MNIST Digit Recognition on an FPGA

A custom integer-inference accelerator on the DE1-SoC

Ifesi Onubogu · Colin Paul Jaworowski · CSEE 4840 Spring 2026

For the full written report (every BRAM port, every signal bit-slice), see [`FINAL_REPORT.pdf`](FINAL_REPORT.pdf).

Part 1 · What we built

A 25-slide tour of the project, organised so each slide adds one clear idea on top of the last. The arc: **(1) hook → (2-3) system context → (4-8) the algorithm → (9-11) mapping math to silicon → (12-14) the hardware as built → (15-18) zoom into key subsystems → (19-22) validation → (23-25) wrap-up.**

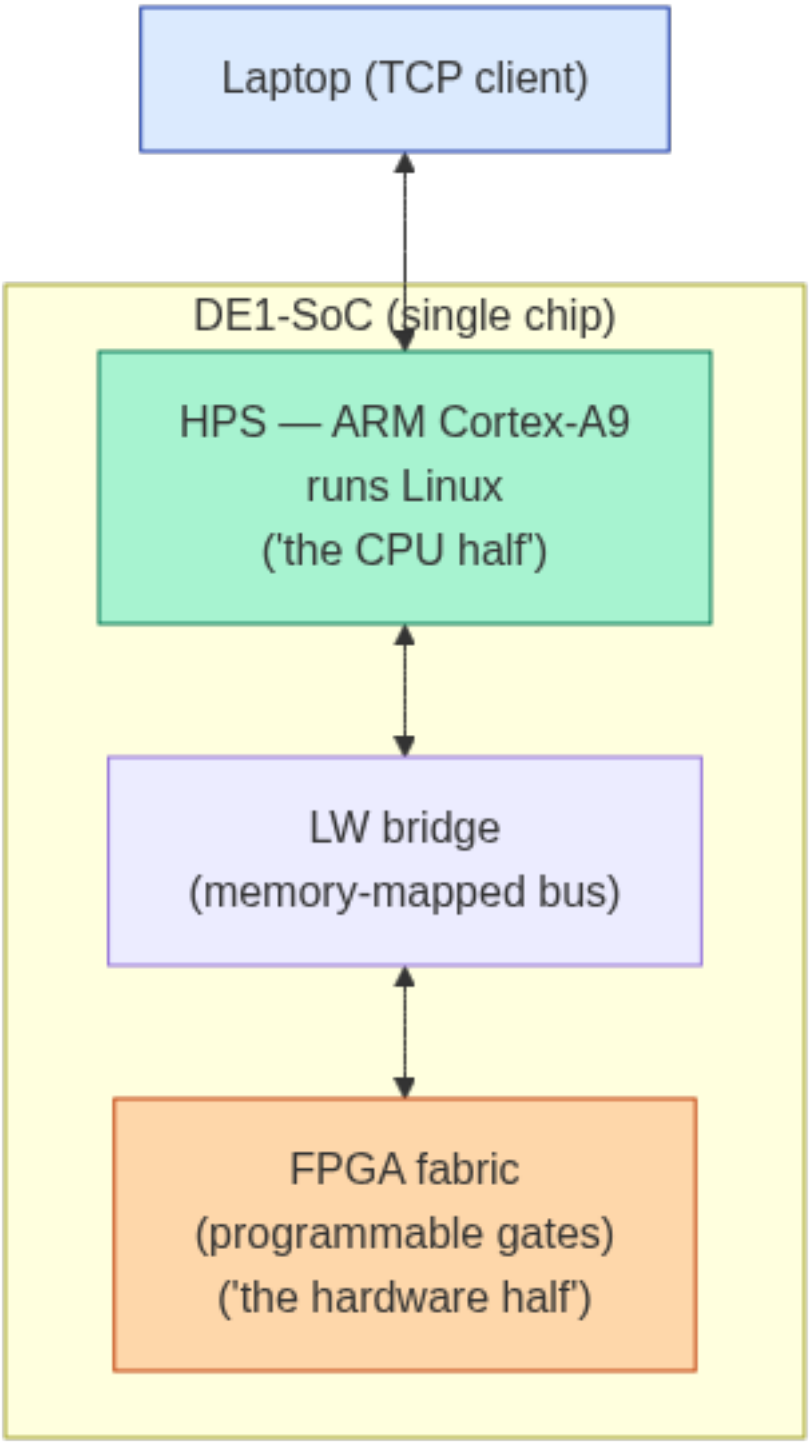
Slide 1 — The pitch in one sentence

> **You draw a digit on your laptop. ~1 ms later, the FPGA shows what it thinks the digit is.**

Why this is interesting: a regular CPU could classify the digit in microseconds. The point is *not* speed. The point is taking a trained neural network and turning it into **dedicated digital hardware** — the same kind of exercise that goes into building a Google TPU or a Tesla FSD chip, just at hobbyist scale.

Slide 2 — The two halves that make this work

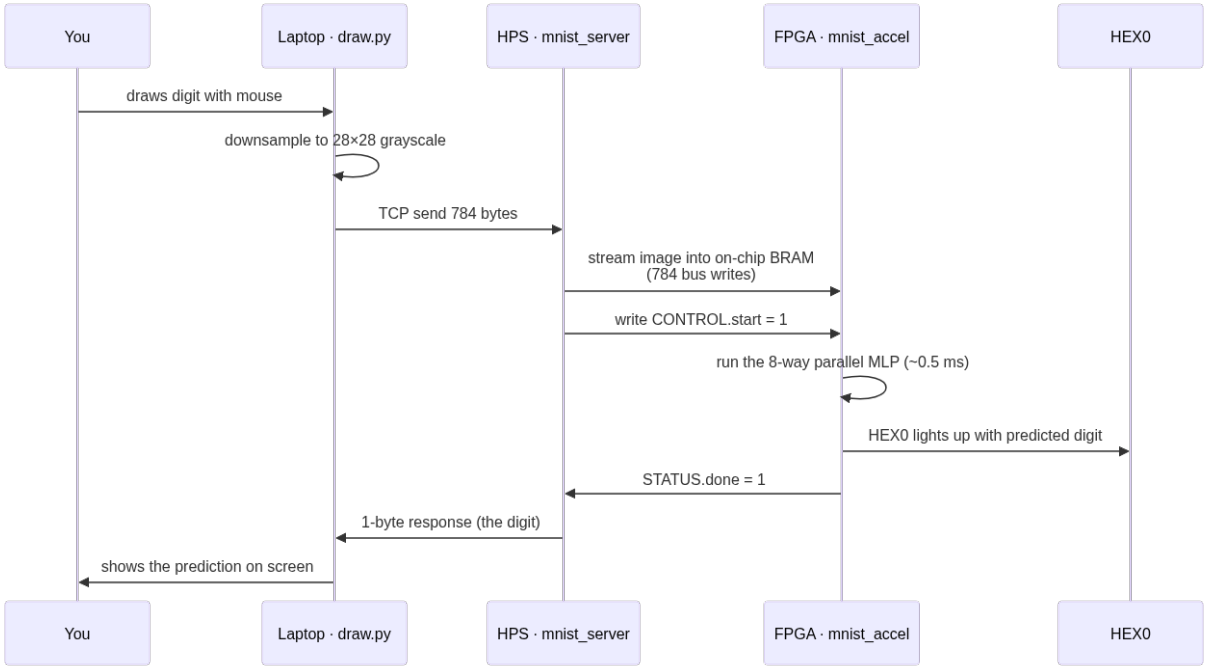
The DE1-SoC chip has two completely separate "computers" on one die — they share nothing except a bus between them.



Half	What it runs	Role
HPS (ARM CPU)	Linux + our `mnist_server` + kernel driver	Talks to the laptop, ferries data through the bus
FPGA fabric	Our custom SystemVerilog circuit (`mnist_accel`)	Does the actual neural-network math in hardware

The CPU half ferries bytes around. The FPGA half does the math. Everything that follows lives on the FPGA side.

Slide 3 — End-to-end flow per digit



Total wall-clock: ~1.4 ms per digit (mostly TCP overhead — the FPGA part is ~90 ţs).

Now that you know the *what*, the next five slides explain the *why* — what's actually being computed inside that ~90 ţs.

