University of Victoria
Faculty of Engineering
SENG440 Embedded Systems: Final Project Report
Summer Term, 2021

French-Character Huffman Decoding in C: Optimizing for ARM920T

Daniel Kinahan, V00887329
Department of Computer Science, University of Victoria
Email: danielkinahan@uvic.ca

Raphael Bassot, V00896541
Faculty of Software Engineering, University of Victoria
Email: rbassot@uvic.ca

To: Dr. Mihai Sima, Associate Professor & Computer Engineering Program Director, University of Victoria
Email: msima@ece.uvic.ca

Submission Date: ____________________August 15, 2021_____________________

Table of Contents

1.0 – List of Figures and Tables

2.0 – Introduction

Compression of data files has been a topic of research for many years, mostly due to the limitations of finite hardware storage in computers, and data transmission time for such files. In 1952, David A. Huffman, an MIT student, developed a compression algorithm in his paper titled “A Method for the Construction of Minimum-Redundancy Codes” [1] that is known today as Huffman Encoding & Decoding. This coding algorithm is a type of optimal, prefix-free encoding that provides lossless data compression for files. Huffman coding is extremely useful, as files retain their exact data prior to encoding, with no loss of bytes. Huffman coding is used for JPEG and PNG image formats, as well as in many popular compression tools today such as GZIP.

This project consisted of creating an optimal Huffman decoder in C to be executed against a test bench of normally distributed UTF-8 symbols from a predetermined French-character alphabet (Appendix B). Goals for the project included: creating an optimal, working implementation of Huffman decoding in C, improving the performance of the decoder by optimizing the C code around the ARM device architecture, and exploring the benefits of Huffman coding as well as ways to further improve the project. First, the theoretical background of Huffman encoding and decoding was studied, and an optimal method of generating prefix-free codes was explored prior to implementation. After first implementation, the decoder was optimized in C around the architecture of the ARM920T v4l device, a 32-bit hardware device with a dedicated ARM instruction set and caching support. An encoder was also built for the purposes of dynamically creating optimal Huffman codes based on the frequency of occurrence of symbols in the input file, as well as encoding the input file. The decoder was built to perform two tasks – building the lookup table data structures dynamically, and decoding the Huffman-encoded file sequentially. The decoding program was cross-compiled from C into an ARM executable file using the *arm-none-linux-gnueabi-gcc* compiler, version 4.3.2, available on the UVic SENG440 lab machines. With a working program on the ARM device and optimizations complete, metrics were taken towards the instruction count and cache misses of the decoder (Appendix C) through *Valgrind*, a software analysis tool. Following this, to further the team’s understanding of the project, the team looked past a purely software design and analyzed the benefits of leveraging a custom hardware lookup table for the decode process. In conclusion, closing remarks were made towards the work completed, methods used for decoding, and the metrics collected and their significance. Also discussed was the future work that could be done, barring no time constraints for the project, and a further desire for the team to explore the potential of Huffman coding.

3.0 – Theoretical Background

Huffman coding solves the modern-day compression problem by providing a fast, efficient, and lossless method of compressing files. This section will discuss the theory behind Huffman encoding/decoding, as well as the basic architecture of the ARM device utilized for the project.

3.1 – The Theory of Huffman Coding

Huffman coding is an algorithm that utilizes key features to achieve maximum, lossless compression. The encoding step encodes files into variable-length binary codes, where these codes are generated for symbols based on the frequency of their appearance in the file. Shorter codes (with less bits) are given to more frequently occurring symbols, which improves compression performance and maximizes the amount of information stored per Huffman code. A table of these codes is generated during the encoding process, and becomes a supplement to the encoded file that is needed during decoding (typically, this table can be appended to the encoded file). In the simple example below, the string *"abcdaabaaabaaa"* yields the following optimal Huffman code table:

Code	Data
0	a
10	b
110	c
111	d

Figure 1. Example Huffman code table for the string *"abcdaabaaabaaa"*.

It is clear in the sample string that the letter '*a*' appears more often than any other letter, having a total of nine occurrences. For this reason, it is granted the shortest code, '0', which has a length of only 1 bit, unique from every other code in the table. To determine the effectiveness of the sample encoding above, the amount of bits required to encode the string can be calculated. The encoding would be as follows, '010110111001000010000', which sums to a total of 21 bits used, assuming a binary file. Prior to encoding, assuming a string containing 8-bit Regular

ASCII values, the string (without encoding) requires: 14 letters x 8 bits/letter = 112 bits in total to store this string into a file (which includes redundant bits based on the ASCII standard). Therefore, this encoding example comprises of only 18.75% of the initial space required by the raw string, obtaining a compression ratio of 5.333 and a space savings of 81.25%. Huffman compression excels when the frequency of specific symbols increases opposed to their peers.

The code table is generated by creating a binary tree, based on the frequency of occurrence of individual symbols. Every symbol appearing in the encoded file is placed into a leaf node in the tree, as seen in Figure 2 for the previous example. The location of leaf nodes directly depends on the frequency of occurrence of a symbol, where more commonly-occurring symbols reside higher up on the tree (closer to the root node). This structure provides each node with a unique and prefix-free code by traversing the path from the root of the tree to any leaf node. Following the example in Figure 2, the code for character *'b'* is found by taking the left branch from the root, then the right branch from that node, obtaining the code '10' for the symbol. In practice, this binary tree is generated by maintaining a minimum heap, and then dynamically inserting intermediate nodes to push symbol nodes down to the leaves of the tree.

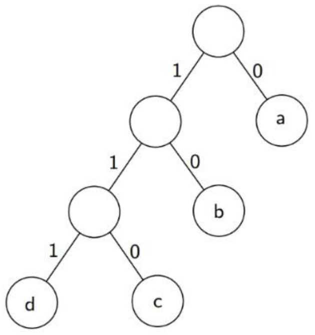


Figure 2. Example Huffman binary tree, that generates the Huffman codes for each symbol in a string.

Huffman decoding takes a different approach and assumes that a code table has previously been generated for the encoded file and is available to the decoder. Decoding leverages the code table to uniquely determine which symbol is being analyzed at any one time throughout the decoding process. Codes are required to be prefix-free to obtain unambiguous decoding – analyzing a variable-length code must uniquely determine what the algorithm does next. The encoded file is represented by its bit string of Huffman codes. Starting from the leftmost bit, the decoder continually shifts off one bit from the bit string, and analyzes the temporary code. If the temporary code is found in the code table, then the respective symbol for that code is returned (and written to a file, for example), and the temporary code is discarded. Otherwise, the decoder now shifts another bit off the bit string, and appends it to the temporary code, increasing its length. The new temporary code is analyzed again against the code table, and the process continues.

Huffman decoding is a sequential process that requires the previous code to be decoded prior to the following code. Optimizations for this algorithm appear in the implementation, such as reducing the retrieval time of codes in the table and minimizing the amount of table lookups performed during execution. The goal of this project is to create a Huffman decoder that successfully decodes any encoded file in a lossless manner, and whose operations are optimized with respect to the project's ARM machine – the 920T. Average codeword transmission rate, average amount of information per symbol, average time complexity of decoding operations, and minimal assembly code line count were the metrics in focus for the project. A further section will discuss how decoding was optimized for the project, as well as how results were obtained for the stated metrics.

3.2 – The ARM920T v4l

The machine used for the project is an ARM-based machine, accessed remotely and hosted by UVic servers. An ARM920T v4l (revision 0) was provided by the teaching team. This ARM, a member of the ARM9TDMI family of microprocessors, was a Harvard architecture device implementing a classic five-stage instruction pipeline. It featured a 32-bit hardware architecture and support for the 32-bit ARM instruction set, which offered the basic operations needed for a Huffman decoder. The 920T specifically was built with an embedded memory-management unit (MMU) to handle dedicated memory and cache operations, and optimized data address translations. The cache support for the 920T consists of virtually-addressed 64-way associative cache for two separate 16KB caches – the instruction cache and the data cache. 16 total registers (each 32-bit) are available within the architecture; however, many are used for machine-dedicated purposes and are not available for program use. A detailed block diagram of the ARM920T device and available peripherals can be seen in Appendix D.

4.0 – Design Process

4.1 – Designing the Huffman Decoder with Lookup Tables

The Huffman code table that is generated upon encoding is an essential piece to the decoding puzzle and is heavily utilized during the Huffman decoding process. A focus point of our project was to design a lookup method that minimizes the complexity of fetching the symbol associated to a given Huffman code. Lookup tables (LUT) were determined to be the most efficient method of resolving these code-symbol pairs.

The lookup table is an array-based data structure that provides excellent search capabilities by "replacing a runtime computation with a simpler array indexing operation" [3]. For decoding, the LUT was built as a 2-column table, with the symbol in the first column, and the length of the symbol's code in the second column. Each entry was placed into an array index that corresponded to the decimal value of the symbol's binary Huffman code. For example, if a symbol *'b'* was assigned a Huffman code of '10', then symbol *'b'*, along with its code length of 2, could be found at exactly index 3 of the LUT. This appears to work extremely well in practice, except for one critical factor – the encoded Huffman string is currently being read one bit at a time. The issue arises when a set of shifted bits from the bit string does not appear in the lookup table, as it does not yet correspond to the full binary code of an alphabet symbol. What happens to our lookup operation when our temporary code being analyzed is '1', and not yet '10'? Further work must be done to account for this.

