SENG 440 Embedded Systems
Term Project:

Huffman Coding

Michael Kitzan (Dept. of CS, V00869307, mkitzan@uvic.ca)
Patrick Holland (Dept. of CS, V00878409, patrickeholland@uvic.ca)

Submission Date/Time: ____________________________

Introduction

Minimizing the data footprint of files and programs is desirable in any context. In the severely memory constrained paradigm of embedded systems, minimizing the data footprint of entities is necessary. For example, reducing the code and data size is a key area of optimization for microprocessors with small flash memories [1]. Minimizing the instruction count or opting for a low space complexity algorithm are options for accomplishing this. However, these methods often come at the cost of overall system performance by impinging on algorithm and data structure optimizations. Instead, compressing the entire executable code and data of the system is a beneficial alternative [1]. There are a number of forms of data compression: lossless, lossy, run-length based, and frequency based. Frequency based compression schemes are particularly effective on text files (like code), and Huffman coding specifically has been shown to have higher compression ratios when compared against other schemes [2].

Implementations of compression algorithms need to be fast. People want the benefits of a minimized data footprint but not at the cost of bottleneck compression times. This project displays an optimized implementation of the Huffman coding compression scheme for embedded systems, particularly the ARM920T microprocessor. The project is organized into four implementations of the compression scheme with performance improvements between each subsequent implementation. The project and deliverables were completed as specified in Table 1.

Table 1: Enumeration of contributions

Task	Contributor	Task	Contributor
Software (C and ASM)	Michael Kitzan	Functional Unit	Patrick Holland
Optimization (C and ASM)	Michael Kitzan	Report (4/8 Sections)	Patrick Holland
Report (4/8 Sections)	Michael Kitzan	Optimization (Lookup Table)	Patrick Holland
Performance Analysis	Michael Kitzan	Performance Evaluation	Patrick Holland
UML Charts	Michael Kitzan		

Theory

Huffman coding is a lossless frequency based compression algorithm commonly used for text files. The intuition behind Huffman coding is that characters appearing more frequently should require less bits of information to represent. The length of a character’s code, the bit sequence representing it, should be directly dependent on its frequency in the text file. From David A. Huffman’s original paper, “A minimum-redundancy code is one constructed in such a way that the average number of coding digits per message is minimized.” [3]

One of the biggest hurdles with this type of scheme is ambiguity. If the characters are represented by variable length codes, how will an algorithm determine when it has reached the end of a code? The Huffman coding scheme guarantees that every character code has a unique prefix. For example, if the character ‘s’ has the code ‘001’, it is guaranteed (and necessary) that no other code in the alphabet will begin with ‘001’. This also guarantees that no other character is represented by ‘0’ or ‘00’. The entirety of an alphabet is represented by a binary tree where left branches represent ‘0’ value bit and right branches represent ‘1’ bit. So for example, character ‘s’ with code ‘001’ is the leaf node found by traversing from the root node taking the path LEFT->LEFT->RIGHT.

The encoding process consists of traversing the Huffman tree to create a mapping from each character to its respective Huffman code. Then the algorithm iterates through the characters in the un-encoded text while printing the code for each character at each iteration. The decoding process can be performed in $O(n)$ time, where n is the number of characters in the text file being encoded.

The naive decoding process involves taking 1 input bit (from the encoded file) at a time and using it to walk the tree until you reach a leaf node. Starting at the tree’s root and whenever a ‘0’ bit is read the algorithm traverses to the current node’s left child. Whenever a ‘1’ bit is read the algorithm traverses to the current node’s right child. Upon reaching a leaf node, print the character contained within that node then return to the Huffman tree’s root and continue. Figure 1 shows the state diagram for tree based Huffman decoding.

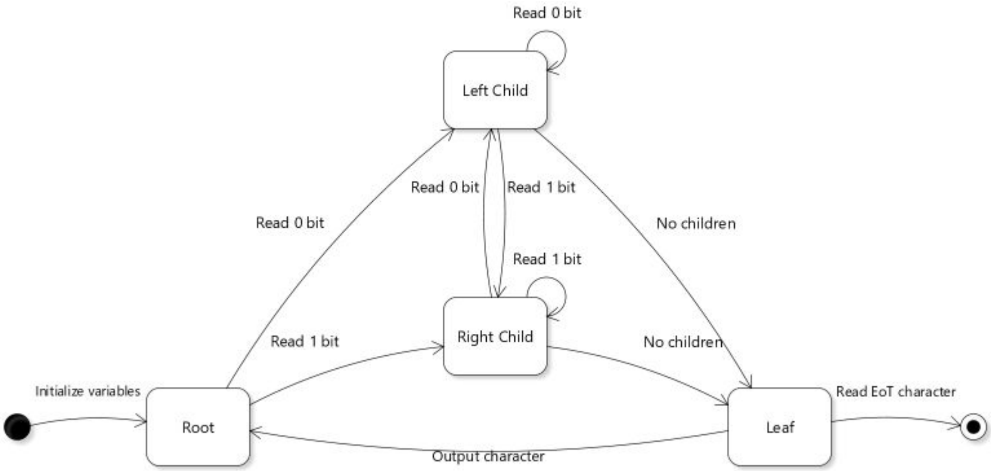


Figure 1: Tree based decoding state diagram

Decoding is an inherently sequential algorithm. Due to the nature of variable length codes, the decoding process cannot benefit from parallelization (multi-threading/multi-core processing). The decoding work can not be divided because it is unclear where a code ends and another begins. Encoding, however, can be parallelized since the input text may be divided into chunks and then the bit sequences combined at the end.