Part 2 · The neural network

Before we can talk about the hardware, you need to know what math the hardware is doing. This part covers exactly that — what the network is, where the weights come from, and why we can replace floats with 8-bit integers without losing accuracy.

Slide 4 — What is a "neural network", really?

A neural network is a stack of layers, where **each layer is just a big pile of multiply-adds.**

For one *neuron* with `n` inputs:

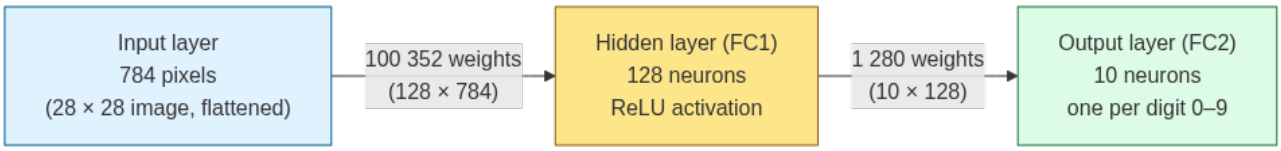
```
output = ReLU(  w₀x + w₁x +  + wₙx + bias  )

                  weighted sum of inputs
```

Where:

- `x` are the inputs (numbers coming from the previous layer)
- `w` are the *weights* — fixed numbers learned during training
- `bias` is one more fixed number
- `ReLU(x)` is just `max(0, x)` — clamps negatives to zero

That's *literally it.* Multiply, add, clamp. A "layer" is just lots of these neurons in parallel, each with their own weights.

Slide 5 — Our network: 784 → 128 → 10



To classify a digit:

1. Flatten the 28×28 image into 784 numbers. 2. Each of the 128 hidden neurons takes all 784 inputs, computes Σ`wx + b`, applies ReLU. Now we have 128 numbers. 3. Each of the 10 output neurons takes all 128 hidden values, does another weighted sum. Now we have 10 scores. 4. **Whichever output score is the largest** → that's the predicted digit.

Total operations per inference: ~101 600 multiply-accumulates.

Slide 6 — How we trained the model

The model has **101 770 parameters**. We don't pick those values — an algorithm finds them by trial and error, *offline on a laptop in PyTorch*. The FPGA never trains; it only runs the already-trained weights at inference.

```
initialize all 101,770 weights to small random values

for epoch in 1..5:                          # 5 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 weight
        2. FORWARD scores = model(batch_images)
                   loss   = cross_entropy(scores, labels)    # one number
        3. BACKWARD loss.backward()                          # fills w.grad
                                                             # via chain rule
        4. UPDATE  w = w - learning_rate * w.grad
```

The four numbered steps in plain English:

1. **Reset** the running gradient buffer so each batch starts fresh. 2. **Forward pass.** Run a batch of 128 images through the network; compute one *loss* value — small if the correct-digit score is largest by a wide margin, big if the model is confidently wrong. 3. **Backward pass.** PyTorch applies the chain rule automatically (`loss.backward()`) to compute the *gradient* of the loss w.r.t. each of the 101 770 weights — one number per weight saying "if I nudge this up by , would loss go up or down?" 4. **Update.** `w` ← `w  learning_rate` × `gradient`. Each weight moves a tiny step (~0.2 % of its size) in the direction that locally reduces loss.

5 epochs × 469 batches ≈ 2 345 weight updates total. Test accuracy plateaus at **96.4 %** around epoch 5; further training risks *overfitting* (memorising training-set quirks instead of generalising). The final weights are saved to `mlp.pt`; `export.py` then quantises them to INT8 (covered in the next two slides).

Slide 7 — Quantisation: how float weights become INT8

There are **two separate operations** to get from float training to integer inference:

- **Weight quantisation**: `W_int8 = round( W_float / s_w1 )` — turns float weights into signed INT8. **Some information is lost** (rounding error per weight), but the ğnext-slide argument shows it doesn't blow up the network.
- **Pixel shift**: `x_int8 = pixel_uint8  128` — turns uint8 pixels into signed INT8. **No information is lost** (exact integer subtraction).

Both yield signed INT8 in `[128, +127]` so the FPGA's `signed` × `signed` multipliers work on them uniformly.

INT8 = **256 buckets** from 128 to +127. Every float weight has to round to one of them.

Weight conversion (used by `export.py`):

```
s_w1 = max(|W_float|) / 127  =  0.608 / 127  =  0.004785
W_int8 = round( W_float / s_w1 )
```

Float weight	œ 0.004785	rounded	INT8
0.608 (largest)	127.06	127	max positive
0.300	62.69	63	
0.043562	9.10	9	
0.005217	1.09	1	
0.000843	0.18	0	rounded to zero
0.018324	3.83	4	
0.231715	48.42	48	
0.608	127.06	127	max negative

Slide 8 — Why this doesn't ruin the network

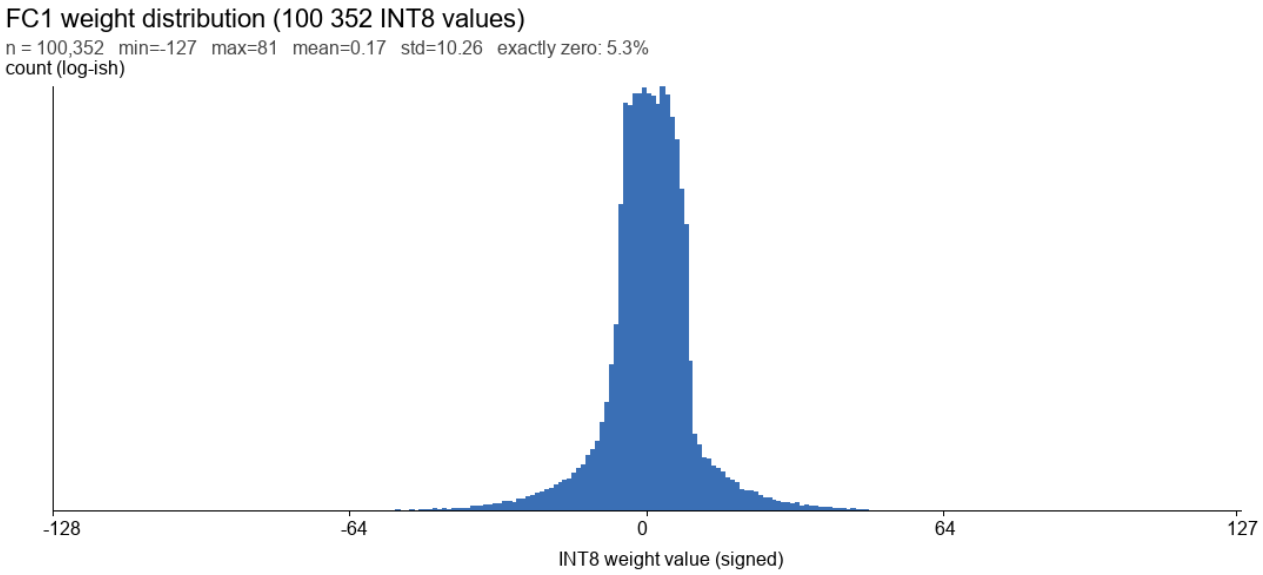
Reconstructed-vs-original error per weight (multiply INT8 back by `s_w1`):

Float weight	INT8	Reconstructed	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

Worst-case error per weight: **ś0.0024** (half a bucket).

- **Decision = which output is biggest, not its exact value** → small wobbles don't flip the winner.
- **784 errors per output, roughly random** → combined error grows like $784 \approx 28$, not 784×. Net output noise **≈ 0.18 % of signal**.
- **Decision margin between winning class and runner-up is ~20–50 %** of the signal → noise is two orders of magnitude smaller than the margin.

Float test accuracy 96.4 % → INT8 hardware accuracy 96.53 %. Quantisation cost = zero.

