FPGA CNN Inference Project Plan (Virtual, Verilog + Verilator)

Core outcome: End-to-end implementation of a quantised CNN (MNIST digit recognition) inference pipeline in Verilog, fully simulated with Verilator, with Python used for training, quantisation verification, and simulated UART I/O. You will understand fixed-point math, HDL design, control flow (FSMs), memory organisation, and benchmarking, with a path to extensions like pipelining, parallelism, and alternative interfaces.

# Milestone Overview

Week | Major Deliverable  
Week 1 | Environment set up, learning Verilog basics, train & quantise CNN in Python, export weights  
Week 2 | Implement and test individual HDL modules: Conv, ReLU, Pool  
Week 3 | Implement Dense + Argmax; fixed-point pipeline verification; Python/Verilog comparison  
Week 4 | System integration FSM, end-to-end testbench, UART simulation  
Week 5 | Benchmarking (latency, throughput, accuracy), documentation draft  
Week 6 | Optional extensions: pipelining, parallel MACs, HLS comparison, alternative interfaces, synthesizable flow

# 1. Environment Setup

Concepts taught: Toolchain usage, build automation, file organisation, simulation workflow.  
  
Goals:  
- Get all tools installed and working.  
- Establish reproducible folder structure.  
- Be able to run Verilog simulations and Python prototypes end-to-end.  
  
Tools to install:  
- Verilator  
- GTKWave  
- Python 3.11+ with torch/tensorflow, numpy, matplotlib, pyserial, bitstring  
- Optional: make, git, editor with Verilog support  
  
Directory structure example:  
fpga\_cnn\_project/

├── python/                      # Training / quantisation / host scripts

│   ├── train.py

│   ├── quantise.py

│   ├── verify\_fixed\_point.py

│   └── uart\_sim.py

├── data/                        # MNIST samples, exported images

│   ├── mnist\_sample0.txt

│   └── ...

├── weights/                     # Exported quantised weights & biases

│   ├── conv1\_weights.mem

│   ├── conv1\_biases.mem

│   ├── fc1\_weights.mem

│   └── fc1\_biases.mem

├── hdl/                        # Verilog sources

│   ├── conv2d.v

│   ├── relu.v

│   ├── maxpool.v

│   ├── dense.v

│   ├── argmax.v

│   ├── uart.v

│   ├── top\_level.v

│   └── fsm\_controller.v

├── tb/                         # Testbenches and simulation harnesses

│   ├── tb\_conv2d.cpp           # Verilator C++ testbench wrapper

│   ├── tb\_full\_pipeline.cpp

│   └── uart\_tb.cpp

├── sim/                        # Simulation output, waveforms

├── docs/                       # Architecture diagrams, report drafts

└── Makefile                   # Build/sim helper

Example Makefile snippet:  
VERILATOR = verilator  
TOP = top\_level  
BUILD\_DIR = obj\_dir  
  
all: sim  
  
sim:  
 $(VERILATOR) --cc hdl/$(TOP).v --exe tb/tb\_full\_pipeline.cpp \  
 -Wall -Wno-fatal --trace  
 make -C $(BUILD\_DIR) -j --quiet  
 $(BUILD\_DIR)/V$(TOP) +vcd  
  
clean:  
 rm -rf $(BUILD\_DIR) sim/\*.vcd  
  
CLI examples:  
- Build & run full pipeline: make sim  
- View waveform: gtkwave sim/output.vcd

# 2. Learning Path (Parallel Track)

Concepts taught: Incremental mastery of Verilog, fixed-point, FSMs, and CNN components.  
  
Week 0–1: Verilog Fundamentals  
- Learn: modules, always blocks, assign, blocking vs non-blocking assignments, parameters, wires/registers, basic testbench.  
- Exercises: HDLBits exercises in this order:  
 1. Combinational logic (e.g., basic gates, multiplexers)  
 2. Sequential logic (flip-flops, counters)  
 3. Simple ALU  
 4. FSM implementation (traffic light, sequence detector)  
- Resources: HDLBits, Verilog by Example, ZipCPU’s Verilator guide.  
  