Generally for naive implementations, the tree nodes are represented using binary tree nodes. They have a field for frequency, pointers to left and right child nodes, and a field for leaf nodes to store a character. The codes used for encoding can be represented as a map or dictionary in a language such as Python, but in C they are generally represented using an array where the character's ASCII value is used for indexing.

Optimizations for the algorithm depend on what assumptions can be made. If the alphabet and frequencies are known ahead of time, the Huffman tree and codes can be hard coded in the program's data section saving processing time to build the tree every time the program runs. If no assumptions can be made about the alphabet or its frequencies then before encoding, the input file must be analyzed to create an alphabet with frequencies so the corresponding tree can be built. This tree would then need to be stored within the encoded output file so that the decoder can reconstruct it.

If the encoder and decoder agree on an alphabet and frequencies ahead of processing, time and memory can be saved, but it also allows other optimization techniques such as using a lookup table instead of a tree. A lookup table can allow the decoder to consume multiple bits at a time instead of just one. For example, if the minimum leaf node height is known to be three, an array of length eight ($2\^3$) can be constructed to hold the eight subtrees which begin at level three of the tree. This allows the decoder to start each iteration by using the first three bits of the encoded input file to index into the subtree array. This example effectively replaces three loop iterations with one and is the basis for the core optimizations in this project's design process.

Alphabet and Tree Building

Building the Huffman Tree

Basic Huffman coding is implemented using a tree called a Huffman tree. This tree is built with a simple algorithm that is easiest to understand with a visual example.

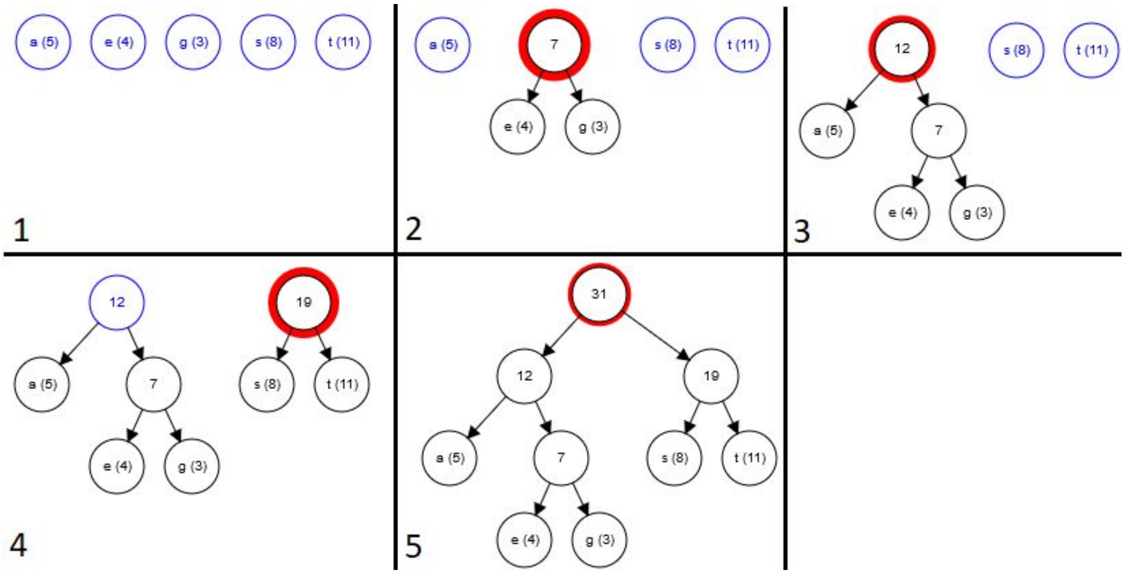


Figure 2: Example of going from an alphabet with frequencies to a Huffman tree. Screenshots taken from https://people.ok.ubc.ca/ylucet/DS/Huffman.html [4].

The algorithm starts with n single node trees where n is the number of characters in the alphabet. Each of these nodes contains the character and its frequency. At each step, the two root nodes with the lowest frequency values become the children of a new node. The new node is given a frequency equal to the sum of its two children. The algorithm continues until there is only 1 tree (the Huffman tree root).

In a Huffman tree, every leaf node contains a character and every internal node does not. To retrieve the Huffman code of a particular character, you walk from the root of the tree to the node containing that character while writing a ‘0’ when the algorithm traverses to the left child and a ‘1’ when the algorithm traverses to the right child. In the Figure 2, to traverse from the root to the node with ‘g’, the algorithm takes the following path LEFT->RIGHT->RIGHT. This translates to ‘011’.