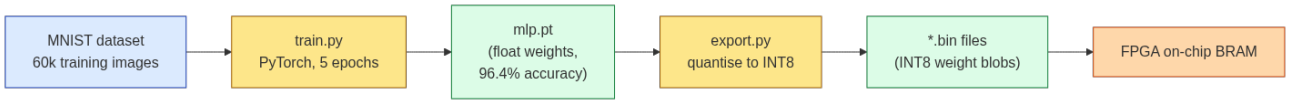


Part 3 · Mapping the math onto FPGA hardware

We now know what the math is. The next three slides show how a single multiply-accumulate becomes a circuit, and how we copy that circuit 64 times to get the speedup our parallel design relies on.

Slide 9 — One MAC (multiply-accumulate) in hardware

The inner-loop operation of *every neural network ever*:



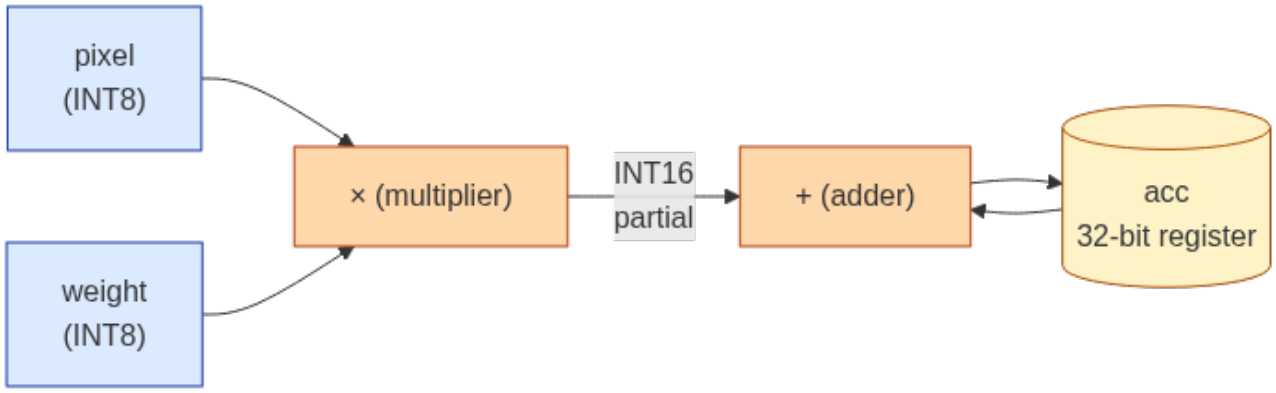
That's **one** MAC. Our naïve sequential design did them one at a time:

- 784 MACs per output neuron × 128 output neurons × 2 cycles/MAC = ~201 000 clock cycles for FC1
- At 50 MHz that's ~4 ms — slow.

The parallel design replicates this 64 times for FC1 and runs all 64 simultaneously. Each cycle does 64 MACs in parallel; FC1 takes ~3 000 cycles ≈ **60 µs** (a ~30× speedup over the sequential reference).

Slide 10 — Why "banking" works: split the work

128 hidden neurons → split into **64 banks of 2 neurons each**.



Each of the 64 banks has its **own** weight memory and **own** accumulator. They all read the same input pixel, but each multiplies by a *different* weight (the weight for *its* neuron). So in one cycle, 64 neurons make progress simultaneously.

After 784 cycles all 64 finish — we've computed 64 neurons' results at the same time. Repeat **twice** to cover all 128 hidden neurons. Done in $2 \times 784 \approx 1\,600$ MAC cycles ($\approx 3\,000$ total with the 2-cycle BRAM read) instead of 100 000.

FC2 uses the same trick but only 2 banks (10 outputs / 2 banks = 5 per bank). We also synthesised a **128-way FC1 variant** which finishes FC1 in ~30 ţs — kept off the default build because 60 ţs is already invisible to the user.

Slide 11 — Cost vs benefit of banking

Resource	Sequential (reference)	Parallel (built)	Note
ALMs (logic)	957	957	Quartus packed 64 narrow 8-bit MACs into the same ALMs the sequential design used (8-bit multipliers are cheap)
DSP slices	8	8	unchanged — the wide multipliers stay on DSP, the new narrow ones land in logic
BRAM blocks	~30	**104**	the FC1 weights are split across 64 small banks instead of 1 big bank, so the M10K count grows but bit-count is the same
FC1 phase time	~4 ms	**60 µs**	~64× faster
Chip utilisation	3 % ALMs	3 % ALMs	still extremely low — plenty of headroom

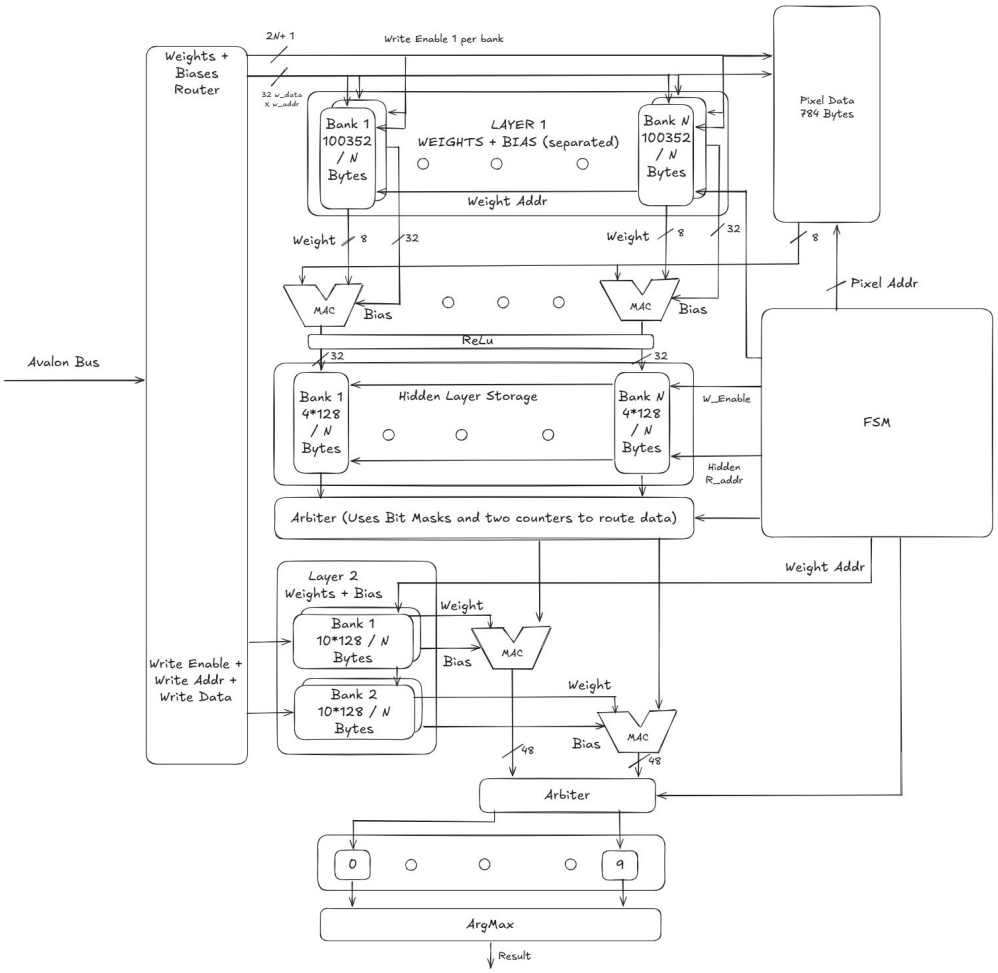
The whole speedup came from re-arranging the BRAM layout into 64 parallel banks. We didn't have to spend more logic to get it — the fit report confirms ALMs barely moved.

Part 4 · System architecture (the hardware, as built)

The next three slides show the actual hardware we put on the FPGA — first the canonical hand-drawn block diagram, then the same blocks with their signal-level bit widths labelled, then the software-facing register interface.