Week 1–2: Fixed-Point Arithmetic & CNN Basics  
- Learn: fixed-point representation (e.g., Q1.7, Q2.6), scaling, overflow handling, rounding/truncation.  
- Practice: implement fixed-point multiply and addition in Verilog and Python to compare.  
- Learn CNN inference primitives: convolution, ReLU, pooling, fully connected, argmax.  
- Resources: one-layer CNN forward-pass tutorials.  
  
Week 2–3: Control Flow and Memory  
- Learn FSM design for sequencing operations.  
- Learn memory organisation: modelling BRAM, double buffering, line buffering.  
- Resources: Verilator examples, memory tutorial.  
  
Week 3–4: UART & Host Interface  
- Learn UART frame structure, baud rate generation, and handshake.  
- Implement basic transmitter/receiver in Verilog and simulate Python-side sender/receiver.  
- Resources: UART protocol docs, PySerial tutorials.

# 3. Model Training & Quantisation

Concepts taught: CNN training, quantisation theory, fixed-point emulation, data formatting for hardware.  
  
3.1 Train CNN (Python)  
Minimal architecture example (PyTorch):  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from torchvision import datasets, transforms  
  
class SmallCNN(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.conv1 = nn.Conv2d(1, 8, kernel\_size=3, stride=1, padding=1)  
 self.pool = nn.MaxPool2d(2)  
 self.fc1 = nn.Linear(8 \* 14 \* 14, 10)  
  
 def forward(self, x):  
 x = F.relu(self.conv1(x))  
 x = self.pool(x)  
 x = x.view(x.size(0), -1)  
 x = self.fc1(x)  
 return x  
  
- Train to >90% on MNIST.  
- Save model weights.  
  
3.2 Quantisation to Fixed-Point  
- Choose representation (e.g., signed Q2.5 for activations, Q1.7 for weights).  
- Simulate quantised forward pass in Python:  
 def to\_fixed(x, frac\_bits):  
 return int(round(x \* (1 << frac\_bits)))  
  
 def from\_fixed(x, frac\_bits):  
 return x / (1 << frac\_bits)  
  
- Evaluate accuracy drop after quantisation.  
  
3.3 Exporting Weights/Biases  
- Flatten convolution weights to match Verilog access patterns.  
- Dense weights: flatten matrix row-major.  
- Save biases as vector.  
- Format example: hex or decimal in .mem file.  
  
3.4 Verification Script  
- Python script loads .mem weights and input, runs quantised inference, prints prediction.  
- Save reference output for HDL comparison.

# 4. HDL Design & Simulation

Concepts taught: Module decomposition, fixed-point hardware implementation, parameterisation, testbenches, BRAM, FSM control.  
  
4.1 Convolution Module (conv2d.v)  
Architecture:  
- Sliding window over input feature map.  
- Multiply-accumulate (MAC) with fixed kernel.  
- Support stride=1, optional padding.  
  
Fixed-point Strategy:  
- Inputs and weights in fixed-point.  
- Multiply produces extended width; accumulate carefully and shift back.  
  
BRAM/storage:  
- Store weights in BRAM; input streamed or line-buffered.  
  
Clocking/FSM:  
- States: IDLE -> LOAD\_WINDOW -> MAC -> WRITE\_OUTPUT.  
- Control signals: start, done, valid, weight\_addr.  
  
Implementation tips:  
- Parameterise kernel size, channels.  
- Use generate blocks for unrolling.  
  
Testbench:  
- Feed known input and weights; compare output with Python reference.  
  
4.2 ReLU Module (relu.v)  
- Comparator: if negative output zero.  
- Testbench: negative, zero, positive.  
  
4.3 MaxPool Module (maxpool.v)  
- 2x2 pooling: four comparators.  
- FSM for window load and output.  
  
4.4 Dense Layer (dense.v)  
- Matrix-vector multiply; add biases.  
- Fixed-point scaling.  
- Weights in BRAM.  
  
4.5 Argmax Module (argmax.v)  
- Reduction tree of comparators to get highest logit.  
- Output predicted class.

# 5. System Integration

Concepts taught: Hierarchical design, control FSM, buffering, sequencing, end-to-end flow.  
  
Top-Level FSM (fsm\_controller.v):  
Sequence: LOAD\_IMAGE -> CONV -> RELU -> POOL -> DENSE -> ARGMAX -> SEND\_RESULT.  
  