Notice how characters with lower frequencies are located deeper in the tree compared to higher frequency characters. This property guarantees that if *freq(‘a’) => freq(‘b’)*, *length(code(‘a’)) <= length(code(‘b’))*. Also, notice how the tree structure guarantees the unique prefix characteristic. If ‘001’ is a valid code, then travelling LEFT->LEFT->RIGHT from the root must traverse to a leaf node. So ‘001X’ will never exist because that would require the leaf node to have a child.

Alphabet Selection

As mentioned in the Theory section, establishing a shared alphabet and frequency set between the encoder and decoder is an important requirement for optimizing Huffman coding. For this project’s Huffman coding implementations the first 128 ASCII characters were decided as the alphabet and the frequencies were based on 13 public domain English eBooks offered by Project Gutenberg [5].

After building the Huffman tree using the selected alphabet and frequencies, we noticed the tree’s height was larger than expected, so 30 unprintable ASCII character were removed from the alphabet. This left the following ASCII characters: 0x03 (used for end of text symbol), 0x0A (new line character), 0x20-0x7F (alphanumeric characters and punctuation).

This new alphabet resulted in a Huffman tree with a minimum height of three and a maximum height of ten. Meaning that the Huffman codes would have lengths between three bits and ten bits with more common characters being closer to three. Traditional text is stored using eight bits for each character so ignoring worst-case inputs, our tree is capable of providing over 50% compression ratios for files.

Design Process

Four implementations of the Huffman coding compression scheme were created for this project: a heap based Huffman tree, an array based tree, a lookup table version, and functional unit version. The implementation iterations display the project’s design process feedback loop.

Node Based

The Huffman coding functions were initially implemented the traditional naive way. The alphabet and frequencies were hard coded into arrays and the program would start by using those arrays to build the Huffman tree. The tree was built using the process described in the "Tree Building" section. Figure 1 shows the node based decode process and Figure 3 shows the encoding process. This implementation was a good starting place because it is fairly easy to understand and represents the traditional process for Huffman coding. As a result, it served as a baseline when comparing the different optimized implementations.

Building the tree requires malloc calls which are expensive from a clock cycle point of view. Because the tree nodes were allocated using malloc, the decoding process (which involves traversing the tree) generates many page faults due to the disparate locality of nodes on the heap. Page faults and the tree building process were the main concerns addressed when brainstorming potential optimization techniques.

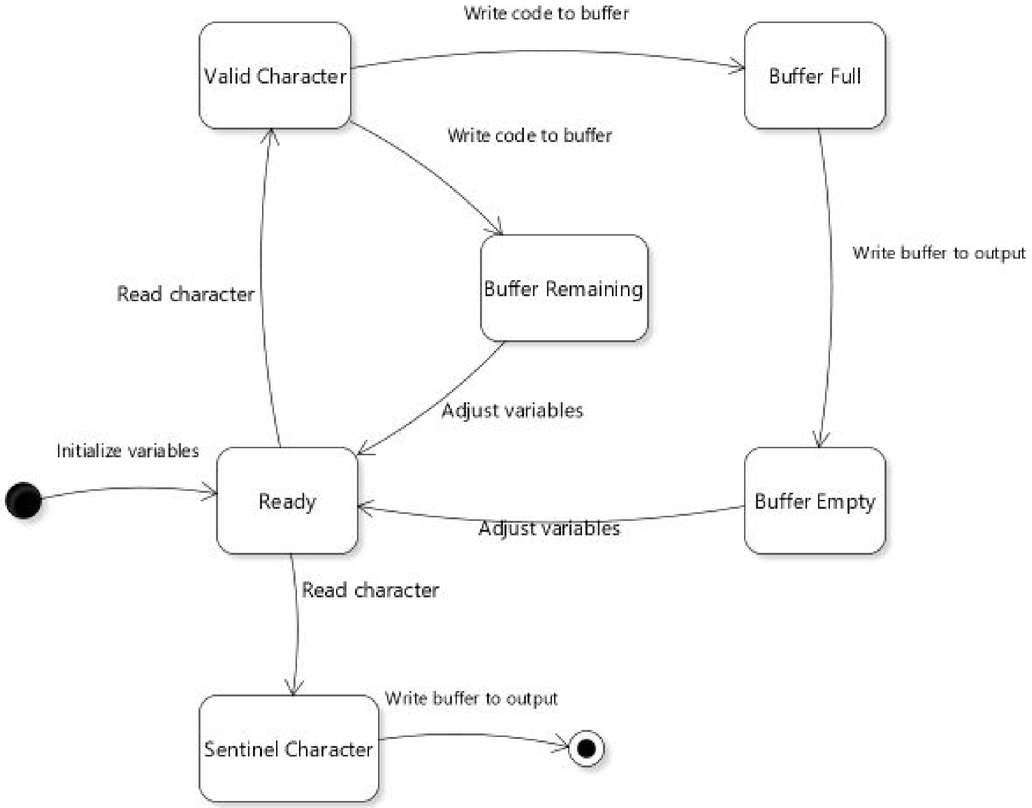


Figure 3: State chart for encoding process used in every implementation.