Slide 12 — The hardware, as we drew it

Hand-drawn block diagram of the parallel `mnist_accel` peripheral (Excalidraw):



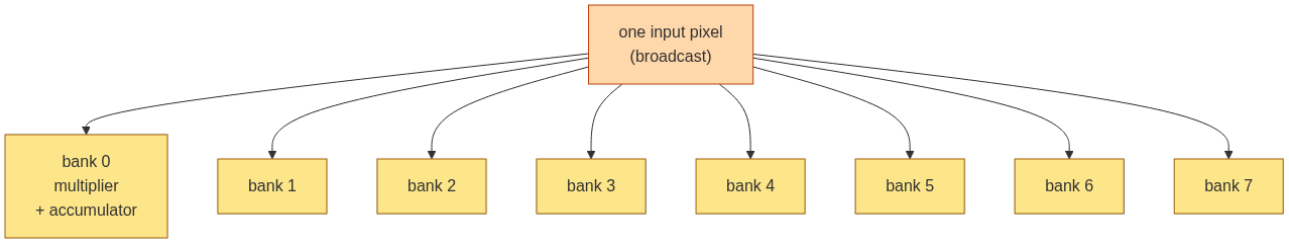
Editable source: <https://excalidraw.com/#json=b0_C1xVcVGlzJ1NlJmrzn,VjF53zlqBiEnRJ79-cxWzQ>

Reading this: Avalon bus comes in 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 weight+bias banks** (centre-left) and the **pixel-data BRAM** (top-right). During inference, an **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 digit is the result.

This is the canonical reference for the rest of the deck — every subsystem zoom in Part 5 zooms into one block here.

Slide 13 — The same diagram with bit widths

The same block diagram, re-drawn in Mermaid with the **actual SystemVerilog signal names and bit widths** on every wire. Useful when reading the source.



Subsystem	One-line job
Avalon-MM slave	Speaks the bus protocol — registers writes/reads
Register file	8 control/status registers software can poke
Smart router	When software streams weights, routes each byte into the correct BRAM bank
BRAMs	On-chip memories holding weights, image, hidden activations
FSM	Sequences the inference — knows which state to be in, which MAC to fire
Datapath	The math — 8 parallel multiply-accumulators, ReLU, 10-way argmax
HEX0 driver	4-bit predicted digit → 7-segment pattern (combinational ROM)

Slide 14 — The bus interface (Avalon-MM register map)

Software sees **8 × 32-bit registers** at addresses `0xFF200040` to `0xFF20005F`.

Offset	Name	Access	What it does
`0x00`	`CONTROL`	W	bit 0 = start inference, bit 1 = load mode, bit 2 = clear done
`0x04`	`STATUS`	R	bit 0 = busy, bit 1 = done (sticky)
`0x08`	`RESULT`	R	predicted digit in bits [3:0]
`0x0C`	`CONFIDENCE`	R	winning neuron's accumulator value (debug)
`0x10`	`LOAD_ADDR`	W	resets the streaming-load counters
`0x14`	`LOAD_DATA`	W	streams one byte into the currently-selected BRAM target
`0x18`	`LOAD_TARGET`	W	which BRAM to stream into (FC1_W, FC1_B, FC2_W, FC2_B, IMG)
`0x1C`	`VERSION`	R	reads constant `0x4D4E5301` ("MNS\{}x01") — sanity check

To run one inference, the kernel driver: 1. Streams the image (784 bytes) into `LOAD_DATA` with `LOAD_TARGET = IMG`. 2. Writes `CONTROL = 1` (start). 3. Polls `STATUS` until bit 1 (done) is set. 4. Reads `RESULT` → returns digit to userspace.

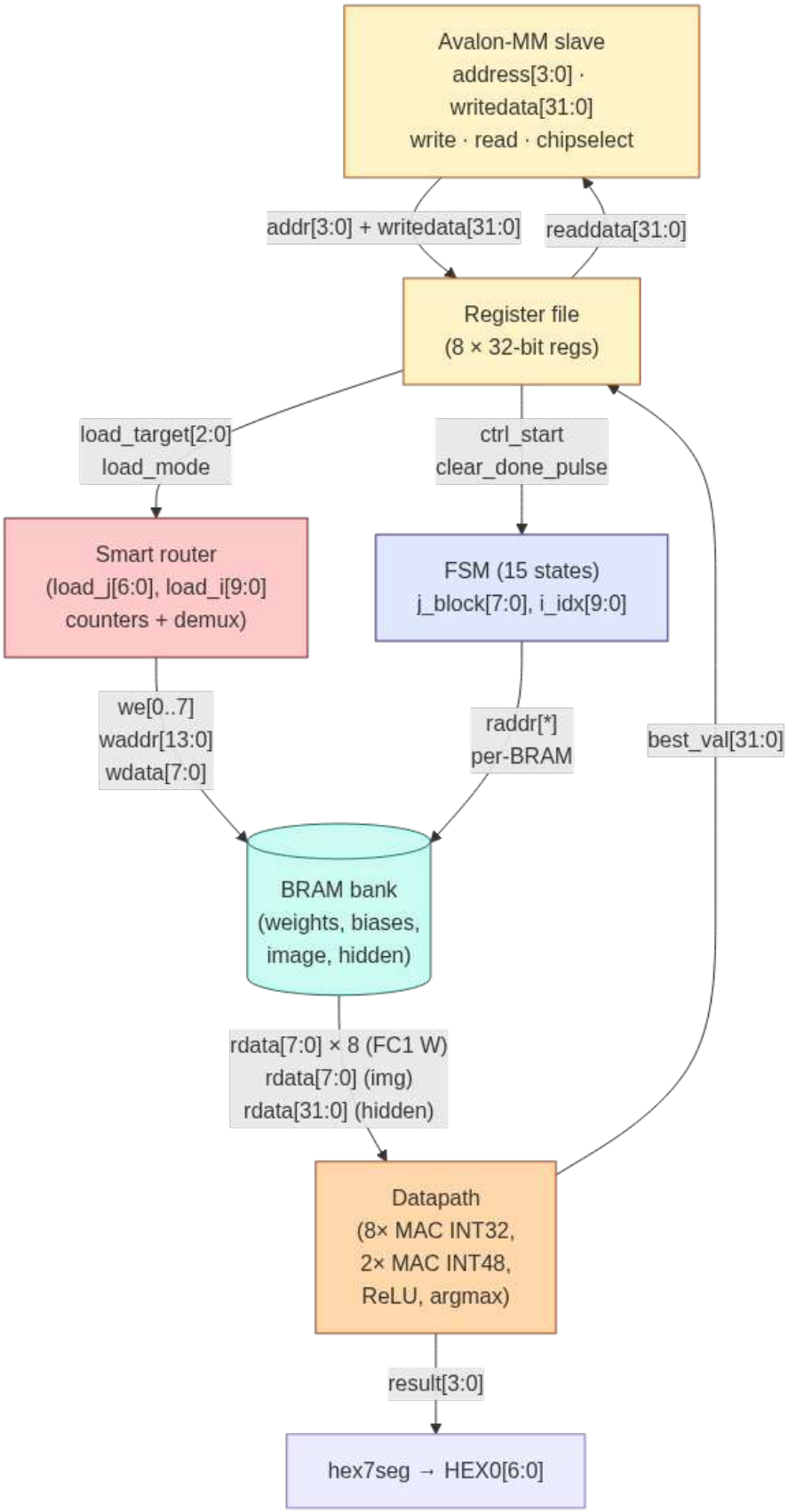
Weights are loaded once at boot, never reloaded between inferences.

Part 5 · Zoom into key subsystems

Four of the subsystems on Slide 12 deserve a slide of their own — the FSM (orchestrator), the smart router (load path), the FC1 weight banking (storage), and the hidden-layer scatter/gather (FC1→FC2 plumbing).