Buffering:  
- Intermediate outputs stored in BRAM or passed if pipelined.  
  
Clock gating / control:  
- Enable signals per module, optional advanced clock enables.  
  
Full Pipeline Testbench:  
- Load .mem input, run inference, capture output class, compare to Python.  
- Measure cycle count from start to finish.

# 6. UART Simulation

Concepts taught: Serial protocol modelling, interfacing host ↔ FPGA, latency emulation.  
  
UART Overview:  
- Frame: start bit, data bits, optional parity, stop bits.  
- Baud rate defines bit duration.  
  
Verilog UART Modules:  
- RX: sample serial line, reconstruct bytes.  
- TX: serialize byte with framing.  
  
Python Host Example:  
import serial  
ser = serial.Serial('/dev/ttyUSB0', baudrate=115200, timeout=1)  
with open('data/image.mem') as f:  
 pixels = f.read().splitlines()  
for px in pixels:  
 byte = int(px)  
 ser.write(bytes([byte]))  
result = ser.read(1)  
print('Predicted digit:', result[0])  
  
Latency & Handshake:  
- Implement ACK/NACK or ready/valid.  
- Simulate transmission delays with cycle timers.

# 7. Benchmarking & Analysis

Concepts taught: Performance measurement, accuracy validation, resource estimation, trade-off analysis.  
  
Latency:  
- Use cycle counter in testbench; convert to time given clock frequency.  
  
Throughput:  
- For pipelined version, measure initiation interval.  
  
Resource Estimation:  
- Manually count MACs, adders, memory usage; optionally run synthesis with Yosys.  
  
Accuracy Comparison:  
- Run N images through both Python and Verilog; compare predictions.  
  
Trade-offs:  
- Bit-width vs accuracy vs area vs latency.

# 8. Documentation

Include:  
- Title & Objective  
- System block diagram  
- Module breakdown  
- Training & quantisation explanation  
- Test strategy  
- Benchmark results  
- Design decisions  
- Bottlenecks & limitations  
- Extensions attempted  
- Future work

# 9. Optional Extensions (Advanced)

9.1 Pipelining:  
- Split conv into stages, insert registers, measure initiation interval.  
  
9.2 Parallel MAC Units:  
- Unroll loops for multiple output channels.  
  
9.3 Vivado HLS Comparison:  
- Implement conv/dense in C/C++ for HLS, compare against RTL.  
  
9.4 Alternative Interfaces:  
- Replace UART with SPI or Ethernet-like stub.  
  
9.5 Larger Models:  
- CIFAR-10 (3-channel, deeper networks). Adjust memory and compute accordingly.  
  
9.6 Synthesizable Flow:  
- Run through Yosys for rough area estimates; prepare for FPGA toolchain reports.

# Diagrams & Pseudocode Examples

Convolution Sliding Window Example:  
Input image:  
[ a b c d ]  
[ e f g h ]  
[ i j k l ]  
[ m n o p ]  
  
Kernel 3x3 over top-left:  
Window1:  
a b c  
e f g  
i j k  
  
Compute: sum = Σ (window\_element \* corresponding\_weight)  
  
FSM State Flow:  
IDLE -> LOAD\_IMAGE -> CONV -> RELU -> POOL -> DENSE -> ARGMAX -> SEND\_RESULT -> IDLE

# Suggested Resources

- HDLBits (verilog practice)  
- ZipCPU Verilator tutorials  
- Fixed-point arithmetic articles for FPGA  
- MNIST CNN forward-pass tutorials  
- UART protocol documentation  
- PyTorch quantisation guides  
- Verilator official examples

# Deliverable Checklist

- [ ] Environment installed and verified  
- [ ] CNN trained; baseline float model accuracy measured  
- [ ] Quantised model simulated in Python; accuracy recorded  
- [ ] Weights/images exported in .mem format  
- [ ] Verilog modules written for all layers  
- [ ] Individual testbenches with Python comparison  
- [ ] FSM controller implemented  
- [ ] Simulated UART interface  
- [ ] End-to-end simulation working  
- [ ] Benchmark metrics collected  
- [ ] Report drafted  
- [ ] Optional extensions attempted