HARDWARE IMPLEMENTATION OF THE HUFFMAN ENCODER FOR DATA COMPRESSION USING ALTERA DE2 BOARD

Ms.D.M.Kate
Department of Electronics Engineering
Bhagwati Chaturvedi College of Engineering
Nagpur, India.
dipalee_kate@yahoo.co.in

Abstract: This work will aim to describe hardware implementations of static and dynamic Huffman encoders written in VHDL. The flexibility of the design allows for hardware-based implementations using FPGAs. The proposed method will have the following properties: (1) high compression for test responses is expected because Huffman coding is well-known as a minimum block coding, (2) zero-aliasing compression can be achieved. (3) It would also reduce transmission time, storage space, translation table space and encoding times.

KEYWORDS: VHDL, FPGA

I. INTRODUCTION

The Huffman coding is a widely used data compression technique. A Huffman algorithm starts by assembling the elements of the alphabet each one being assigned a ‘weight’—a number that represents its relative frequency within the data to be compressed. It is based on the concept of mapping an alphabet to a different representation composed of strings of variable size such that symbols with a high probability of occurring have a smaller representation than those that occur less often.

II. DATA COMPRESSION

Data compression is the process of encoding information using fewer bits than an uncoded representation. Compression refers to reducing the quantity of data used to represent a file, image or video content without excessively reducing the quality of the original data. Compression refers to reducing the number of bits required to store and/or transmit digital media. To compress something means that you have a piece of data and you decrease its size.

A. Data compression methods

B. Block Diagram for Data Compression

C. Compressed representation overview

- A compressed data set consists of a series of blocks, corresponding to successive blocks of input data.
- The block sizes are arbitrary, except that non-compressible blocks are limited to 65,535 bytes.
- The Huffman trees for each block are independent of those for previous or subsequent blocks.
- Each block consists of two parts:
  - A pair of Huffman code trees that describe the representation of the compressed data part
  - A compressed data part. (The Huffman trees themselves are compressed using Huffman encoding.)
- The compressed data consists of a series of elements of two types:
1. Literal bytes.
2. Pointers to duplicated strings where a pointer is represented as a pair <length, backward distance>.
   - The representation used in the “deflate” format limits distances to 32K bytes lengths to 258 byte but does not limit the size of a block, except for uncompressible blocks, which are limited.
   - Each type of value (literals, distances, and lengths) in the compressed data is represented using a Huffman code, using
     - One code tree for literals and lengths
     - A separate code tree for distances.
     - The code trees for each block appear in a compact form just before the compressed data for that block.

D. Lossless Compression
In lossless data compression, the integrity of the data is preserved. The original data and the data after compression and decompression are exactly the same because, in these methods, the compression and decompression algorithms are exact inverses of each other: no part of the data is lost in the process. Redundant data is removed in compression and added during decompression. Lossless compression methods are normally used when we cannot afford to lose any data.

E. DEFLATE Compressed Data Format
   - For the implementation of Huffman encoder the DEFLATE Compressed Data Format can be used.
   - This DEFLATE Compressed Data Format specification defines a lossless compressed data format that compresses data using Huffman coding, with efficiency comparable to the best currently available general purpose compression methods.

The purpose of this specification is to define a lossless compressed data format that:
   - Is independent of CPU type, operating system, file system, and character set, and hence can be used for interchange;
   - Can compress or decompress a data stream to produce another data stream, using only an a priori bounded amount of intermediate storage, and hence can be used in data communications or similar structures such as Unix filters;
   - Compresses data with efficiency comparable to the best currently available general-purpose compression methods, and in particular considerably better than the “compress” program;
     - Can be implemented readily in a manner not covered by patents, and hence can be practiced freely;
     - Is compatible with the file format produced by the current widely used gzip utility, in that conforming
     - Decompressors will be able to read data produced by the existing gzip compressor.

III. HUFFMAN CODING
Huffman coding assigns shorter codes to symbols that occur more frequently and longer codes to those that occur less frequently. For example, imagine we have a text file that uses only five characters (A, B, C, D, E). Before we can assign bit patterns to each character, we assign each character a weight based on its frequency of use. In this example, assume that the frequency of the characters is as shown in Table 1.

<table>
<thead>
<tr>
<th>Character</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>17</td>
<td>12</td>
<td>12</td>
<td>27</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 1: Frequency of characters

Figure 3: Huffman coding

A character’s code is found by starting at the root and following the branches that lead to that character. The code itself is the bit value of each branch on the path, taken in sequence.

Figure 4: Final tree and code

IV. ENCODING
Let us see how to encode text using the code for our five characters. Figure 4 shows the original and the encoded text.
A. Literal encoding

The Huffman code for two alphabets are fixed, & are not represented explicitly in the data. The Huffman code lengths for the literal / length alphabet are:

Steps:
1. Scan the literal value from Range given in the table
2. Find the specific value to be added or subtracted
3. Convert result into specified bits from the table
4. Reverse this binary value.