Array Based

Because the project's alphabet and frequencies are fixed, the array based implementation capitalized on this by generating and hardcoding the Huffman tree and Huffman codes within the files "src/huffman_data.h" and "src/huffman_utils.c". These files can be found in all the versions besides the Node based version.

To link a character with a Huffman code, three pieces of information are required: the character, the code, and the bit length of the code. The bit length of the code is required, because despite Huffman codes being variable length they must be stored in fixed length data types. The bit length specifies how many bits in a fixed length data type contribute to the code it stores. For example, if the letter ‘a’ had the code “0111”, its numeric representation would be 0x07. However, 0x07 in binary is “111” so the length of the code must be known to pick up the leading 0. By storing the codes within an array where the index of the array corresponds to that character’s ASCII code, the character does not need to be a data member in the data structure.

To represent a node in a Huffman tree, you must have pointers to the left and right child, and the character that the node represents. As mentioned earlier, nodes which represent a valid character will always be leaf nodes and nodes with invalid characters will always be internal nodes.

Table 2: Huffman codes and tree nodes being initialized in program memory.

Construct	**Huffman Code**	**Huffman Node**

Struct	`struct hcode {`
	`    uint16_t code;`
	`    uint8_t len;`
	`};`
		`struct hnode {`
		`    char letter;`
		`    hnode_t *left, *right;`
		`};`

Macro	`#define C102 { .code=5, .len=6 }`	`#define N100 { .letter=0x80, .left=(hnode_t*)(102*sizeof(hnode_t)), .right=(hnode_t*)(103*sizeof(hnode_t)) }`
Array	`hcode_t ALPHABET[128] = { C0, C1, …, C127 };`	`hnode_t TREE[188] = { N0, N1, …, N187 };`

Table 2 shows the constructs used to serialize the Huffman codes and tree nodes within “src/huffman_data.h” and “src/huffman_utils.c”. The serialized nodes are placed in an array and the base of the array is added the left and right pointers (which are serialized as offsets from the array base). This means the program no longer constructs the Huffman tree on each execution of the program, so no malloc calls are required. An important side effect was that the programmers could strategically place related nodes closely together in the array to promote space locality. This reduces the likelihood of a cache miss since it increases the likelihood that a node’s children will be in the same cache line as itself.

This design provided significant performance improvements when compared with the first node based implementation. The improvements were mainly seen in the decoding process and are discussed in the performance section of the report. These performance gains were promising, but more importantly, this design opened the door to the next implementation.

Lookup Table Based Implementation

The array based implementation reduced the cache misses and also allowed the program to avoid rebuilding the tree on each execution. However it still decoded by reading a single bit at a time. Development focus shifted to converting the Huffman tree into a lookup table so the program could utilize multiple bits per iteration. A strategy for decoding using only lookup tables, was explored in this third iteration as suggested by Dr. Sima's Huffman coding slide deck [6].

The tree's minimum height is three which means that it could be split into eight the subtrees rooted at depth three. Every code is at least three bits long so instead of looking at those bits one at a time and arriving at depth three after three iterations, the decoder starts by indexing the array of subtrees with the first three bits of the input. For example, if the input starts with '011', the unique prefix property guarantees that the code's character will be found in subtree three.

This allows the decoder to skip 2 iterations per decoded character, but this same logic could be applied to the rest of the tree: indexing a table and jumping directly to the correct leaf node. The problem at the time was that unprintable characters were still a part of the alphabet so the level three subtrees were of heights 0, 0, 1, 1, 3, 3, 12, and 27. This meant that to create a second layer of arrays to act as lookup tables for the subtrees, one of them would have to be of length 2^27. This would require an array with 134,217,728 elements or special case decode logic to handle extra lookup tables both of which are inefficient.

The next issue to arise was ambiguity. For example, in Figure 4 Nodes A and B are both leaf nodes representing characters in the alphabet: A has the code '100', and B has the code '1'. The lookup table array would be indexed with the first three bits from the input. Were the three bits '100', the node containing A would be found without trouble. However, any bit sequence whose least significant bit is '1' should index to the node containing B. B's node wouldn't be represented exclusively by code '001', but instead by codes '001', '101', '011', '111'. The node itself needs to contain a field telling the algorithm how many bits from the indexing bit string actually contribute to the character's code. So if the decoder saw '101', it would know to only consume the first '1' from the input stream and to save the remaining '10' to be used in the next code.