Essentially, the above issue represents a code that follows the path of the binary tree of Huffman codes, but is not yet at a leaf node, where a symbol can be resolved. The nature of the variable-length codes is the inherent cause – it cannot be determined if the next symbol to decode will be of length 1, 2, or more bits. The solution to this is to modify the decoder. With the 2-column LUT described above, a code's length is now available to the decoder on every lookup operation. Instead of shifting off one bit at a time, the decoder is built to shift off a constant amount of bits from the encoded bit string. This constant (Huffman code width, named HWIDTH in the program) is assigned in the program and customized based on the input alphabet of the encoded file. Its implementation will be discussed later. The temporary code that is reassigned on every shift operation of the bit string is now always of constant length, HWIDTH, and it's binary values range from 0 (code '000…') to 2^{HWIDTH} (code '111…'). Logically, HWIDTH needs to be large enough for the largest code existing in the code table, so that every symbol can be resolved by continuously grabbing bit string portions of length HWIDTH. Now, assuming a symbol has a code of length exactly HWIDTH, the temporary code can be looked up in the LUT in one array indexing operation, requiring constant time. But what happens with our above example – symbol *'b'* – assuming that its code length of 2 is less than the defined

HWIDTH? The code length field must be used, and lookup table needs to be modified to be able to resolve smaller codes.

For example, assuming an HWIDTH of 4, should the code shifted off the bit string be ‘1011’, the decode should still be able to determine the code being analyzed and therefore the current alphabet symbol. The Huffman codes for this project are generated to be prefix-free, which defines each code to have a unique code that does not appear as a prefix to any other code in the code table. Therefore, grabbing any code of length HWIDTH ensures that our decoder can know which code is being analyzed – exactly the first one it finds while it reads bits from the left. Instead of iterating through the temporary code in search of the first Huffman code hit, the lookup table is modified. The LUT is built to have 2^{HWIDTH} indices. Upon filling the lookup table, a symbol’s Huffman code is analyzed, as well as its length. The Huffman code’s binary value is then compared to the binary prefix (leftmost ‘n’ bits) of each index in the code table, and should this prefix match, the table is filled with that symbol & code length pair. For example, a symbol with a Huffman code of ‘10’ would match a table index of ‘101000’, corresponding in decimal to the index 40, where it would then be slotted into the table at this index. An example of what a table would look like can be seen in Figure 4, which corresponds to the example alphabet analyzed previously in Figure 1.

Index	Index (binary)	Code	Bit required
0	000	a	1
1	001	a	1
2	010	a	1
3	011	a	1
4	100	b	2
5	101	b	2
6	110	c	3
7	111	d	3

Figure 3. Example Lookup Table for the Alphabet in Figure 1.

In this implementation, there are duplicate entries that take up space in the table. Symbol *‘a’* above can be found at four different indices, as its Huffman code of ‘0’ is found in the binary prefix of numbers 0 through 3. Therefore, our table of length 2^{HWIDTH} is larger than the number of entries needed to contain all Huffman codes for the alphabet. However, this storage trade off is largely worth the ability to obtain constant time lookup for every decoded symbol. With these modifications, the decoder grabs the next set of bits of length HWIDTH, and looks up the full HWIDTH-length code in the LUT. The value found at that index will always correspond to the correct symbol, as the index's prefix will always match the symbol's Huffman code. The symbol is returned, and then the temporary code has its leftmost bits truncated by exactly the code length of the symbol just decoded. Because each code is variable in length, the bit string must be shifted by the correct number of bits to avoid losing any bits for other symbols during the truncation step. With these modifications to the data structures and the decoder, constant time lookups have been obtained; the encoded bit string of length N can be decoded (ignoring overhead) in exactly N amount of constant-time lookup operations.

4.2 – Creating the Huffman Decoder in C

The Huffman decoder and lookup table building were programmed in C. Two main functions drive the decoding process: build_lookup_tables(), and huffman_decode(). The program accepts two files as input – a txt.lut file and a txt.huf file – that are both generated from the encoding program. The .lut file (Appendix B), contains a list of symbol & Huffman code pairs on each line, with the first line in the file denoting the minimum and maximum length code in the table. The .huf file contains a single continuous bit string of characters, representing the encoded file. For the project, this string was generated as ASCII characters '0' and '1', and not actual bits, for easier handling. These files were read into buffers during program execution.

The lookup table was implemented as two partitions, one *main* table of width HWIDTH, and the other *extended* table of width MAXWIDTH, where MAXWIDTH was the length of the longest Huffman code in the alphabet. The smaller value of HWIDTH was selected to cover ~90% of all symbols seen throughout file decoding, with only 10% of symbols or less having to perform a secondary lookup into the extended LUT. Implementing the whole LUT as two pieces reduced the word-width of the primary lookup table, and the HWIDTH selection provided the main LUT with enough entries to minimize accesses to the extended table.

Filling the table by binary prefix, as discussed previously, required the use of logical gates to shift the bits into the correct locations for comparison. A bitmask was created for the purpose of isolating the N leading bits of an index, where N was the current code length:

```
bitmask = ((1 << HWIDTH) - 1) ^ ((1 << (HWIDTH - code_len)) - 1);
```

Following this, the bitmask was bitwise ANDed with both the array index, and the Huffman code itself. Then, the result of each operation was XORed to determine if a table entry was filled:

```
if(((i & bitmask) ^ (num_huff_code8 & bitmask)) == 0){
        LUT[i][0] = huff_letter;
        LUT[i][1] = code_len;
 }
```

The project requirements involved using a French character alphabet of UTF-8 characters, which are each two bytes long. Instead of storing these entire two bytes in the LUT, only the second byte – which was unique across all symbols in the encoding alphabet – was stored. Therefore, uint8_t type integers could be used for the set of entries in the main lookup table. Once a symbol was decoded, the decoding loop handled printing out the first byte of the UTF-8 character, which was constant among alphabet symbols.

Decoding was a logically easier process once the tables were properly built. The huffman_decode() function accepts, as parameter, the file pointer to the input file containing the encoded bit string. This string was read into a memory-allocated buffer based on the length of the input file, and a NULL character was appended to the buffer. For the decoding loop, to simulate shifting off the bits from the bit string, a single decoded_shift offset variable is maintained, and incremented based on the length of the Huffman code recently resolved. The next iteration starts at the offset value, which is at the beginning of the next encoded symbol in the buffer. Extra overhead is needed via temporary buffers to handle grabbing a fixed amount of bits from the bit string's buffer, and performing a secondary lookup in the rare case that the first lookup in the main LUT is not found. All UTF-8 characters that are resolved are printed with fprintf() to a new decoded file, by prepending the UTF-8's first byte value – decimal 195:

```
fprintf(output_fp, "%c%c", 195, decoded_letter);
```

4.3 – Optimizing in C for More Efficient and Resourceful Decoding

To minimize the execution time and memory footprint of our program, careful optimizations were performed to tailor the program to the ARM920T device. We have unrolled our loops to allow for processor multi-tasking. We have optimized cache access on the lookup tables. Loop initialization has been optimized, where we previously had 'i=0' we now have 'i^=i', which saves us CPU clock cycles by avoiding a STORE operation. Integer data types and variable footprints have been reduced to save memory, such as reusing the incrementing variable 'i'. These loop counters are unsigned int type to optimize for architecture's ARM instruction set, which uses 32-bit addition, according to page 3-9 of the Samsung S3C2440A reference manual, the model of ARM920T used for this project [6]. Wherever possible we have replaced multiplication and division with cheaper operations such as logical bit shifts.

5.0 – Performance Evaluations on the ARM920T v4l

5.1 – The Project Test Bench

A common test bench was created to follow the specifications of the project. An initial input text file consisting of 389 Unicode Standard, French UTF-8 characters, with a selected French alphabet consisting of 15 distinct French symbols (Appendix B). The symbols followed a normal distribution, with some characters having a much greater frequency of occurrence than others, which maximized the opportunity for the dedicated encoder to produce optimal Huffman codes by the methods previously discussed.

The decoding program, ‘decode_optimized.c’, was configured with an HWIDTH of 6 and a MAXWIDTH of 8, therefore creating a main lookup table of size 2^6 and an extended lookup table of size 2^8. Input to the Huffman decoder consisted of two files, each of which generated by the custom encoder created for this project. The .lut file (Appendix B) consisted of a first line describing the minimum and maximum size code seen in the code table, as well as the Huffman code table itself. The code table carried 15 entries each having two columns – one for a French symbol and the other for its optimal, prefix-free Huffman code – and was used to build the main and extended lookup tables dynamically. For the purpose of this project, the Huffman codes were left as ASCII characters for usability, instead of converting them to bits directly. The .huf file (Appendix B) consisted of the encoded input text file generated from the custom Huffman encoder, and represents the exact bit string to which the decoding process was applied. The output of the Huffman decoder program was exactly one file – the decoded text file – that was compared against the original French-character input file for exact duplication, representing a working lossless compression algorithm.

Testing was carried out remotely over SSH, using the ARM920T v4l device, supplied over intranet by SENG440 machines at University of Victoria. For compatibility reasons, some tests were performed outside the ARM, and will be noted in later sections.

5.2 – Cache Misses & Instruction Count Results of Optimized Huffman Decoding

After optimizing the C code for the ARM920T, some metrics were taken to determine how the changes benefitted the execution of the Huffman decoding program.