Slide 15 — The FSM: sequencing one inference

15 states; counters `j_block[7:0]` (outer) and `i_idx[9:0]` (inner). The FSM is suspended whenever `load_mode = 1`. Phase-level view:



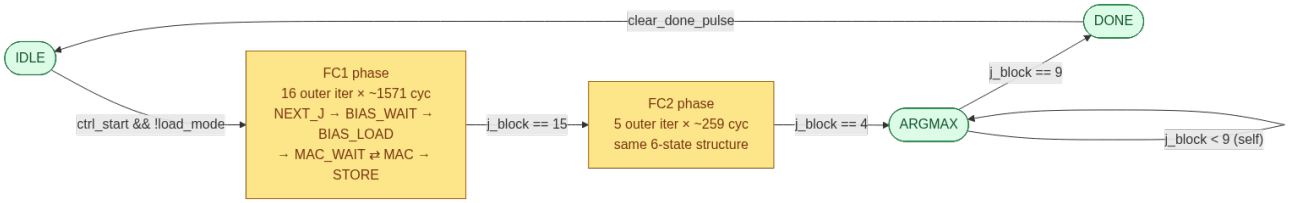
Each phase box hides a 6-state inner loop — full list of states + transitions:

State	Next state	Trigger
FC1_NEXT_J → BIAS_WAIT → BIAS_LOAD → MAC_WAIT → 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 → BIAS_WAIT → BIAS_LOAD → MAC_WAIT → 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 (64-way FC1): FC1 $\approx 2 \times 1571 = 3\,000$ cyc · FC2 $\approx 5 \times 259 = 1\,300$ cyc · ARGMAX $= 10$ cyc · misc transitions ≈ 140 cyc → ~**4 500 cyc ≈ 90 µs @ 50 MHz** (FC1 phase alone ≈ 60 µs).

Slide 16 — Smart router: serial in, parallel out

Software writes one byte at a time. Hardware splits the 7-bit `load_j` counter into a **bank-id** (low bits) and a **row index** (high bits) — the bank-id picks which of 64 BRAMs gets the write enable; the row index becomes part of the write address.



Target	Bank-select bit-slice	Row-select bit-slice	# 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	(no banking — single BRAM)	`load_i[9:0]` is the waddr	1

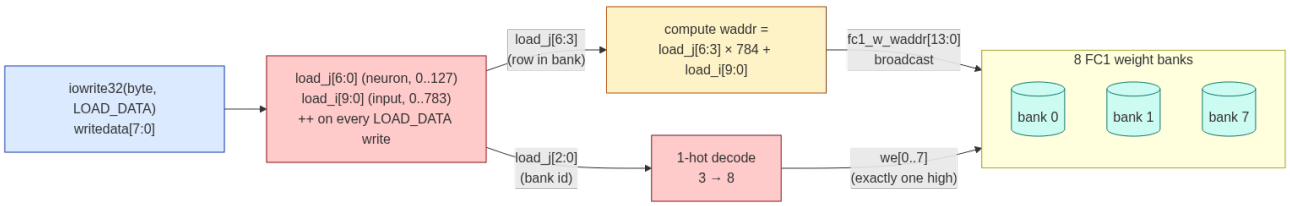
The whole router is one ‘case‘ statement plus one decoder. ~50 ALMs total.

Slide 17 — FC1 storage: how the weights are laid out

128 hidden neurons split into 64 banks of 2. **Interleaved** layout: bank `b` holds neurons whose number is congruent to `b` mod 64.

```
Bank 0  holds neurons:  0, 64
Bank 1  holds neurons:  1, 65
Bank 2  holds neurons:  2, 66

Bank 63 holds neurons: 63, 127
```

When the FSM holds `j_block = 0`, all 64 banks collectively read neurons 0..63 in **the same cycle**. When `j_block = 1`, they read 64..127. Two reads cover all 128 neurons.



Each bank is **one M10K block** of on-chip BRAM (1 568 × 8 bits ≈ 12.5 kbits). Total weight storage: 64 × 12.5 kbits ≈ 800 kbits, ~20 % of the chip's on-chip memory — the same total bits as before, just spread across more (smaller) blocks for parallel access.

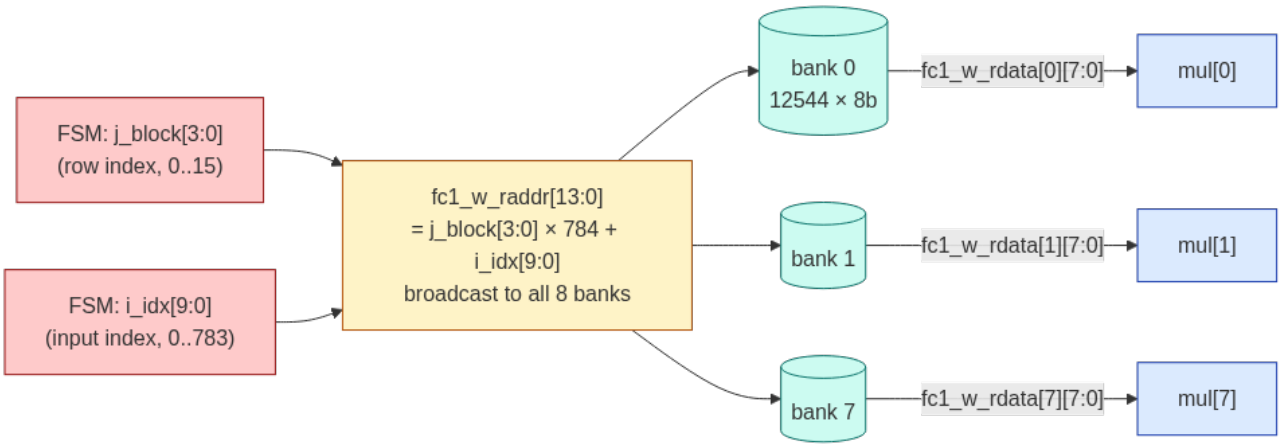
Why 'j_block[0]' and not the full 'j_block[7:0]'? The register is 8 bits because it gets reused during FC2 (5 iterations) and argmax (10 iterations). For FC1 only the low bit matters (range 0..1).

Slide 18 — The clever bit: hidden-layer scatter/gather

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

The 7-bit FC2 inner counter `i_idx[6:0]` is **physically the same wires**, but the FSM interprets it as two fields:

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


When i_idx = 73 (binary 1001001):
       i_idx[6]   = 1        row 1 inside each bank
       i_idx[5:0] = 001001 = 9  pick bank 9's output
    hidden_rdata = h_ram[9][1] = hidden activation #73
```



All 64 BRAMs read the same row in parallel; a 32-bit 64:1 mux picks the right bank's output. **Cost: ~250 LUTs.** One line of SystemVerilog (`mnist_accel.sv:91`):

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

Part 6 · Validation

Hardware is built — but does it actually classify correctly? Three slides on capacity + accuracy + failure modes.

Slide 19 — Resource budget on the chip

We're using a Cyclone V 5CSEMA5F31. Here's our footprint:

Resource	Used	Available	%
ALMs (logic cells)	957	32 070	3 %
Registers	1 617	—	—
DSP slices (27×27)	8	87	9 %
M10K BRAM blocks	104	397	26 %
On-chip memory bits	827 840	4 065 280	20 %
Pins	362	457	79 % (mostly DDR3, HEX, switches)
Max f_max	50 MHz (target)	—	held with margin

Numbers are straight from `soc_system.fit.summary`. **Plenty of headroom** — we synthesised a 128-way FC1 variant that drops FC1 time from 60 ţs to 30 ţs, with no resource crisis. We left the production build at 64-way because 60 ţs is already 100× faster than human-interactive speed.

Slide 20 — Three layers of validation

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

Layer	Tool	What it proves
1	`train/golden.py`	The **quantisation math** is internally consistent and still classifies MNIST correctly — pure-NumPy integer forward pass, same `.bin` weights the FPGA loads, dumps every accumulator to `golden_trace.txt`. Validates the algorithm before any RTL exists.
2	`tb/tb_mnist_accel.sv`	The **RTL is bit-exact** to layer 1 — Questa BFM drives the Avalon slave, runs one inference, diffs every FC1/FC2 accumulator against `golden_trace.txt`. Zero mismatches required. Catches sign-extension, off-by-one MAC indexing, BRAM drain bugs, argmax tie-break direction.
3	`sw/test_corpus.py`	The **whole pipeline classifies real MNIST** at near-float accuracy — 10 000 hardware-in-the-loop inferences with confusion matrix. The headline result.

Layers 1 and 2 are upstream gates: the layer-3 number is meaningless unless they're both green first. The rest of this slide is layer 3.

Slide 20b — Layer 3 in detail: hardware-in-the-loop on the MNIST test set

Once the bitstream is on the board, **we don't trust simulation alone** — we test against real labelled data. The tool: `project/sw/test_corpus.py` (a peer of `draw.py`).

Setup:

```
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, RESULT[3:0]
  4.  compare to ground truth
  5.  record in CSV + accumulate
      confusion matrix
```

Why this corpus matters:

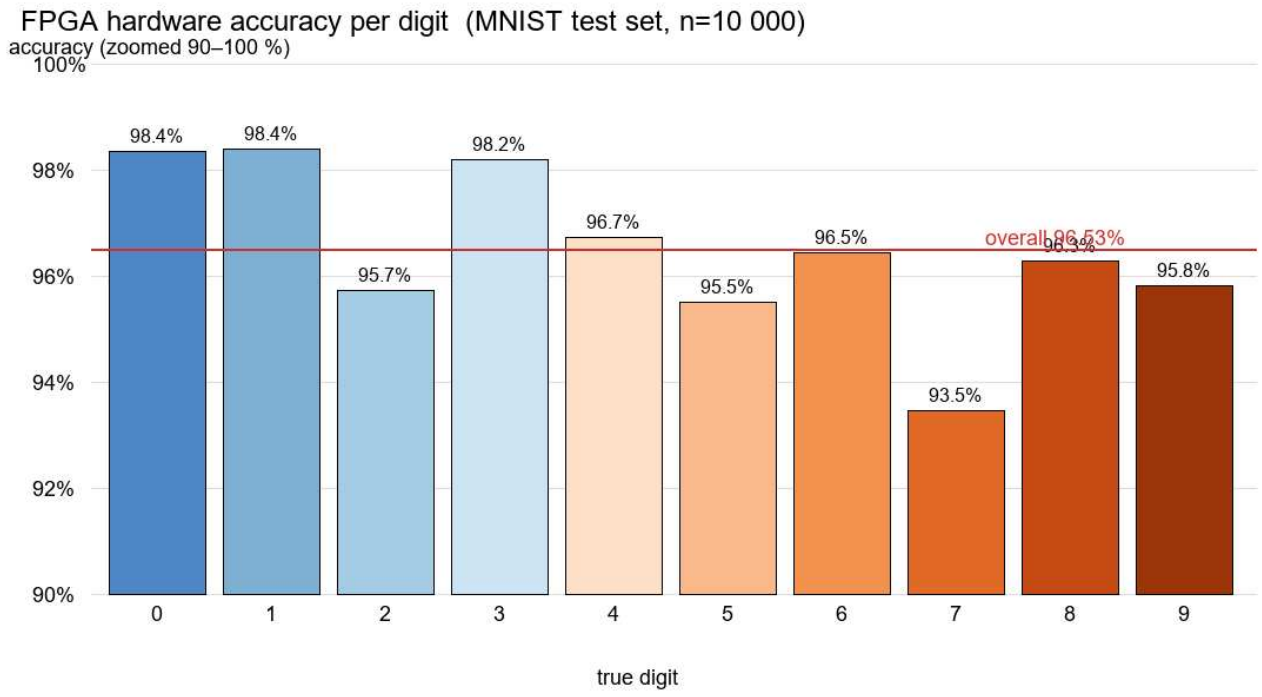
- The **MNIST test set** is 10 000 labelled images the network **never saw during training**. The 96.4 % figure quoted from PyTorch refers exactly to this set — it is the canonical "did we generalise?" benchmark.
- This is **end-to-end**: hardware integer math, the streaming-load path, the FSM, the polling, the TCP plumbing — all in the loop. If anything along the way silently miscomputes, accuracy drops.
- Comparing per-digit hardware accuracy to the float reference also tells us *which* digit (if any) the quantisation hurt — useful diagnostic.

Outputs saved for the record (under `project/eval/`):

- `test10k_2026-05-11_report.txt` — summary + per-digit table + 10×10 confusion matrix
- `test10k_2026-05-11_results.csv` — one row per image: `idx, split, true, predicted, correct`
- `misclassified/` — 347 PNGs, one per wrong prediction, filename encodes the index, true label, and predicted label

Wall-clock budget: 215.4 s for 10 000 images → ~46 inferences/second end-to-end. Inference on the FPGA is ~90 ţs; the rest is TCP round-trip + Python overhead.

Slide 21 — Result: 96.53 % accuracy on the 10 000-image test set

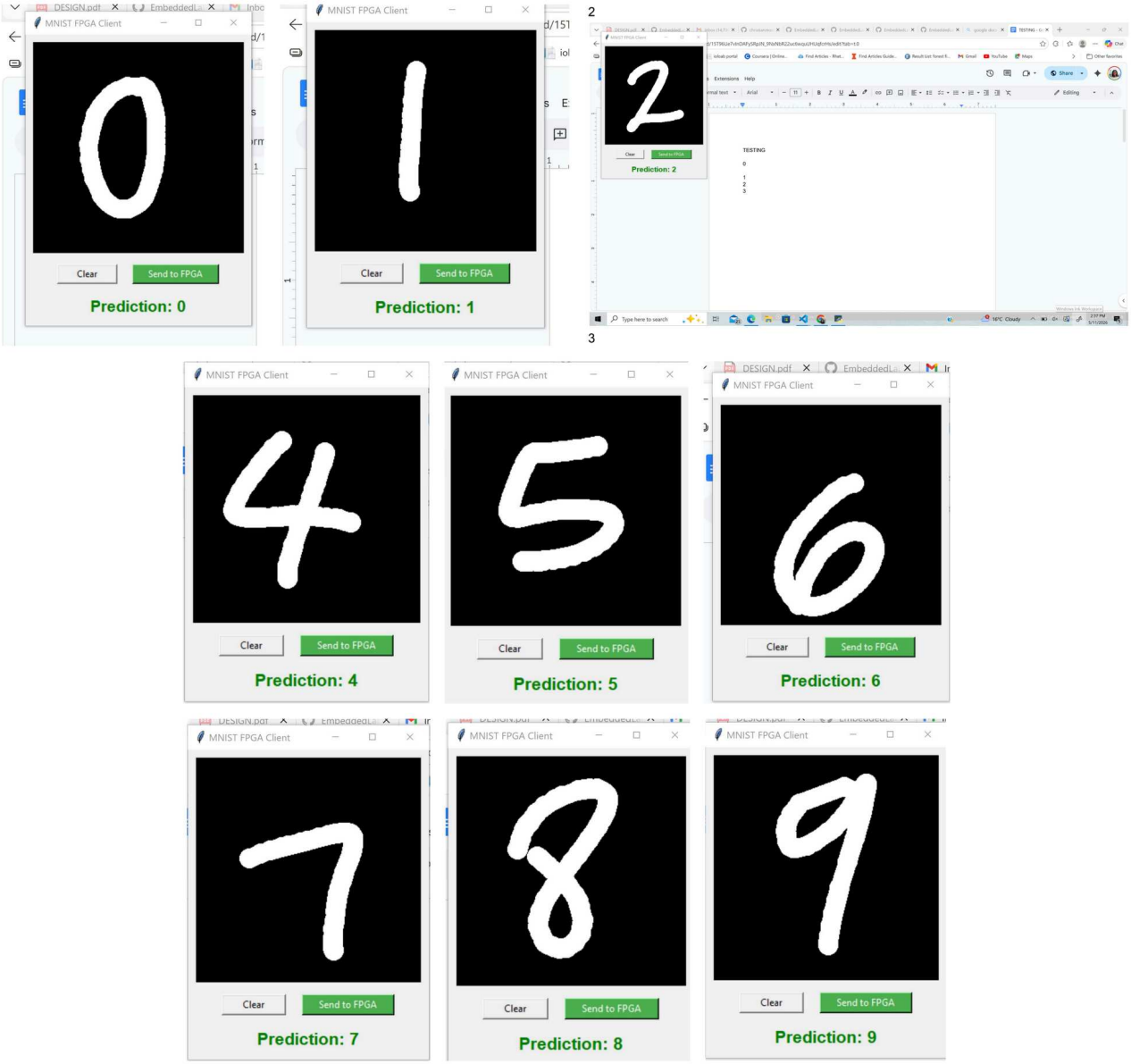


Overall	**96.53 %**	(vs 96.4 % float baseline → quantisation cost ≈ 0)
Best	digits 0 and 1	98.4 % each
Worst	digit 7	93.5 %
Network failures	0 / 10 000	streaming-load path is reliable

Hardware accuracy **slightly exceeds** the float baseline — normal noise. Quantisation rounding sometimes flips a borderline classification in our favour as often as against us; the margin is well within statistical error for $n = 10\,000$.

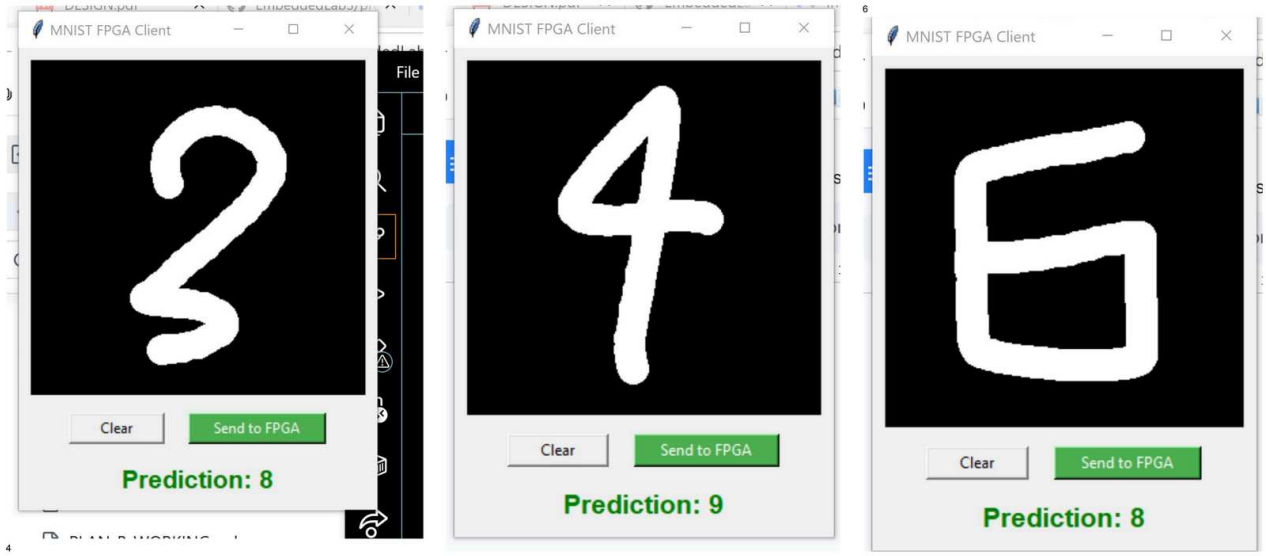