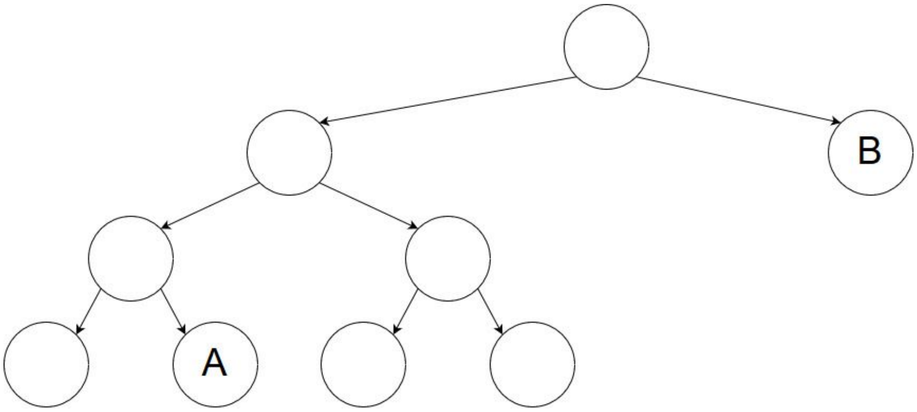


Figure 4: Example showcasing the potential ambiguity when designing a lookup table.

At this point, there was an array of eight subtrees which would be indexed by the first 3 bits from the input. The array stores pointers to the lookup tables representing its subtree and an integer field representing how many bits were required to index that lookup table. For example, if the first three bits were ‘110’, LOOKUP[6].tree would contain a pointer to the lookup table for subtree six, and LOOKUP[6].draw would be a mask with its 12 least significant bits set to 1 (because subtree 6 has height 12). An AND operation between the mask and input buffer would produce an index value for LOOKUP[6].tree which in turn would produce the character and an integer representing how many of those 12 bits to remove from the buffer.

The biggest issue at this point was the subtree with height 27. It was at this point that all of the unprintable ASCII characters were removed from the alphabet. Around 30 characters were removed and the tree was rebuilt. As a result, the tree’s maximum height went from 30 to 10. This meant that the longest code in the alphabet would be 10 bits which made it reasonable to have every character accessible by lookup table.

To put the entire tree in one lookup table would require 2^10 (1024) array elements which isn’t terrible, but two layers of lookup tables could be used to minimize the amount of memory required. The two data fields of each array element can fit in eight bit data types so each array element is two bytes. Using two layers of lookup tables, there would be: the initial level three subtree array (eight elements that are five bytes each), one lookup table with one element, two lookup tables with two elements, one lookup table with eight elements, three lookup tables with 64 elements, and one lookup table with 128 elements. The total memory requirements for the two layers is then 690 bytes compared to the 2028 byte memory requirement for the single table option. If the memory requirement is not an issue then the algorithm can be modified slightly to work on a single lookup table. The functional unit based implementation that will be discussed next follows this design path. Figure 5 shows the decode state diagram for the two layer lookup table implementation.

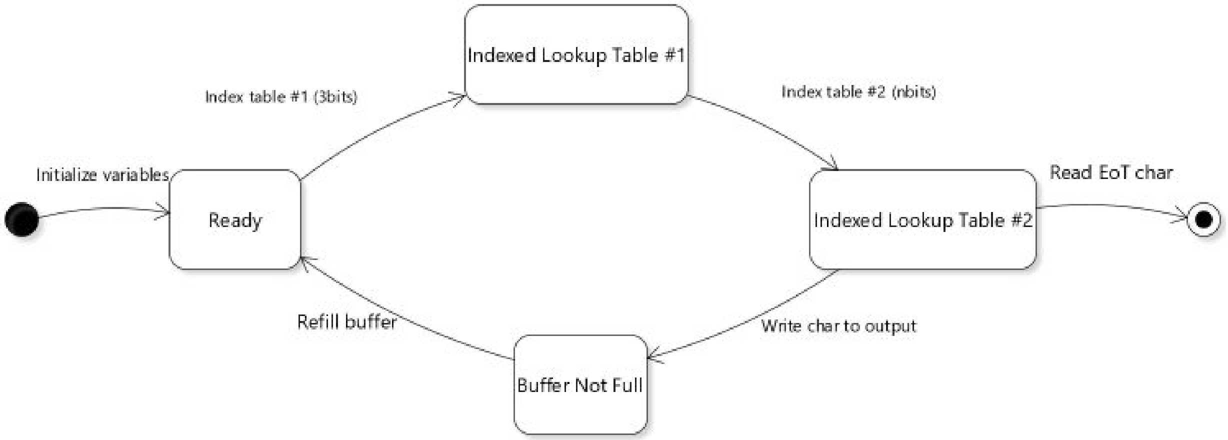


Figure 5: State chart for two lookup table based decoding.

Functional Unit Based Implementation

Performance analysis of the lookup based implementation showed that cache misses contributed significantly to the program’s runtime. This is what the functional unit aims to address. According to the user manual for the Samsung S3C2440A, the board has a cache with an 8-word (32 byte) line length [7]. The lookup tables are accessed in a random order so they cannot benefit from sequential accesses like the input array can.

A functional unit acting as a programmable cache could help eliminate a significant number of cache misses. With the maximum height of the tree being 10, that means that 2^10 (1024) array elements would be required. Each element is 2 bytes so the entire lookup table can fit in 2048 bytes of storage. This unit would require a 10 bit number as input and its output would be a 2 byte struct containing the character and the number of bits to be consumed. It would not require any state and interrupts would not need to be disabled while using it.