Valgrind is an open-source software distribution that provides many tools for analyzing code, such as memory management, cache and branch-prediction, and instruction counts of a function. The latter two tools – called *cachegrind* and *callgrind* – were executed on the original and optimized code, which were both tested against project test bench. Due to compatibility issues with the ARM device not having gcc or Valgrind installed, these tests were carried out on a separate Linux VM running an Intel x86-based processor. Therefore, the metrics obtained are estimations of the improvements made towards the ARM machine.

The test bench was executed first with *callgrind*. This tool provided insight on the amount of instructions used for the important function of the decoding code, 'huffman_decode', which handles the iterative step of resolving symbols from their Huffman encoding. Prior to optimizations, the original decoder performed 49,843 instructions towards the buffer loading overhead, symbol lookups and variable store operations, and printing symbols to the file (Appendix C). Once the C optimizations were applied as discussed in a previous section, callgrind was re-executed on the test bench for an improved result of 40,398 instructions, obtaining an 18.94% decrease of instructions performed for the decoding process. This minimization of instruction use is largely due to the reduced amount of operations performed by leveraging loop unrolling during the decode step (ie. less index incrementation operations), and performing operator strength reductions for branch comparisons (ie. XOR a variable instead of comparing equality to an ASCCI character) and loop initializations (ie. XOR a loop index with itself instead of reassigning it to zero). Because iteration is heavily relied upon for the nature of the sequential Huffman decoding process, these small changes displayed large benefits towards the overall minimization of the number of instructions performed by the CPU for decoding of the test bench scenario.

Cachegrind displayed how the decoding program was using its cache throughout execution, as well as the number of logical branches taken. The total amount of cache misses was reported, and improved upon for the optimized code. The original decoding program – including both lookup table building and Huffman decoding functions – obtained 611,355 cache references, with only 3,533 of them being expensive cache misses. Through optimization of the LUT data structure and performance of the 'build_lookup_tables' function, these metrics were improved. Commonly used variables, such as the index calculated for the lookup table, were initialized as 'register' variables to provide the opportunity for the compiler to keep such variables in main memory. The main lookup table was initialized as a 256 x 2 array, optimized for the ARM920T architecture that supports 16KB, 64-way associative cache, so that large chunks of the lookup table had the opportunity to be stored in nearby cache at once, as opposed to needing a fetch operation for symbol lookups. After optimizations, the improved decoding program resulted in 604,817 cache lookups, and an identical 3,533 cache misses. Furthermore, the branch count reduced from 109,832 in the original code, to 108,232 after optimizations

(Appendix C). These improvements were marginal, an unchanged with regards to the number of cache misses. No further possible C optimizations were identified by the project team. Therefore, modifying the assembly code or possibly implementing a custom hardware for lookup table operations (or leveraging a microprocessor with hardware lookup table support) would be the next step towards improving the number of cache misses that occur during decoding.

5.3 – Average Transmission Rates & Huffman Data Compression Results

An important metric towards the efficiency of Huffman coding is the comparison of average transmission rates. A codeword's transmission rate is equal to the amount of bits in its Huffman code, required to represent its respective symbol. The average transmission rate represents the mean of these rates for every Huffman code in the table, weighted by probability of occurrence. Using our example of the string *"abcdaabaaabaaa"*, the average transmission rate for the string with unoptimized Huffman coding applied can be calculated as:

```
Avg. Transmission rate = (2 bits needed for each symbol) * (probability of any
symbol) = 2 bits/cycle
```

Seeing as there are only four symbols in this alphabet, the naïve method of encoding would be to assign all permutations of 2-bit codes ('00', '01', '10', '11') to the symbols. Instead, This project implemented an optimal algorithm, with code table seen in Figure 1, that placed significance on the most frequently occurring symbols in the alphabet. The average transmission rate for the optimal Huffman encoding used for this project is calculated (in alphabetical order):

```
Avg. Transmission rate = sum[(bits needed per symbol i) * (probability of
symbol i occurring)]
      = (0.643×1 + 0.214×2 + 0.07×3 + 0.07×3) = ~1.49 bits/cycle
```

Therefore, with this small example string, the average codeword transmission rate is minimized within the program, and is below 75% of the original average rate. This provides a more efficient use of CPU resources during decoding. The amount of information per bit used in a Huffman code is maximized and therefore less bits can be used to represent the same set of symbols.

Data compression results were also of interest to the project team. The project test bench was used to calculate the difference between storage space required for the input file before and after encoding. Initially, the French-character input text file (Appendix B) consisted of 389 two-byte UTF-8 characters. This represents an uncompressed file size of:

```
(389 characters * 2 bytes/char) = 778 bytes = 6,224 bits
```

By using the project's Huffman encoder, the file size could be drastically reduced, largely due to the probability distribution of the symbols, the optimized table of Huffman codes, and the size of the two-byte French characters. The reduced file size was determined to be 1,164 bits in size, representing a theoretical maximum space savings of 81.30% (compression ratio of 5.347) against the project test bench. These results were obtained by encoding ASCII characters of '1's and '0's to the encoded file for usability purposes (Appendix B), which by itself does not represent a compression. Instead, this method was used to provide an accurate estimation of proper bitwise Huffman encodings. Outside the scope of this project, the potential margins of error to the efficiency of bitwise Huffman encoding include redundant padding bits due to C data type restrictions, and appending the Huffman code table to the encoded file. Additionally, the probability distribution of symbols influences the efficiency of the encoding, and must be taken into account for future explorations.

5.4 – Theoretical Hardware Lookup Table Improvements

In digital logic, a lookup table is typically implemented with a multiplexer whose select lines are driven by the address signal and whose inputs are the values of the elements contained in the array. These values can either be hard-wired, as in an ASIC whose purpose is specific to a function or provided by D-latches which allow for configurable values. In this case, the configurable LUT would be accessed *4n* times where *n* is the number of letters contained in the file.

Assuming there is a file of 1000 characters, and an average cache memory access time of 1.33µs [5] this file would take 5320µs just for memory access. If the team considers implementing the LUT in hardware using an FPGA, which can handle seven million 32-bit transactions per second for either read or write [4] operations via DMA, they will record an average access time of 0.1429µs. The entire file would take 571.6µs to translate through the LUT. This results in an 89% decrease in access time, improving efficiency significantly.

6.0 – Conclusions

This project for *SENG440: Embedded Systems* was successfully completed, as a working Huffman encoder & decoder for a French-character alphabet were created in C programming language (Appendix A), optimized in C for the ARM920T microprocessor, and executed against the test bench on the device with various metrics retrieved. The encoder algorithm proved to optimally compress input files, and the decoder program effectively minimized its memory footprint and execution time by using a constant-time lookup table for code & symbol pairs. The applied optimizations showed a minimization of the instruction count for the ‘huffman_decode’ function, which was a significant improvement. On the other hand, *cachegrind* revealed that there were minimal improvements to the number of cache misses occurring throughout execution of the project’s test bench, and only a small decrease in cache references.

Continuing beyond the scope of this project, the project team would like to explore a true bitwise Huffman encoding algorithm – instead of substituting ASCII characters – and push the capabilities of the C programming language. Furthermore, given the required time to do so, the project team would investigate the ARM assembly code, and try to minimize the register footprint & reduce LOAD/STORE operations within the assembly code. Lastly, due to the benefits that a dedicated hardware lookup table would bring to the project, the team can decrease the memory access time significantly to optimize further.

7.0 – Bibliography

[1] – D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Resonance*, vol. 11, no. 2, pp. 91–99, Dec. 1951.

[2] – “ARM920T Technical Reference Manual,” *Documentation – arm developer*. [Online].

Available: https://developer.arm.com/documentation/ddi0151/c/I1004722. [Accessed: 14-Aug-2021].

[3] – “Lookup table in data Structures TUTORIAL 15 AUGUST 2021 - Learn lookup table in data Structures Tutorial (7155): WISDOM Jobs India,” *Wisdom Jobs*. [Online]. Available: https://www.wisdomjobs.com/e-university/data-structures-tutorial-290/lookup-table-7155.html. [Accessed: 14-Aug-2021].