<table>
<thead>
<tr>
<th>Lit values</th>
<th>Bits</th>
<th>codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 143</td>
<td>8</td>
<td>0011000 through 10111111</td>
</tr>
<tr>
<td>144 - 255</td>
<td>9</td>
<td>110010000 through 1111111111</td>
</tr>
<tr>
<td>256 - 279</td>
<td>7</td>
<td>00000000 through 0010111</td>
</tr>
<tr>
<td>280 - 287</td>
<td>8</td>
<td>110000000 through 11000111</td>
</tr>
</tbody>
</table>

Table 2: Huffman code length for the literal

Range 1: (14)b + (48)b → Reverse code in 8-bit binary form.
Range 2: (145)b + (256)b → Reverse code in 9-bit binary form.
Range 3: (257)b - (256)b → Reverse code in 7-bit binary form.
Range 4: (282)b + (8B)b → Reverse code in 8-bit binary form.

- The code lengths are sufficient to generate the actual codes, as described above.
- Literal / length values 286-287 will never actually occur in the compressed data, but participate in the code construction. Distance codes 0-31 are represented by (fixed-length) 5-bit codes, with possible additional bits.
- The distance codes 30-31 will never actually occur in the compressed data.

**Part 1:** For range 0-143:
Add binary of (48) D to the input binary data. Convert it into 8-bit binary & reverse it.

**Part 2:** For range 144-255:
Add binary of (256) D to the input binary data. Convert it into 9-bit binary & reverse it.

**Example:**
Data in → 01000001.
(48) D → 110000.
Code → data in + (48) D → 01110001.
Data out → 10001110.

B. Distance Encoding

Steps:
1. Scan the distance values from range given in the table
2. Distance start of that range is subtracted from input distance
3. This result is converted into specified number of bits
4. The distance code value is appended to the result
5. Both are parts are then independently reverse

<table>
<thead>
<tr>
<th>Extra</th>
<th>Extra</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Bits</td>
<td>Dist</td>
<td>Code Bits</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>9-12</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>17-24</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>25-31</td>
</tr>
</tbody>
</table>

Table 3: Distance Code

**Example:** Distance = '365'

- Start comparing the distance (365) with distance end from table-2.
- Stop when distance end (384) > distance 365.
Check the distance code value (16 here) from table and convert it into 5-bit binary form ("1 0000" here).

Read the number of extra bits (7 here) from table.

Extra bit value = distance - distance_start 
= 365 - 257 
= 108

Convert the extra bit value found out in above step into 7-bit binary form ("1 10000" here).

Code \(\rightarrow\) 10000_1101100
Data out \(\rightarrow\) 00001_0011011

C. Length Encoding

Encoded data blocks in the "deflate" format consist of sequences of symbols drawn from three conceptually distinct alphabets:

1. Either literal byte, from the alphabet of byte values (0 ... 255),
2. Length, backward distance pairs, where the length is drawn from (3 ... 258)
3. The distance is drawn from (1...8192).

In fact, the literal and length alphabets are merged into a single alphabet (0 ... 285)

Where values 0 ... 255 represent literal bytes

Value 256 indicates end-of-block

Values 257 ... 285 represent length codes

<table>
<thead>
<tr>
<th>Extra</th>
<th>Extra</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Bits</td>
<td>Length</td>
<td>Code Bits</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td>....</td>
</tr>
<tr>
<td>257</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>268</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>269</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>269</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>269</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>269</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>269</td>
<td>1</td>
<td>11,12</td>
</tr>
<tr>
<td>269</td>
<td>1</td>
<td>13,14</td>
</tr>
</tbody>
</table>

Table 4: Length Code

Steps:

1. Scan the length values from range given in the table (eg.14).
2. Each range is assigned a specific code
3. Take the particular Huffman code from the matched range (266)
4. Take a binary value of this code from table 1 (0001010)
5. Reverse this binary value and Append the extra bits (0101000_1)

The extra bits should be interpreted as a machine integer stored with the most-significant bit first, e.g., bits 1110 represent the value 14.

Example: length = 14

Check the code value for length = 14 from table 1 (here 266).

Find the Huffman Code for 266 from table 1. It is (here "0001010").

Find the no. of extra bits to be added for length 14 from table no 3 (here 1).

Code \(\rightarrow\) 0001010_1
Data out \(\rightarrow\) 01010001.

V. SYNTHESIS AND SIMULATION RESULTS

A. Block Diagram obtained on synthesis

B. RTL Schematic
C. Technology Schematics

D. Output Waveforms

E. Results after dumping the code into

ALTERA DE2 BOARD

(cyclone - ii FPGA device)

REFERENCES

1) FPGA-Based Lossless Data Compression using Huffman and LZ77 Algorithms Suzanne Rigler Department of Electrical and Computer Engineering University of Waterloo Waterloo, Ontario, Canada

URL: http://www.ietf.org/rfc/rfc1951.txt