If the unit’s memory was volatile, the entire lookup table would need to be loaded in at the start of program execution. If the memory was persistent, the lookup table would only need to be loaded in one time since the alphabet and frequencies do not change. The primary focus of this unit is to provide fast reads which are as close in latency to L1 cache as possible. Figure 6 depicts the state diagram implemented by decoding with the hypothetical functional unit.

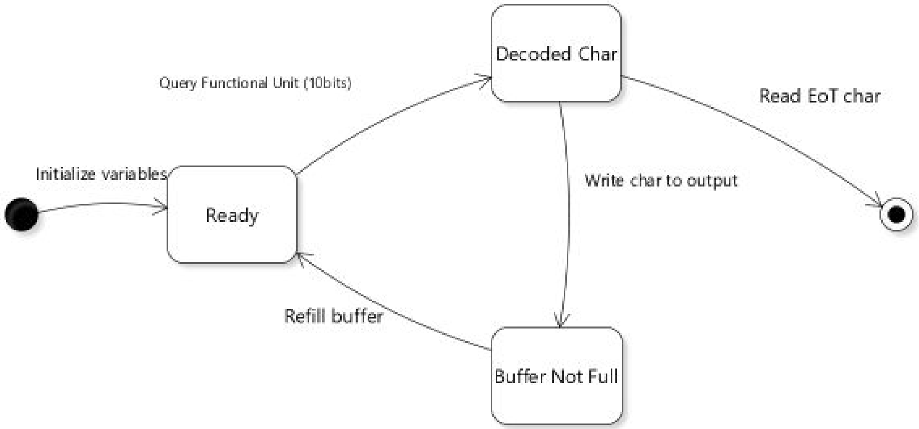


Figure 6: State chart for functional unit based decoding.

Optimizations

Between each implementation significant optimizations were made, and many of the optimizations carry over between versions. The most significant optimizations were algorithmic changes between each version, like the change from an array based binary tree to lookup tables. However, with each version many smaller individual C level optimizations were made, like reducing direct dependencies and using bit test predicates. Only with the completion of the lookup table version were assembly optimization made. Finally, after proposing a functional unit, the code was optimized to capitalize on the hypothetical hardware. The realized performance improvements from the optimizations will be covered in the performance section following.

C Level Optimizations

C level optimizations primarily focus on the decode function because it was identified as the primary bottleneck early in development. Only minor C level optimizations were made between versions of encode, in no small part due to the function not being affected by version changes. For both functions, loop unrolling was an unrealistic optimization technique, because the number of variables required would exceed the register file.

Encode Optimizations

Five main optimization methods were leveraged when optimizing the encode function, highlighted in Table 3. Because the encode function was not radically changed between versions, optimizations between each core implementation persisted in subsequent versions. Table 3 shows a before and after of optimizations, “Before” is pulled from the encode function for the node based version and “After” is from the encode function for the lookup based implementation. The table shows the culmination of optimizations between versions.

Table 3: C level encode optimizations

Method	Before Optimization	After Optimization

Restrict Pointers	`encode(const char *text, uint32_t *code)`	`encode(const char *restrict text, uint32_t *restrict code)`

Variable Data Types	`unsigned int loc = 0,`
	`  len = 0, i = 0;`
	`uint32_t buffer = 0;`
	`uint8_t key;`
		`register uint32_t i = 0,`
		`  loc = 0, len = 0,`
		`  buffer = 0, key;`

Bit Test Predicates	`// note: CODE = 32`
	`if(len >= CODE) {`
		`// note: CODE = 32`
		`if(len & CODE) {`

Direct Dependency	`len -= CODE;`
	`buffer = DICT[key].code >> (DICT[key].len - len);`
	`code[loc++] = buffer;`
		`len -= CODE;`
		`code[loc++] = buffer;`
		`buffer = DICT[key].code >> (DICT[key].len - len);`

Software Pipelining	`for(; text[i]; ++i) {`
	`  key = text[i];`
	`  buffer |= DICT[key].code << len;`
	`  ...`
	`}`
		`key = text[i];`
		`while(text[i++]) {`
		`  buffer |= DICT[key].code << len;`
		`  ...`
		`  key = text[i];`
		`}`

Decode Optimizations

Between the node-based and array-based implementation, a number of improvements were made beyond the overarching change in Huffman tree structure. When traversing the tree, the predicate checking whether the current node is a leaf or not was significantly improved. Originally it was a compound predicate checking if the left and right children are both null, but was changed to a bit test on the node's "letter" data member. During tree construction, the "letter" data member of internal nodes are set to 128, because it is the next power of two outside the alphabet range and enables bit testing for internal nodes.

The Huffman tree is traversed by checking the least significant bit of the code buffer, moving to left or right child, then shifting the buffer. If the buffer becomes empty during this process it must be refilled. The array-based implementation moved the buffer empty check and refill logic outside the tree traversal loop reducing branching in the main bottleneck loop of the tree based decoding algorithm.