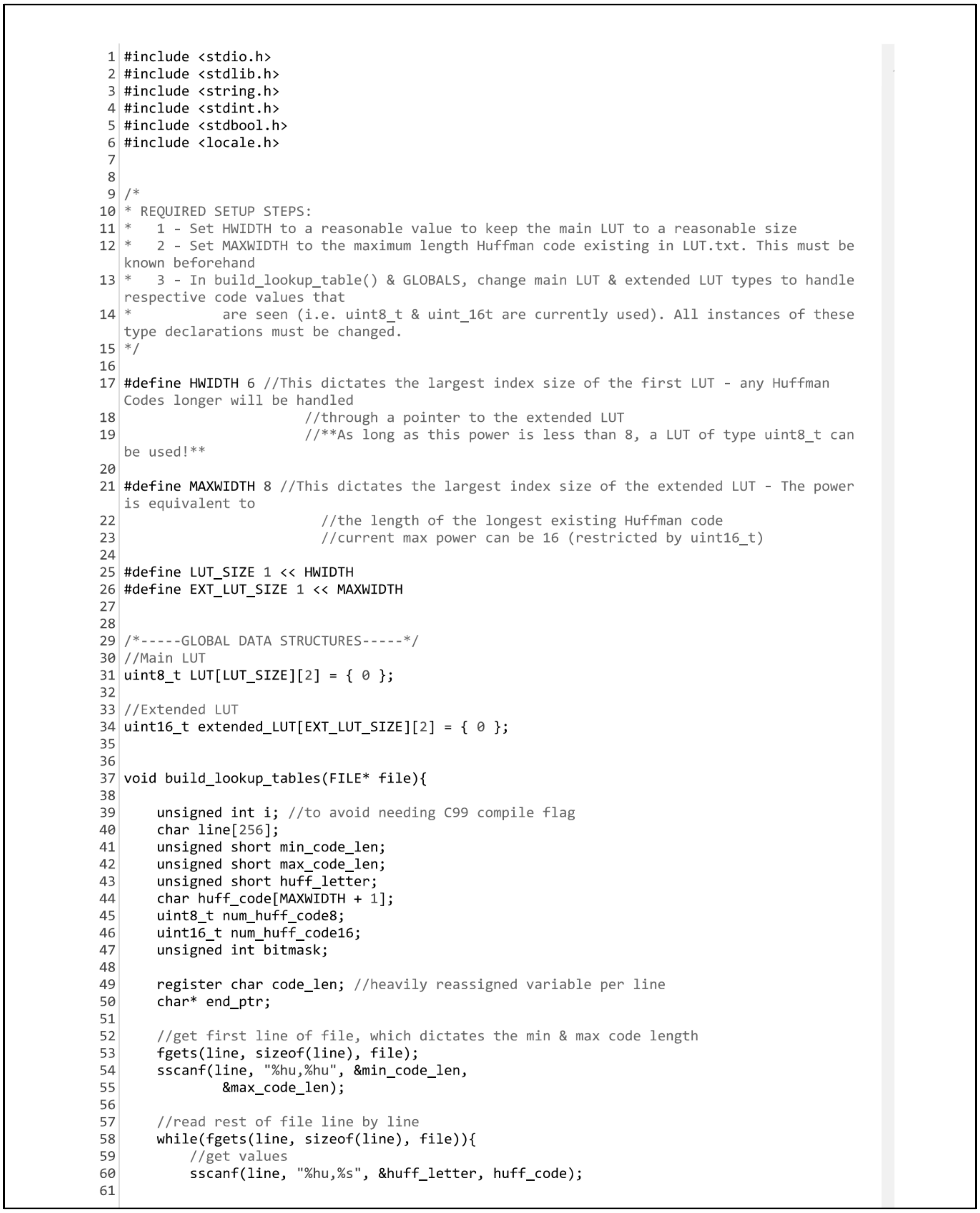
[4] – “DE1-SoC: ARM HPS and FPGA Addresses and Communication,” *FPGA/HPS communication*. [Online]. Available: https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/HPS_peripherials/FPGA_addr_index.html. [Accessed: 13-Aug-2021].

[5] – *Concept of direct memory Access (DMA)*. [Online]. Available: https://www.tutorialspoint.com/concept-of-direct-memory-access-dma. [Accessed: 13-Aug-2021].

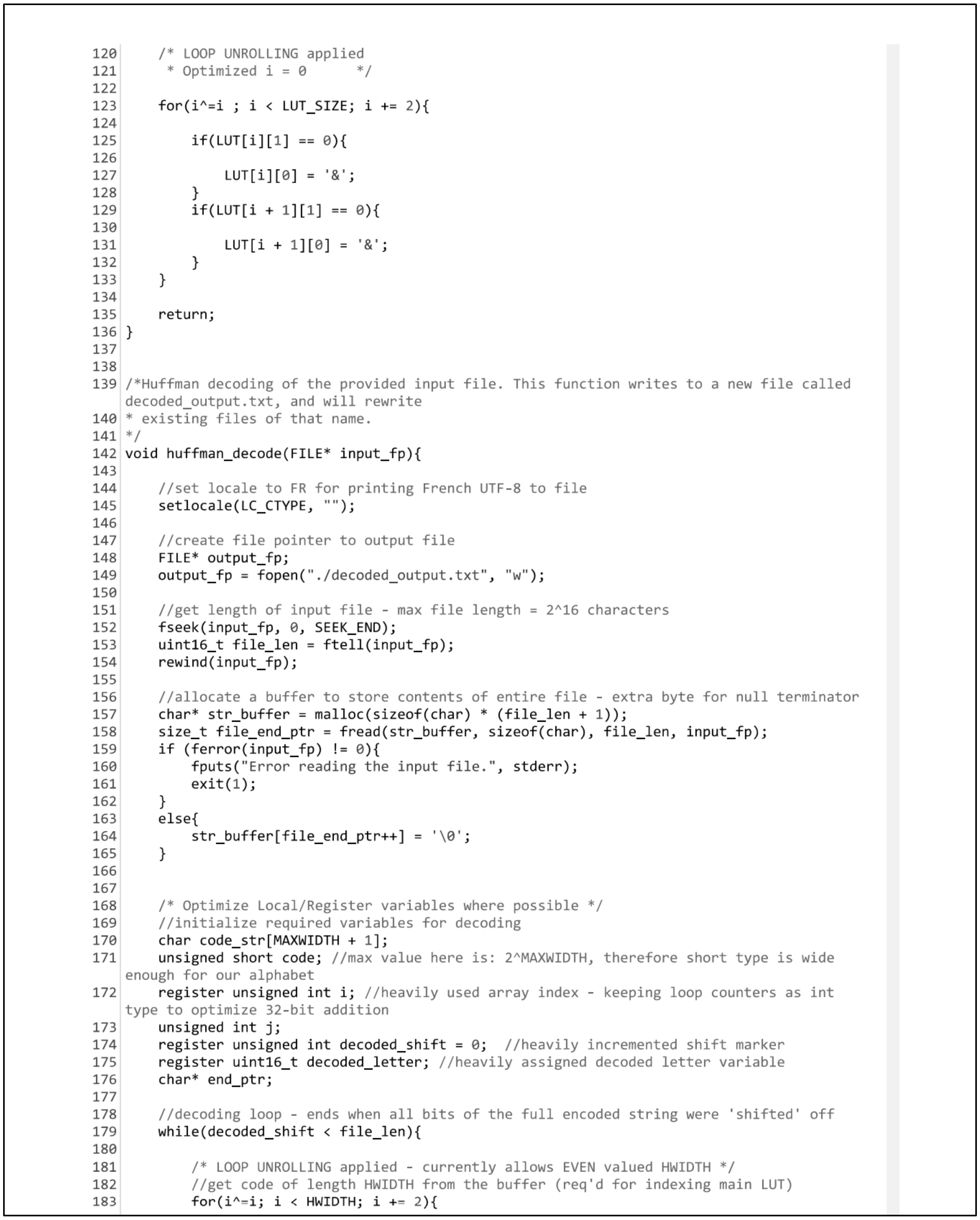
[6] – *S3C2440A 32-bit CMOS Microcontroller User’s Manual*, Samsung Electronics, Korea, August 2004. [Accessed: 13-Aug-2021].

