrated by the authors. The RTL design decisions (banking layout, FSM structure, the BRAM-drain bug, the `ramstyle="M10K"` fight, the Quartus/WSL toolchain workarounds) were owned by the authors. The hardware-in-the-loop accuracy numbers reported in ğ5 are measurements, not estimates.

Open-source software

Tool	Used for
Quartus Prime Lite 21.1 (Intel)	synthesis, place-and-route, programming
ModelSim ASE 18.1 (Intel, free edition)	RTL simulation for the testbench
PyTorch 2.x and torchvision	float MLP training
NumPy	integer reference (`golden.py`) and validation tooling
Pillow	image preprocessing (`draw.py`), chart rendering (`plot_eval.py`)
pypdfium2	extracting test screenshots from `TESTING.pdf`
tectonic	LaTeX engine for this PDF
mermaid.ink	server-side Mermaid diagram rendering

Repo: <https://github.com/Nuk3-W/EmbeddedLab3>, branch `Parallel`.