Development Journal: DeepSpeech-Inspired Hardware Accelerator

Author: Vinod Kumar Bandela

Time Period: From Hat XOR Toy Design to LSTM-Based Neural Accelerator (March 2025 - June 2025)

Phase 1: Conceptualization through Hat XOR Toy Design

Objective:
Begin experimenting with pattern detection and sequence learning at a very primitive level using XOR/XNOR logic to compare characters and detect patterns.

Design Flow:

- Chose a simple word pattern like "hello" to detect using XNOR operations.
- Represented letters as binary values and tried shifting a window over the data stream.
- Implemented parallel XNOR gates for character-level matching.

Struggles & Fixes:

- Initial implementation sliced the input stream incorrectly. Pattern matching failed when characters overlapped across slices.
- Realized the need for a sliding window approach instead of fixed chunk processing.
- Converted to a window-based design that matched each bit pattern against known characters.

Insights:
This early work on XNOR and matching acted as a mental model to understand how a neural layer could function at a high level, reinforcing the need for a temporal memory mechanism.

Phase 2: Audio Processing and Bitstream Generation

Objective:
Simulate real-world audio input using .wav files and convert them to a usable format for testing hardware pattern detection.

Steps Taken:

- Used Python to read .wav files and perform envelope detection.

- Encoded the quantized outputs into 32-bit chunks.
- Printed the output to text files for use in Verilog testbenches.

Struggles:

- Initially tried to use raw amplitude directly but got erratic results.
- Introduced a low-pass filter and squaring function to extract energy envelope.

Alternative Path:
Shifted from raw audio to using MFCC features for more structured, meaningful input. This aligned better with DeepSpeech architecture.

Phase 3: Matrix Multiplication and ReLU Implementation

Objective:
Emulate a fully connected layer by multiplying MFCC inputs with weight matrices and applying ReLU activation.

Implementation:

- Defined mfcc_matrix[402][26] and weight_matrix[26][128].
- Performed dot products to get product_matrix[402][128].
- Applied ReLU: relu_matrix[i][j] = (product_matrix[i][j] < 0) ? 0 : product_matrix[i][j];

Bugs and Fixes:

- **Bug:** Signed multiplication overflowed in 16-bit representation.
  Fix: Extended bit width of intermediates to 2*N.
- **Bug:** ReLU misbehaved because the signed result wasn't being interpreted correctly in the negative range.
  Fix: Used correct signed comparisons.
- **Bug:** Warnings about $display in synthesizable code.
  Fix: Wrapped debug and pass/fail display logic in ` ifndef SYNTHESIS.

Experimentation:

- Tested with random MFCC data and known weights.
- Simulated using QuestaSim to validate dot product and ReLU functionality.

Phase 4: Integration of LSTM Gates

Objective:
Model an LSTM unit in hardware to capture temporal dependencies in speech.

Design:

- Reused the ReLU output as input to all LSTM gates: i, f, o, g.
- Approximated activation functions for sigmoid and tanh.

Approximations:

```
function automatic logic signed [2*N-1:0] sigmoid(input logic signed [2*N-1:0] x);
  if (x < -64) return 0;
  else if (x > 64) return 255;
  else return (x + 128) >> 1;
endfunction
```

Bugs and Fixes:

- **Bug:** Local logic declarations inside always_ff after procedural code.
  Fix: Moved gate variables outside loops.
- **Bug:** Used same ReLU input for all four gates.
  Fix (Planned): Future work includes adding independent weights per gate.

Testing:

- Validated the LSTM update logic.
- Checked c_t and h_t updates over rows.

- Used waveform viewer to inspect gate values, memory states.

Phase 5: FSM-Based Top Module Design

Objective:
Control the full process in a clean FSM-style hardware module.

States:

- IDLE: Wait for start
- LOAD: Load inputs
- COMPUTE: Perform multiplication + ReLU + LSTM
- DONE: Signal end of operation

Bugs and Fixes:

- **Bug:** Doing entire matrix multiply in a single clock cycle.
  Fix: Planned transition to index-driven FSM using counters i, j, k.

Enhancement Plan:

- Replace full nested loops with sequential i, j, k counters.
- Reduce resource usage and enable timing closure.

Phase 6: Self-Checking Testbench

Objective:
Validate output against expected results without manual waveform debugging.

Implementation:

- Randomly initialized weights and inputs.
- initial block to compare output matrix with placeholder expected.
- Printed pass_count, fail_count, and mismatch locations.

Bugs and Fixes:

- **Bug:** Used uninitialized expected values.
  Fix: Will later generate real expected values from software Python version.
- **Bug:** Failure to properly slice strings during file parsing.
  Fix: Improved index bounds and error handling with $sscanf.

Remaining:

- Implement golden reference model in Python.
- Generate actual expected LSTM outputs.

Conclusion & Future Work

This journey from simple XOR-based bit pattern recognition to a deep learning-inspired LSTM accelerator highlighted both the challenges and immense learning opportunities of hardware design.

Achievements:

- Custom Verilog pipeline for MFCC to LSTM sequence modeling.
- Self-checking testbench.
- Corrected major logic synthesis issues.

Next Steps:

1. Implement FSM-based sequential processing of matrix ops.
2. Add per-gate weights and biases.
3. Connect memory-mapped AXI bus for SoC integration.
4. Synthesize and test on FPGA (Intel/AMD).

Development Journal: DeepSpeech-Inspired Hardware Accelerator

Author: Vinod Kumar Bandela

Time Period: From Hat XOR Toy Design to LSTM-Based Neural Accelerator (March 2025 - June 2025)

This journal reflects the iterations, fixes, brainstorms, and bug hunts across every phase. Each phase represents an important pivot that informed the next, mimicking real-world design evolution.

Prepared by: **Vinod Kumar Bandela**
Last updated: **June 15, 2025**