8.0 – Appendices

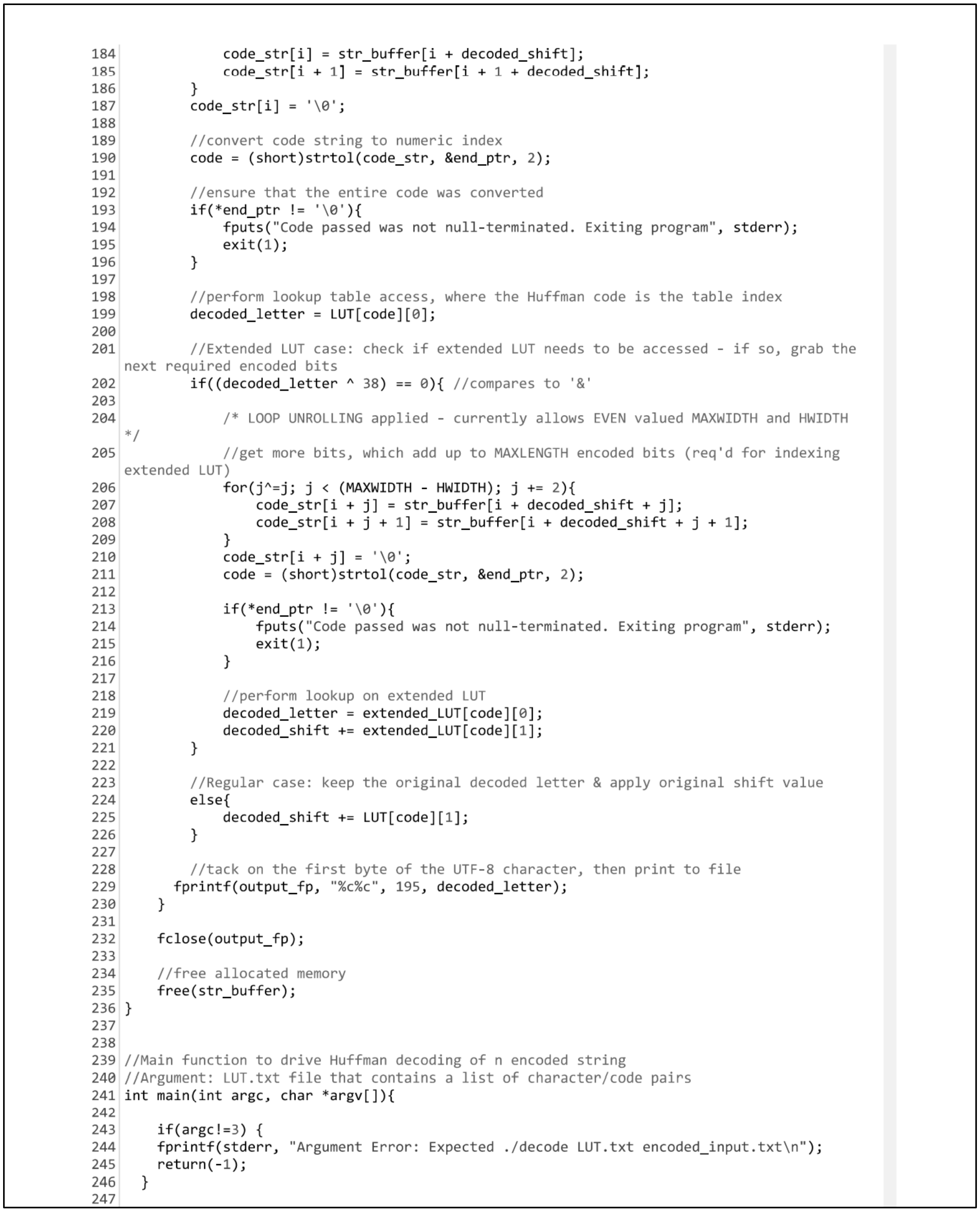
8.1 – Appendix A – decode_optimized.c



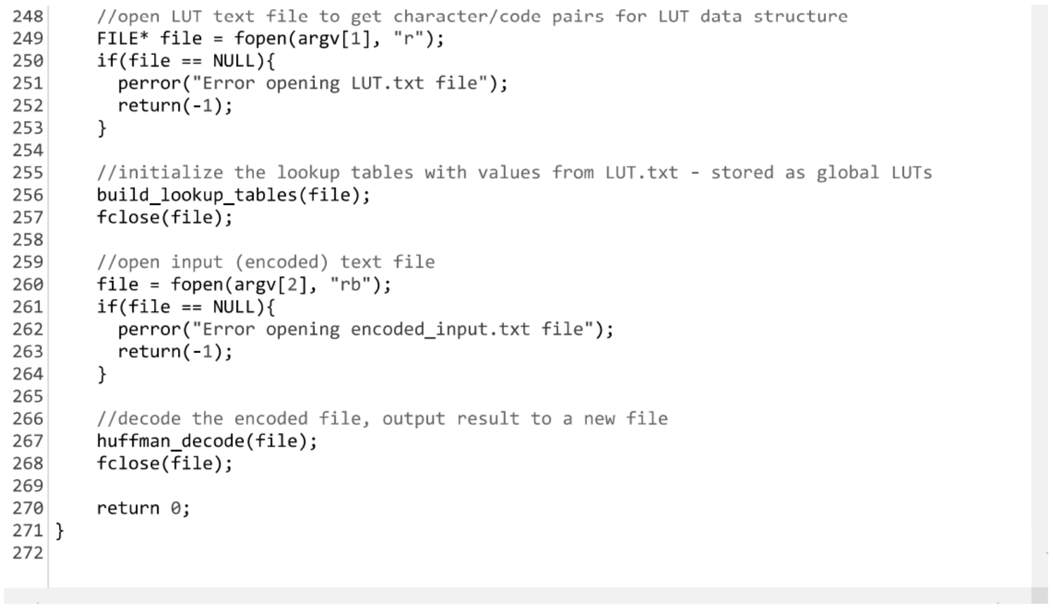
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>
#include <locale.h>


/*
* REQUIRED SETUP STEPS:
*   1 - Set HWIDTH to a reasonable value to keep the main LUT to a reasonable size
*   2 - Set MAXWIDTH to the maximum length Huffman code existing in LUT.txt. This must be known beforehand
*   3 - In build_lookup_table() & GLOBALS, change main LUT & extended LUT types to handle respective code values that
*           are seen (i.e. uint8_t & uint_16t are currently used). All instances of these type declarations must be changed.
*/

#define HWIDTH 6 //This dictates the largest index size of the first LUT - any Huffman Codes longer will be handled
                      //through a pointer to the extended LUT
                      //**As long as this power is less than 8, a LUT of type uint8_t can be used!**

#define MAXWIDTH 8 //This dictates the largest index size of the extended LUT - The power is equivalent to
                        //the length of the longest existing Huffman code
                        //current max power can be 16 (restricted by uint16_t)

#define LUT_SIZE 1 << HWIDTH
#define EXT_LUT_SIZE 1 << MAXWIDTH


/*-----GLOBAL DATA STRUCTURES-----*/
//Main LUT
uint8_t LUT[LUT_SIZE][2] = { 0 };

//Extended LUT
uint16_t extended_LUT[EXT_LUT_SIZE][2] = { 0 };


void build_lookup_tables(FILE* file){

    unsigned int i; //to avoid needing C99 compile flag
    char line[256];
    unsigned short min_code_len;
    unsigned short max_code_len;
    unsigned short huff_letter;
    char huff_code[MAXWIDTH + 1];
    uint8_t num_huff_code8;
    uint16_t num_huff_code16;
    unsigned int bitmask;

    register char code_len; //heavily reassigned variable per line
    char* end_ptr;

    //get first line of file, which dictates the min & max code length
    fgets(line, sizeof(line), file);
    sscanf(line, "%hu,%hu", &min_code_len,
            &max_code_len);

    //read rest of file line by line
    while(fgets(line, sizeof(line), file)){
        //get values
        sscanf(line, "%hu,%s", &huff_letter, huff_code);

        //get length of current Huffman code - for storage in the LUT
        code_len = strlen(huff_code);

        //Case 1 - slot character & code length pair into Main LUT
        if(code_len <= HWIDTH){
            //convert string code (binary representation) to number
            //then, left shift the huffcode (LUT index) until it is left aligned
            num_huff_code8 = (uint8_t)strtol(huff_code, &end_ptr, 2) << (HWIDTH - code_len);

            //slot character & code length into the table in all indices that have the given binary prefix
            bitmask = ((1 << HWIDTH) - 1) ^ ((1 << (HWIDTH - code_len)) - 1);

            /* LOOP UNROLLING applied - always works as our tables have an even number of indices (size is power of 2) */
            /* Cache Optimized LUT - accessing indices where first dimension of LUt is the largest dimension */
            for(i^=i; i < LUT_SIZE; i += 2){

                if(((i & bitmask) ^ (num_huff_code8 & bitmask)) == 0){
                    LUT[i][0] = huff_letter;
                    LUT[i][1] = code_len;
                }
                if(((i + 1 & bitmask) ^ (num_huff_code8 & bitmask)) == 0){
                    LUT[i + 1][0] = huff_letter;
                    LUT[i + 1][1] = code_len;
                }
            }
        }

        //Case 2 - slot character & code length pair into Extended LUT
        else{
            //convert string code (binary representation) to number
            //then, left shift the huffcode (LUT index) until it is left aligned
            num_huff_code16 = (uint16_t)strtol(huff_code, &end_ptr, 2) << (MAXWIDTH - code_len);

            //slot character & code length into the table in all indices that have the given binary prefix
            bitmask = ((1 << MAXWIDTH) - 1) ^ ((1 << (MAXWIDTH - code_len)) - 1);

            /* LOOP UNROLLING applied - always works as our tables have an even number of indices (size is power of 2) */
            /* Cache Optimized LUT - accessing indices where first dimension of LUt is the largest dimension */
            for(i^=i; i < EXT_LUT_SIZE; i += 2){

                if(((i & bitmask) ^ (num_huff_code16 & bitmask)) == 0){
                    extended_LUT[i][0] = huff_letter;
                    extended_LUT[i][1] = code_len;
                }
                if(((i + 1 & bitmask) ^ (num_huff_code16 & bitmask)) == 0){
                    extended_LUT[i + 1][0] = huff_letter;
                    extended_LUT[i + 1][1] = code_len;
                }
            }
        }
    }

    //iterate through main LUT to fill in pointer to extended LUT, wherever there currently exists no index
    //main LUT filling
    unsigned int prev_letter;
    unsigned int prev_code_len;
    unsigned int prev_code;

```



```c
    /* LOOP UNROLLING applied
     * Optimized i = 0      */

    for(i^=i ; i < LUT_SIZE; i += 2){

        if(LUT[i][1] == 0){

            LUT[i][0] = '&';
        }
        if(LUT[i + 1][1] == 0){

            LUT[i + 1][0] = '&';
        }
    }

    return;
}


/*Huffman decoding of the provided input file. This function writes to a new file called decoded_output.txt, and will rewrite
* existing files of that name.
*/
void huffman_decode(FILE* input_fp){

    //set locale to FR for printing French UTF-8 to file
    setlocale(LC_CTYPE, "");

    //create file pointer to output file
    FILE* output_fp;
    output_fp = fopen("./decoded_output.txt", "w");

    //get length of input file - max file length = 2^16 characters
    fseek(input_fp, 0, SEEK_END);
    uint16_t file_len = ftell(input_fp);
    rewind(input_fp);

