Huffman Decoding

Jackson Konkin - jkonkin@uvic.ca - V00910699
Tyler Pires - tylercpires@uvic.ca - V00968314

Environment:

- **Model:** Macbook Air 2025
- **CPU:** Apple M4
    - Four 4.4 GHz performance cores
        - 512KB L1 data cache per core
    - Six 2.85 GHz efficiency cores
- **ARM:** ARMv9.2-A
    - https://developer.arm.com/documentation/ddi0602/2025-03?lang=en
    - Note: ARMv9-A is an extension of 8-A, which is an extension of 7-A
- **Compiler:** Clang (Apple clang version 15.0)
- **IDE:** Xcode 15
- **Memory:** LPDDR5X-7500
    - 120GB/s bandwidth
    - 768KB L1 instruction cache

Relevant Instructions:

1. **CLZ (Count Leading Zeros)**: Critical for determining bit lengths in variable-length codes. Single-cycle execution enables efficient logarithmic operations for tree traversal.
2. **NEON SIMD instructions**: Process multiple symbols in parallel using 128-bit vector registers. Key instructions include vld1q_u8 (parallel load), vshlq_u16 (vector shift), and vceqq_u16 (parallel compare).
3. **Bit manipulation instructions**: RBIT (reverse bits), BFI/BFM (bitfield insert/move), and barrel shifter integration for zero-cost shift operations during decoding.
4. **CSEL/CSNEG (Conditional Select)**: Replace unpredictable branches in variable-length decoding paths, avoiding branch misprediction penalties.

Optimizations:

1. **Cache-aligned lookup tables**: Decode tables organized to fit within L1 cache (512KB) with alignment to 64-byte cache lines.
2. **Fixed-size bit buffer management**: 64-bit buffers reduce refill frequency. No dynamic allocation ensures predictable memory access patterns.

3. **Register-resident local variables**: Frequently accessed decode state maintained in registers, avoiding memory round-trips.
4. **Parallel multi-stream decoding**: NEON instructions process 8-16 independent bit streams simultaneously for high-throughput applications.
5. **Branch-free decode loops**: Table-driven state machines and conditional select instructions minimize branch mispredictions.

Additional information:

- Coding alphabet is the English alphabet (A-Z), with the following symbol additions:
    - **! ? . , ; :**
- We are using Canonical Huffman Coding
    - Steps:
        - Count the frequency of each symbol in the input string.
        - Create the typical min-heap Huffman tree.
        - Assign code lengths (how many steps it takes from the root to reach each leaf respectively).
        - Sort by code length, and then by symbol.
        - Use the code lengths to encode each symbol.
    - Benefits:
        - We only have to send code lengths as opposed to traditional encoding requiring us to send the entire Huffman tree.
        - The codes in naive encoding are dependent on how the tree is constructed, meaning there's potential for different codes to be produced if you create a tree based on the same string twice. In canonical encoding, we always get the same codes.
        - Lookup tables can be used for decoding.

Questions:

1. Would SME (Scalable Matrix Extension) instructions provide benefits for our canonical Huffman implementation, or should we focus exclusively on NEON optimization?
2. For variable-length code extraction, is there a performance advantage to using BFI/BFM instructions versus traditional shift-and-mask operations on the M4?
3. Given our 32-symbol alphabet, would a two-level lookup table (first 8 bits + remainder) provide better cache utilization than a single 256-entry table?