When the implementation changed to decoding with lookup tables, the traversal loop was removed entirely. Lookup tables guarantee a constant number of pointer dereferences to decode a letter (rather than dereferences depending on the letter's code length). However, this came at the cost of increased bookkeeping operations to maintain the buffer. Introducing software pipelining and minimizing direct dependencies between instructions were the primary optimization techniques used to address the increased bookkeeping.

Assembly Level Optimizations

For reference, the assembly code used as the basepoint for optimization was generated using GCC with optimization flag -O3. Contrary to expectation the encode function required the most assembly optimization compared to decode.

Encode Optimizations

Despite what appears to be a simple function at the C level with only two apparent branch points, GCC's encode function features five different branch points. On average there are six instructions between each branch and no predicate guarded instructions are used. GCC's assembly encode function was removed entirely and rewritten by hand. The rewrite for the function focused on minimizing the branches to a single core loop, capitalizing on predicate guarded instructions, and increasing software pipelining. GCC used branching to navigate around the buffer refill section and to cover the edge case where the first character is null. In the rewriting the encode function, the buffer refill section was implemented as four predicate guarded instructions removing the pipeline bubble introduced by branching. Finally, software pipelining was increased from the C level function's existing minor pipelining. The assembly function not only loads the next character from the input sequence, but also transforms the character into an index value for the Huffman code array. The full assembly functions and their expected behavior in the microprocessor pipeline are present in Appendices A and B.

Decode Optimizations

Fortunately, GCC's decode assembly function was a close translation of the C level function. The assembly optimization stage for decode focused on three optimization methods: collapsing multiple instructions into single instructions, introducing assembly level software pipelining, and reducing direct dependencies. The core decode loop was minimized by eight instructions corresponding to a reduction in expected cycles per iteration from 49 to 39. Similar to the encode function, the decode function was able to capitalize on assembly level software pipelining by prefetching the level three lookup table before each iteration. Only one of the existing three direct dependencies in GCC's decode was removed by interrupting the dependency with an unrelated operation. Table 4 shows the removal of this direct dependency. The full assembly functions and their expected behavior in the microprocessor pipeline are present in Appendices C and D.

Table 4: Eliminated direct dependency between usage of r9 in decode function

GCC Decode Assembly	Optimized Decode Assembly

`lsr r9, r3, #3`
`and r4, r4, r9`
	`lsr r9, r9, #3`
	`ldrb r4, [r7, #4]`
	`ldr lr, [r7]`

	and r4, r4, r9

Functional Unit Optimization

Utilizing a lookup table cache functional unit enabled two optimizations for the decode function: minimizing bookkeeping operations to preserve state between iterations, and eliminating four memory access operations. Table 5 contrasts the decode logic in the lookup table and the functional unit versions. The instructions in Table 5 have been reordered for readability, so there are direct dependencies shown which do not exist in the executable versions. In the lookup table version, the first two blocks index the lookup tables to decode the present character. However, four of the seven instructions are concerned with manipulating the buffer and setting up the next lookup operation. With the functional unit these code blocks are collapsed into a single call to the functional unit and one shift operation after the call to the functional unit. Also, though not explicitly shown in Table 5, all direct dependencies between operations are eliminated in the functional unit implementation (place “temp = code[pos] >> seen” after the functional unit call).

Table 5: Core decode logic without and with a lookup table functional unit

<table>
<tr><th>Lookup Table Decode</th><th>Functional Unit Decode</th></tr>
<tr><td>// index level 3 table
temp = buffer & 7;
table = LOOKUP[temp].table;
temp= LOOKUP[temp].draw;
buffer >>= 3;

// index subtree table
temp &= buffer;
bits = table[temp].contrib + 3;
buffer >>= table[temp].contrib;

// adjust control variables
text[loc++] = table[temp].letter;
temp = code[pos] >> seen;
seen += bits;
buffer |= temp << (CODE - bits);</td><td>// utilize functional unit
entry = lookup_unit(buffer);
buffer >>= entry.contib;

// adjust control variables
text[loc++] = entry.letter;
temp = code[pos] >> seen;
seen += entry.contib;
buffer |= temp <<
(CODE - entry.contib);</td></tr>
</table>

Performance

The three fully executable versions, node, array, and lookup table (before assembly optimization), underwent a testing regime evaluating the performance on three different test cases: worst, best, and average. Each test input was 100mb in size and were run four times each on the three versions. The average case input was constructed by concatenating together all the eBooks used to generate alphabet frequencies. The concatenated file was then concatenated to itself eight times resulting in the average case input. The worst and best case inputs were created by replacing all the characters in the average case file with the characters corresponding with the largest and smallest Huffman codes. Testing was performed on a ARM920T virtual machine and statistics were gathered using the “perf” Linux performance evaluation tool. Performance was evaluated across three categories corresponding to the function was running: encode, decode, and syscalls. The results of the four test runs were averaged and the aggregated statistics were used to generate the charts in following subsections. The aggregated data can be found in Appendix E, and the raw data in the “performance” folder of the project directory. Because of virtualization, the performance numbers are inflated compared to those on the actual hardware. Part of the decision to test on 100mb files was to amortize anomalies from virtualization.