    //allocate a buffer to store contents of entire file - extra byte for null terminator
    char* str_buffer = malloc(sizeof(char) * (file_len + 1));
    size_t file_end_ptr = fread(str_buffer, sizeof(char), file_len, input_fp);
    if (ferror(input_fp) != 0){
        fputs("Error reading the input file.", stderr);
        exit(1);
    }
    else{
        str_buffer[file_end_ptr++] = '\0';
    }


    /* Optimize Local/Register variables where possible */
    //initialize required variables for decoding
    char code_str[MAXWIDTH + 1];
    unsigned short code; //max value here is: 2^MAXWIDTH, therefore short type is wide enough for our alphabet
    register unsigned int i; //heavily used array index - keeping loop counters as int type to optimize 32-bit addition
    unsigned int j;
    register unsigned int decoded_shift = 0;  //heavily incremented shift marker
    register uint16_t decoded_letter; //heavily assigned decoded letter variable
    char* end_ptr;

    //decoding loop - ends when all bits of the full encoded string were 'shifted' off
    while(decoded_shift < file_len){

        /* LOOP UNROLLING applied - currently allows EVEN valued HWIDTH */
        //get code of length HWIDTH from the buffer (req'd for indexing main LUT)
        for(i^=i; i < HWIDTH; i += 2){
```

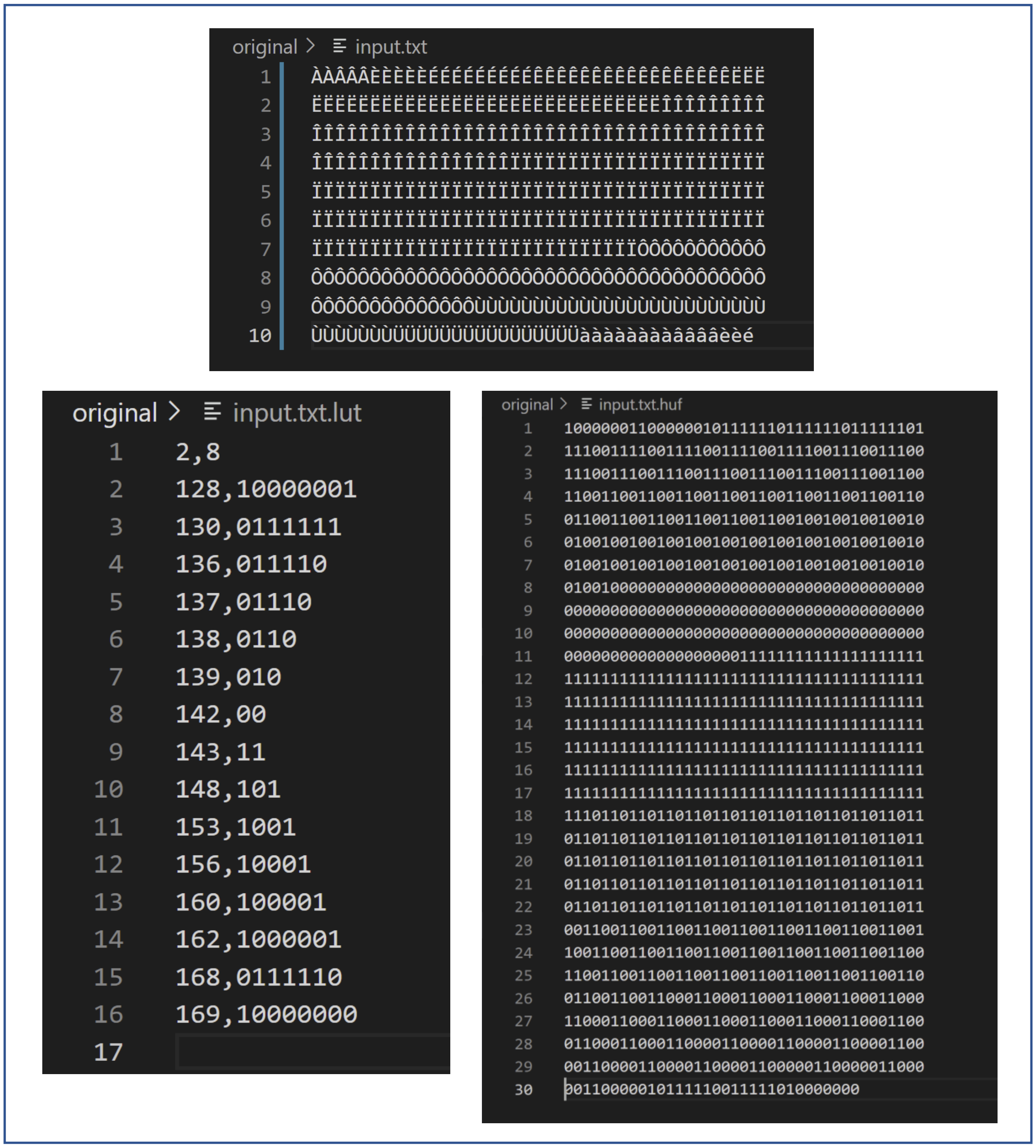


```c
            code_str[i] = str_buffer[i + decoded_shift];
            code_str[i + 1] = str_buffer[i + 1 + decoded_shift];
        }
        code_str[i] = '\0';

        //convert code string to numeric index
        code = (short)strtol(code_str, &end_ptr, 2);

        //ensure that the entire code was converted
        if(*end_ptr != '\0'){
            fputs("Code passed was not null-terminated. Exiting program", stderr);
            exit(1);
        }

        //perform lookup table access, where the Huffman code is the table index
        decoded_letter = LUT[code][0];

        //Extended LUT case: check if extended LUT needs to be accessed - if so, grab the next required encoded bits
        if((decoded_letter ^ 38) == 0){ //compares to '&'

            /* LOOP UNROLLING applied - currently allows EVEN valued MAXWIDTH and HWIDTH */
            //get more bits, which add up to MAXLENGTH encoded bits (req'd for indexing extended LUT)
            for(j^=j; j < (MAXWIDTH - HWIDTH); j += 2){
                code_str[i + j] = str_buffer[i + decoded_shift + j];
                code_str[i + j + 1] = str_buffer[i + decoded_shift + j + 1];
            }
            code_str[i + j] = '\0';
            code = (short)strtol(code_str, &end_ptr, 2);

            if(*end_ptr != '\0'){
                fputs("Code passed was not null-terminated. Exiting program", stderr);
                exit(1);
            }

            //perform lookup on extended LUT
            decoded_letter = extended_LUT[code][0];
            decoded_shift += extended_LUT[code][1];
        }

        //Regular case: keep the original decoded letter & apply original shift value
        else{
            decoded_shift += LUT[code][1];
        }

        //tack on the first byte of the UTF-8 character, then print to file
      fprintf(output_fp, "%c%c", 195, decoded_letter);
    }

    fclose(output_fp);

    //free allocated memory
    free(str_buffer);
}


//Main function to drive Huffman decoding of n encoded string
//Argument: LUT.txt file that contains a list of character/code pairs
int main(int argc, char *argv[]){

    if(argc!=3) {
    fprintf(stderr, "Argument Error: Expected ./decode LUT.txt encoded_input.txt\n");
    return(-1);
  }

```



```c
    //open LUT text file to get character/code pairs for LUT data structure
    FILE* file = fopen(argv[1], "r");
    if(file == NULL){
      perror("Error opening LUT.txt file");
      return(-1);
    }

    //initialize the lookup tables with values from LUT.txt - stored as global LUTs
    build_lookup_tables(file);
    fclose(file);

    //open input (encoded) text file
    file = fopen(argv[2], "rb");
    if(file == NULL){
      perror("Error opening encoded_input.txt file");
      return(-1);
    }

    //decode the encoded file, output result to a new file
    huffman_decode(file);
    fclose(file);

    return 0;
}

```

8.2 – Appendix B – Project Test Bench: Generated input.txt.lut & input.txt.huf