Slide 21b — Live demo: classifying actual hand-drawn digits

The automated test answers "does the hardware match the float model." The next question is "does it work on *our* handwriting?" We drew digits in `draw.py` and captured what the FPGA returned:



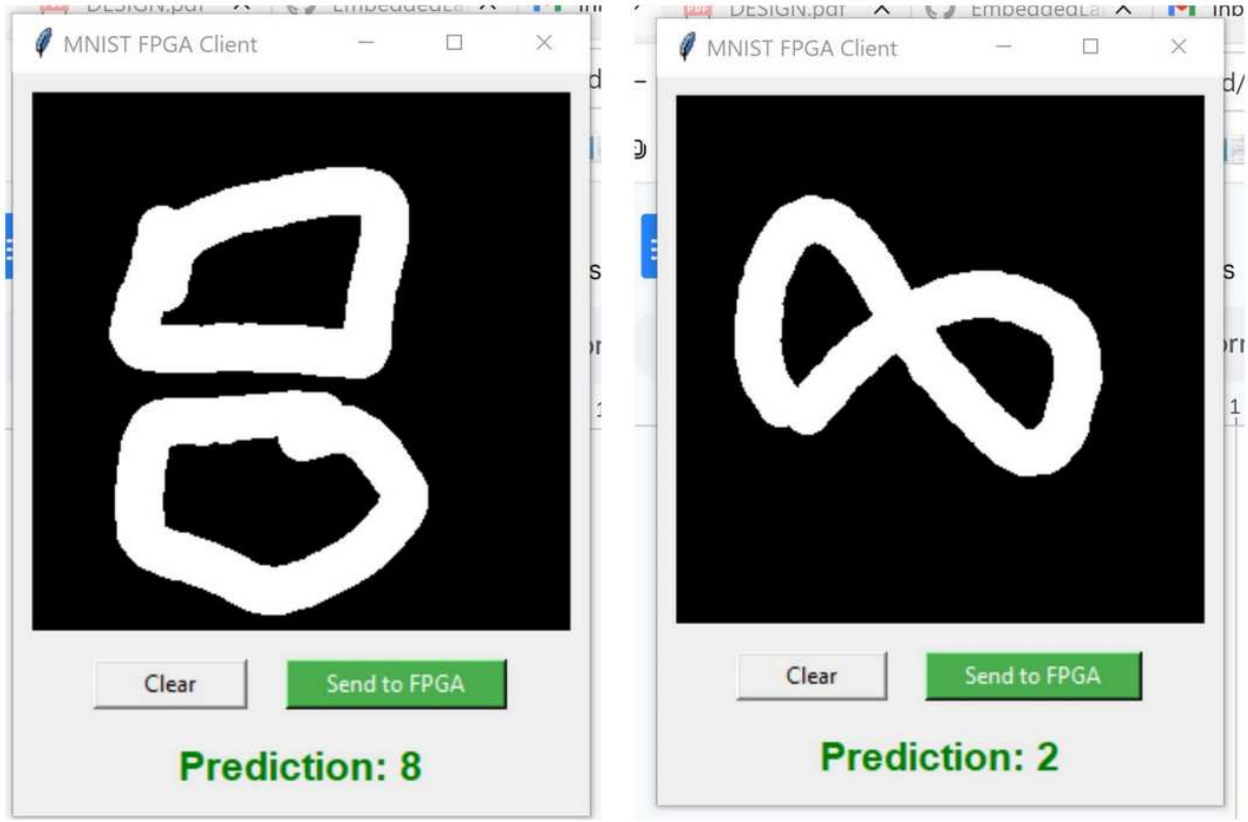
Each panel is one `Send to FPGA` click. Hardware path: user draws → Pillow preprocess (centring + 20×20 fit + signed-INT8 shift) → TCP → ioctl → FPGA inference → 1-byte response → green text. **All nine digits classified correctly on the first attempt** with a reasonable drawing.

Failure modes match the confusion matrix



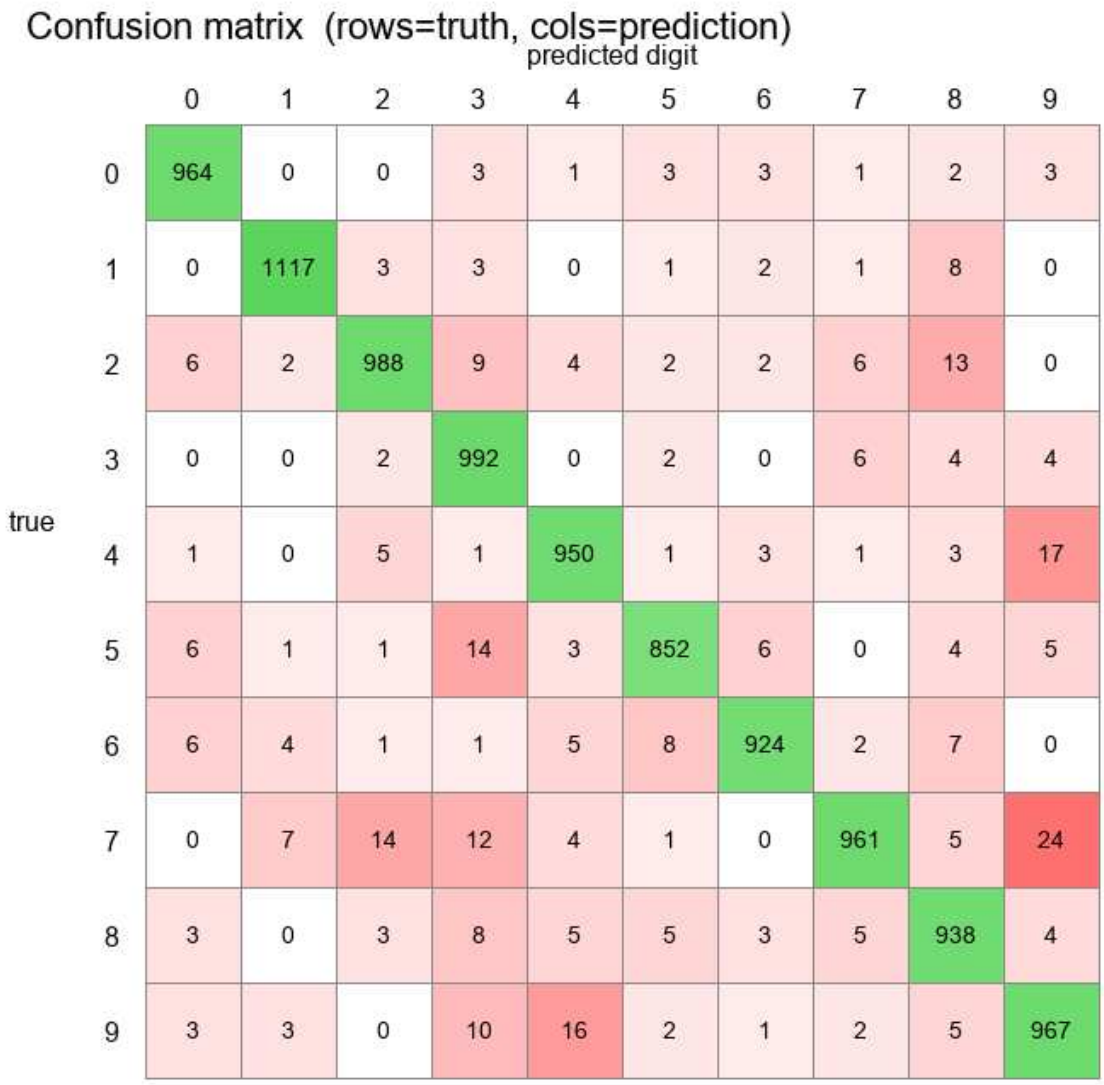
Open-curve "2" → 8 · tall thin "4" → 9 · boxy "6" → 8. Same blind spots the model has on official MNIST.

Non-digit shapes — model still picks the closest class



Glasses (two circles) → 8 · infinity → 2. Reasonable interpretations given only 10 output classes.

Slide 22 — Where the model still makes mistakes (confusion matrix)



Reading the matrix: row = ground-truth digit, column = what the FPGA predicted.

- Strong green diagonal → most predictions are correct.
- The biggest off-diagonal cells tell us **which substitutions the network actually makes**:

- `7` → `9` (24 cases) and `7` → `2` (14 cases): slanted 7s look like 1s/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 makes** — capacity limits of the 128-hidden-neuron architecture, **not** hardware bugs. The hardware faithfully reproduces what the trained float network would do.

Slide 22b — Test 1: `golden.py` (is the integer recipe correct?)

What it proves: if we run the network using *only integers* (no floats), does it still recognise digits correctly? Because the FPGA can only do integer math, and a broken conversion would silently destroy accuracy.

How: we wrote the whole forward pass a second time in Python, forbidden from using floats. Three lines:

```
hidden = relu( fc1_b + fc1_w @ x )       # FC1, int32 throughout
out    =       fc2_b + fc2_w @ hidden    # FC2, int64 throughout
pred   = argmax(out)
```

Feed it an MNIST image → it prints a digit. It matches the true label ~96 % of the time across the test set — so the integer recipe is fine.

Bonus output: while running, it writes down every intermediate number (128 hidden values + 10 output values) to a text file. **This file becomes the answer key for Test 2.**

> Without Test 1, we'd build a circuit, hit 50 % accuracy, and not know whether the math is wrong or the hardware is wrong. Test 1 isolates the math.

Slide 22c — Test 2: Questa testbench (is the SystemVerilog bit-exact to the answer key?)

What it proves: every intermediate number our SystemVerilog circuit computes matches what `golden.py` said it should be — bit for bit, all 138 of them.

Why bit-exact matters: hardware bugs like wrong sign-extension, off-by-one counters, or a BRAM read happening one cycle too early don't crash anything — they just make some numbers slightly wrong. The only way to catch them is to compare *numbers*, not just the final predicted digit.

How: we run our SystemVerilog in **Questa** (a simulator on the laptop — not the real FPGA). The testbench:

```
1. Load the answer key file into memory  ($readmemh)
2. Load the same weights + image into the simulated circuit
   (using the SAME register writes the Linux driver uses)
3. Press "start", poll STATUS until done
4. For each j in 0..127:
       check  dut.hidden_mem[j]  ==  expected_fc1_relu[j]
   For each j in 0..9:
       check  dut.out_regs[j]    ==  expected_fc2[j]
5. Print "ALL CHECKS PASSED" or "FAILED with N errors"
```

The key move is step 4: `dut.hidden_mem[j]` reaches *inside* the simulated circuit to read its internal memory directly. If even one value is off by a single bit, that exact neuron's index is printed in the failure message — pinpointing the bug.

> Without Test 2, you boot the board, the corpus accuracy is 14 %, and you have to debug the bus + driver + TCP + SystemVerilog *all at once*. Test 2 lets us guarantee the SV is correct before any hardware ever powers on.

Part 7 · Wrap up

Slide 23 — Summary: what changed from sequential to parallel

	Sequential design	**Parallel design (this presentation)**
Multipliers in FC1	1	**64 (in parallel)**
Multipliers in FC2	1	**2 (in parallel)**
Cycles per inference	~204 000	~**4 500**
Wall time per inference @ 50 MHz	4 ms	~**90 µs** (FC1 phase: 60 µs)
Resource usage	3 % ALM / 9 % DSP / 26 % M10K	3 % ALM / 9 % DSP / 26 % M10K (the parallelism cost essentially zero extra logic — re-banking the existing M10K bits was the win)
Speedup	1×	~**45**×
Hardware accuracy on MNIST test set	(sim only)	**96.53 %** (n=10 000)

Slide 24 — Where to dig deeper

- Full written report: [`FINAL_REPORT.pdf`](FINAL_REPORT.pdf)
- Source code: [`hw/mnist_accel.sv`](hw/mnist_accel.sv)
- Test corpus and results: [`project/eval/`](eval/)

Slide 25 — Acknowledgements

Course / dataset. DE1-SoC, Quartus tooling, lab template, and HPS Linux image courtesy of **CSEE 4840 (Prof. Stephen A. Edwards)**. The MNIST dataset is from **LeCun, Cortes & Burges (1998)**. The `hex7seg.sv` decoder is reused unchanged from Lab 1.

AI assistance. Substantial portions of the *non-RTL* surface — Python training pipeline, SystemVerilog testbench, Linux kernel driver, userspace clients (`draw.py`, `mnist_server.c`, `test_corpus.py`), build scripts, this slide deck, the report, and the design doc — were drafted, iterated, and debugged with **Anthropic Claude (Opus 4 / 4.6 / 4.7, claude.ai/code)**. All AI-generated artefacts were read, edited, and validated by the authors against the bit-exact testbench and the 10 000-image hardware-in-the-loop test. The RTL design decisions and bring-up debugging were owned by the authors.

Open-source tools. Quartus Prime Lite 21.1 · ModelSim ASE 18.1 · PyTorch 2.x · NumPy · Pillow · pypdfium2 · tectonic · mermaid.ink.

Slide 26 — Questions?

Things we found surprising during this project, in case you want to ask:

- How much of the design effort was **debugging the load path**, not the math.
- The Quartus M10K-vs-register inference fight (had to add `(* ramstyle = "M10K" *)` explicitly).
- The 2-cycle BRAM read latency causing a drain bug — found by the testbench in seconds.
- WSL/Quartus toolchain pain — a separate writeup if you're curious.
- Why we picked 64 banks for FC1 specifically: $128 / 64 = 2$ fits cleanly, and $2 \times 784 = 1\,568$ fits in one M10K block at 8 bits.
- The hand-drawn architecture diagram (Slide 12) vs the bit-annotated mermaid (Slide 13) — neither one alone tells the full story.