```
ÀÀÂÂÂÈÈÈÈÈÉÉÉÉÉÉÉÉÉÊÊÊÊÊÊÊÊÊÊÊÊÊÊÊÊÊÊËËË
ËËËËËËËËËËËËËËËËËËËËËËËËËËËËËËÎÎÎÎÎÎÎÎÎÎ
ÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎ
ÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÎÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏ
ÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏ
ÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏ
ÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÏÔÔÔÔÔÔÔÔÔÔÔ
ÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔÔ
ÔÔÔÔÔÔÔÔÔÔÔÔÔÔÙÙÙÙÙÙÙÙÙÙÙÙÙÙÙÙÙÙÙÙÙÙÙÙÙÙ
ÙÙÙÙÙÙÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜÜàààààààààââââèèé
```

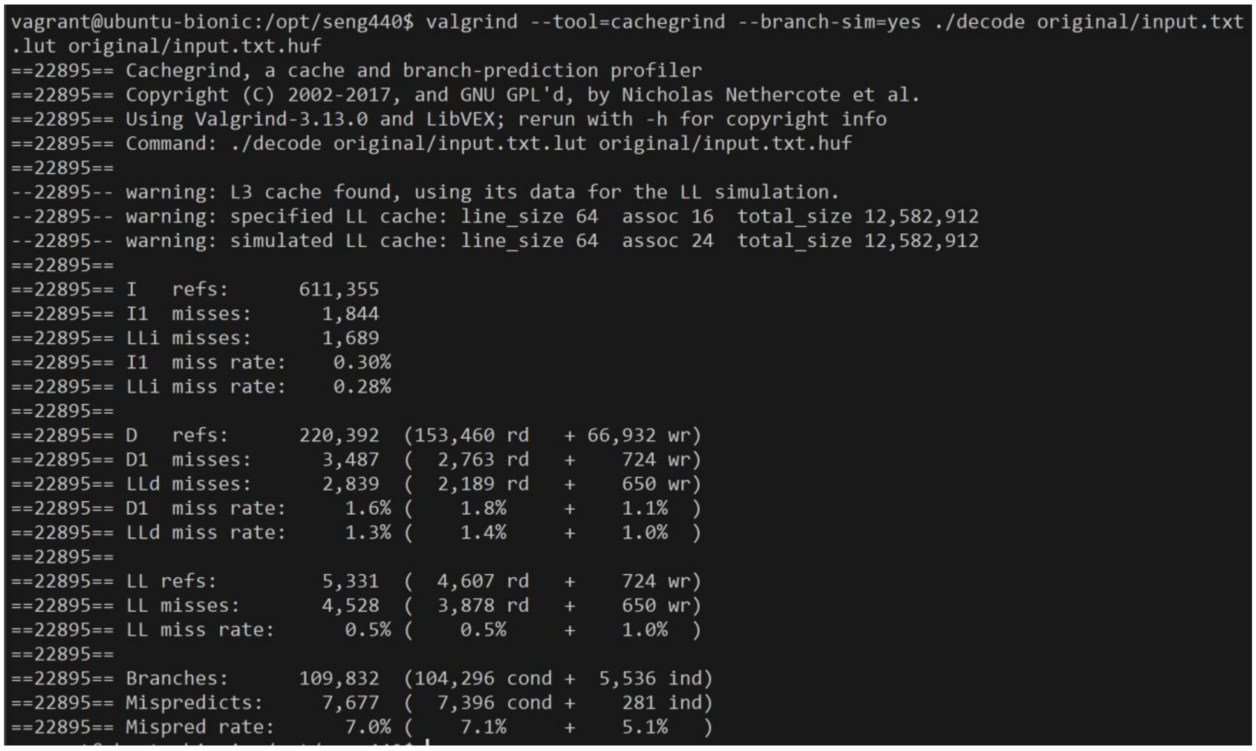
```
2,8
128,10000001
130,0111111
136,011110
137,01110
138,0110
139,010
142,00
143,11
148,101
153,1001
156,10001
160,100001
162,1000001
168,0111110
169,10000000
```

```
1000000110000000101111110111111011111101
1110011110011110011110011110011100111100
1110011100111001110011100111001111001100
1100110011001100110011001100110011001100110
0110011001100110011001100100100100010010
0100100100100100100100100100100100010010
0100100100100100100100100100100100010010
0100100000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000111111111111111111111
1111111111111111111111111111111111111111
1111111111111111111111111111111111111111
1111111111111111111111111111111111111111
1111111111111111111111111111111111111111
1111111111111111111111111111111111111111
1111111111111111111111111111111111111111
111011011011011011011011011011011011011
011011011011011011011011011011011011011
011011011011011011011011011011011011011
011011011011011011011011011011011011011
011011011011011011011011011011011011011
001100110011001100110011001100110011001
100110011001100110011001100110011001100
110011001100110011001100110011001100110
011001100110001100011000110001100011000
110001100011000110001100011000110001100
01100001100001100001100001100001100001100
001100000110000011000001100000110000011000
0011000000101111100111111010000000
```

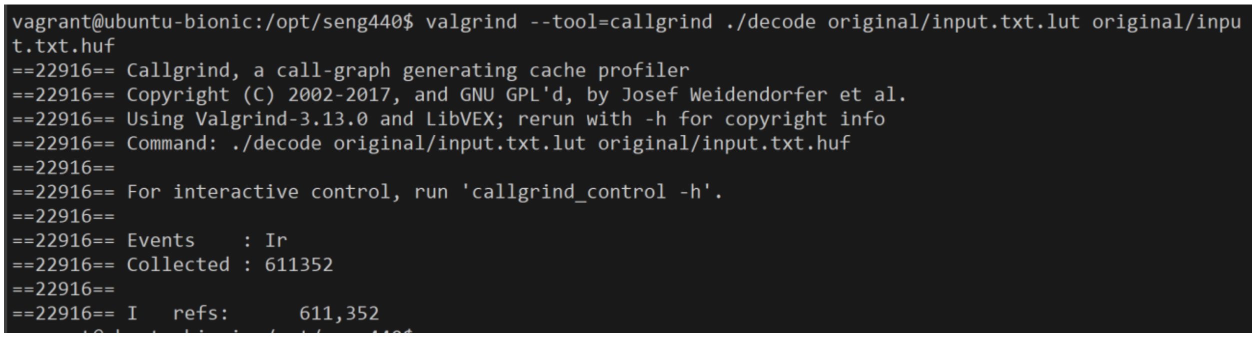
Figure B-1. Sample LUT (left) and HUF (encoded text file, right) files that were generated from the original French-character input file (top) as a project demonstration. The characters in the input file followed a normal distribution. Both LUT and HUF files are passed as input to the Huffman decoder.

8.3 – Appendix C – Comparison of Valgrind Cache and Instruction Count Results

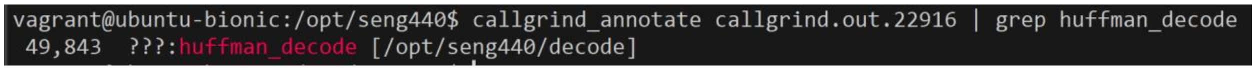
Original Program:



```
vagrant@ubuntu-bionic:/opt/seng440$ valgrind --tool=cachegrind --branch-sim=yes ./decode original/input.txt
.lut original/input.txt.huf
==22895== Cachegrind, a cache and branch-prediction profiler
==22895== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==22895== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==22895== Command: ./decode original/input.txt.lut original/input.txt.huf
==22895==
--22895-- warning: L3 cache found, using its data for the LL simulation.
--22895-- warning: specified LL cache: line_size 64  assoc 16  total_size 12,582,912
--22895-- warning: simulated LL cache: line_size 64  assoc 24  total_size 12,582,912
==22895==
==22895== I   refs:      611,355
==22895== I1  misses:      1,844
==22895== LLi misses:      1,689
==22895== I1  miss rate:    0.30%
==22895== LLi miss rate:    0.28%
==22895==
==22895== D   refs:      220,392  (153,460 rd   + 66,932 wr)
==22895== D1  misses:      3,487  (  2,763 rd   +    724 wr)
==22895== LLd misses:      2,839  (  2,189 rd   +    650 wr)
==22895== D1  miss rate:     1.6% (    1.8%     +    1.1%  )
==22895== LLd miss rate:     1.3% (    1.4%     +    1.0%  )
==22895==
==22895== LL refs:         5,331  (  4,607 rd   +    724 wr)
==22895== LL misses:       4,528  (  3,878 rd   +    650 wr)
==22895== LL miss rate:      0.5% (    0.5%     +    1.0%  )
==22895==
==22895== Branches:      109,832  (104,296 cond +  5,536 ind)
==22895== Mispredicts:     7,677  (  7,396 cond +    281 ind)
==22895== Mispred rate:      7.0% (    7.1%     +    5.1%   )
```



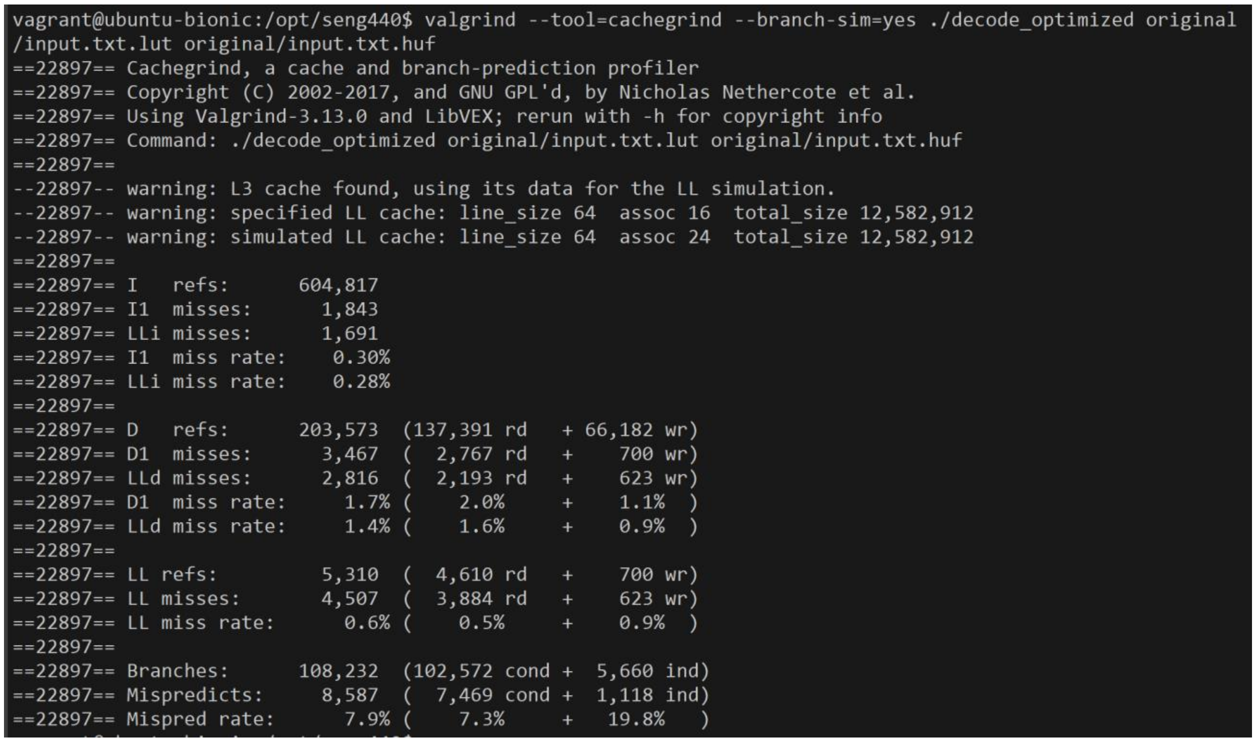
```
vagrant@ubuntu-bionic:/opt/seng440$ valgrind --tool=callgrind ./decode original/input.txt.lut original/inpu
t.txt.huf
==22916== Callgrind, a call-graph generating cache profiler
==22916== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==22916== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==22916== Command: ./decode original/input.txt.lut original/input.txt.huf
==22916==
==22916== For interactive control, run 'callgrind_control -h'.
==22916==
==22916== Events    : Ir
==22916== Collected : 611352
==22916==
==22916== I   refs:      611,352
```



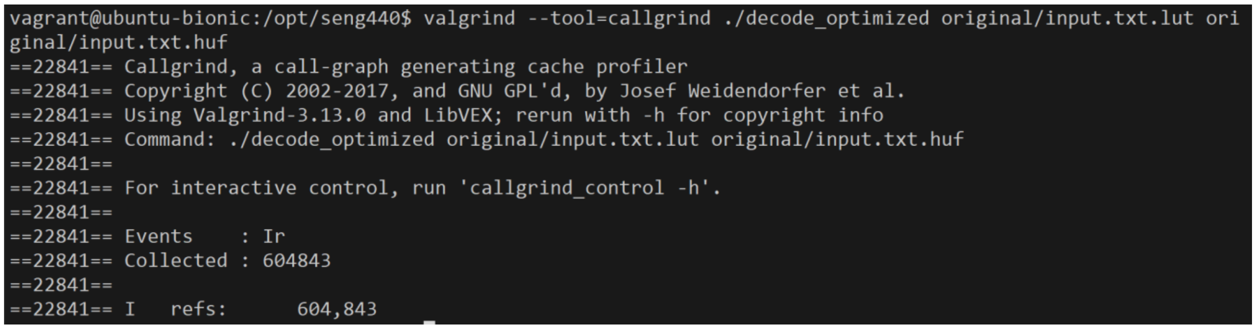
```
vagrant@ubuntu-bionic:/opt/seng440$ callgrind_annotate callgrind.out.22916 | grep huffman_decode
 49,843  ???:huffman_decode [/opt/seng440/decode]
```

Figure C-1. Valgrind results for the unoptimized decoding code, tested with the project test scenario. Results were obtained in a Linux environment on an Intel x86 machine with 32KB L1 cache, 256KB L2 cache, 12288KB L3 cache.

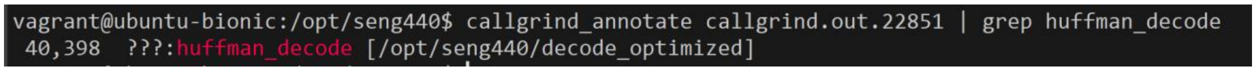
Optimized Program:



```
vagrant@ubuntu-bionic:/opt/seng440$ valgrind --tool=cachegrind --branch-sim=yes ./decode_optimized original
/input.txt.lut original/input.txt.huf
==22897== Cachegrind, a cache and branch-prediction profiler
==22897== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==22897== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==22897== Command: ./decode_optimized original/input.txt.lut original/input.txt.huf
==22897==
--22897-- warning: L3 cache found, using its data for the LL simulation.
--22897-- warning: specified LL cache: line_size 64  assoc 16  total_size 12,582,912
--22897-- warning: simulated LL cache: line_size 64  assoc 24  total_size 12,582,912
==22897==
==22897== I   refs:      604,817
==22897== I1  misses:      1,843
==22897== LLi misses:      1,691
==22897== I1  miss rate:    0.30%
==22897== LLi miss rate:    0.28%
==22897==
==22897== D   refs:      203,573  (137,391 rd   + 66,182 wr)
==22897== D1  misses:      3,467  (  2,767 rd   +    700 wr)
==22897== LLd misses:      2,816  (  2,193 rd   +    623 wr)
==22897== D1  miss rate:     1.7% (    2.0%     +    1.1%  )
==22897== LLd miss rate:     1.4% (    1.6%     +    0.9%  )
==22897==
==22897== LL refs:         5,310  (  4,610 rd   +    700 wr)
==22897== LL misses:       4,507  (  3,884 rd   +    623 wr)
==22897== LL miss rate:      0.6% (    0.5%     +    0.9%  )
==22897==
==22897== Branches:      108,232  (102,572 cond +  5,660 ind)
==22897== Mispredicts:     8,587  (  7,469 cond +  1,118 ind)
==22897== Mispred rate:      7.9% (    7.3%     +   19.8%   )
```



```
vagrant@ubuntu-bionic:/opt/seng440$ valgrind --tool=callgrind ./decode_optimized original/input.txt.lut ori
ginal/input.txt.huf
==22841== Callgrind, a call-graph generating cache profiler
==22841== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==22841== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==22841== Command: ./decode_optimized original/input.txt.lut original/input.txt.huf
==22841==
==22841== For interactive control, run 'callgrind_control -h'.
==22841==
==22841== Events    : Ir
==22841== Collected : 604843
==22841==
==22841== I   refs:      604,843
```



```
vagrant@ubuntu-bionic:/opt/seng440$ callgrind_annotate callgrind.out.22851 | grep huffman_decode
 40,398  ???:huffman_decode [/opt/seng440/decode_optimized]
```

Figure C-2. Valgrind results for the ARM-optimized decoding code, tested with the project test scenario. Results were obtained in a Linux environment on an Intel x86 machine with 32KB L1 cache, 256KB L2 cache, 12288KB L3 cache.

8.4 – Appendix D – Block Diagram of the ARM920T Microprocessor [2]

1.2 Processor functional block diagram

Figure 1-1 shows the functional block diagram of the ARM920T processor.

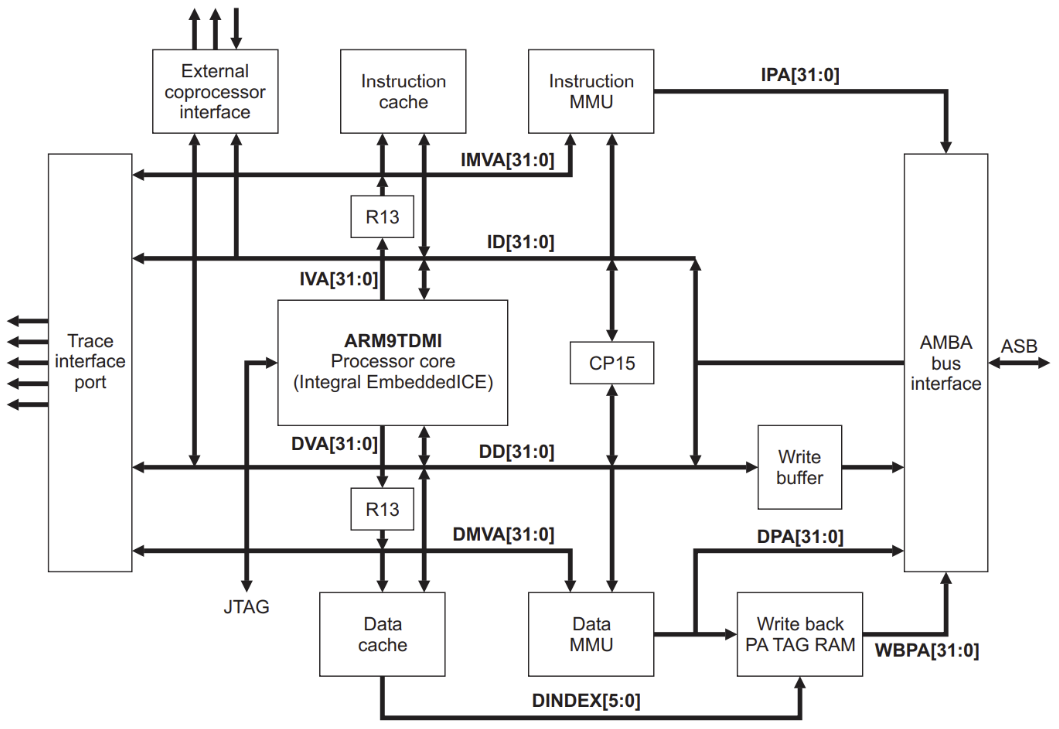


Figure D-1. Functional Block Diagram of the ARM920T Microprocessor [2].