Encode Performance

With the knowledge that between each version the encode was largely unaffected, the performance results, in Figure 7, are not unexpected. The most significant performance improvement is between the node and array version which is largely attributable to changing the variables data type optimization (where all function variables became word sized register data types).

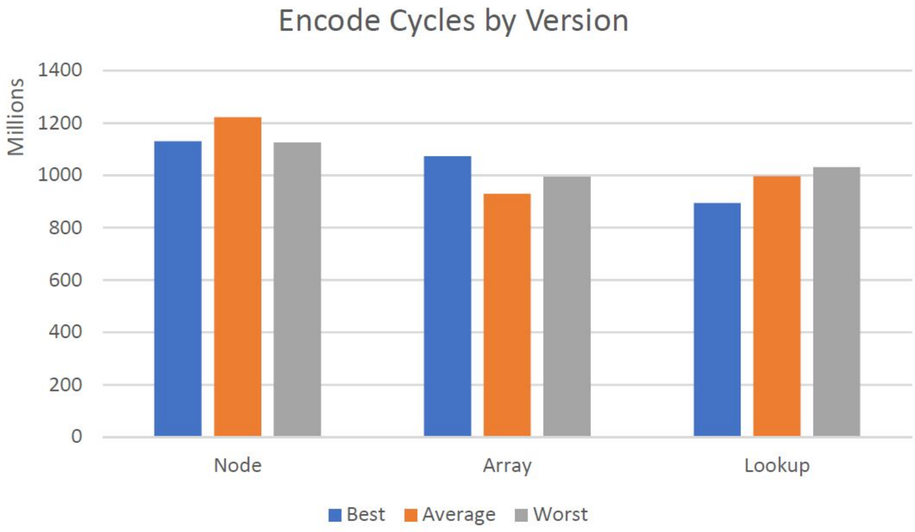


Figure 7: Encode performance across versions

Decode Performance

The benefits gained from the version changes are apparent in performance testing results of the decode function, shown in Figure 8. The switch from a heap based tree to a statically allocated array based tree generated a 50% performance increase. Then an additional 33% performance increase was gained from changing to lookup tables.

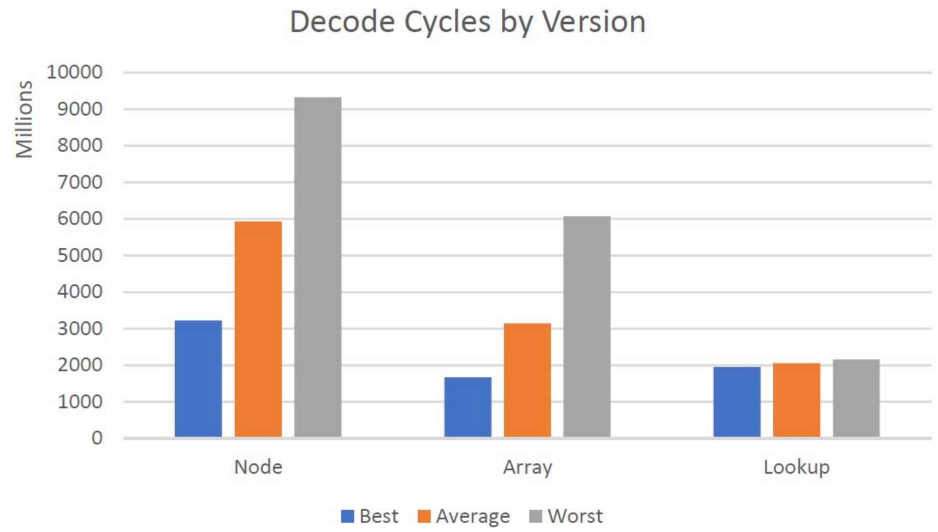


Figure 8: Decode performance across versions

Both tree based versions have widely varying performance between input file test cases. This makes sense considering each bit in a code is used to traverse a tree, so letters with shorter codes decode faster and decode slower for longer codes. The lookup table version was able to stabilize this behavior by having a constant number of pointer dereferences per decoded character. As a result, the best case input file took longer for the lookup table implementation because two pointer dereferences must occur, rather than the array based single dereference for the character with the shortest code (space). It is worth noting that even after significant optimization between versions the lookup table decode function is still the bottleneck function, running at 2000million clock cycles compared to encode's 1000million clock cycles. Making the need for a functional unit all the more apparent.

Syscalls Performance

Virtualization affects this category the most, because trapping into the virtualized operating system's kernel requires also trapping into the host operating system's kernel. Meaning these functions are run at a significantly handicapped speed. Because of this, it is difficult to assess the performance shown in Figure 9. The lack of overall change between versions is suspicious, but is likely due to the reliable number of page faults which will occur across all versions simply from loading the input data and writing the output data. Regardless of the implementation the entirety of the input must be read and the same output must also be written.

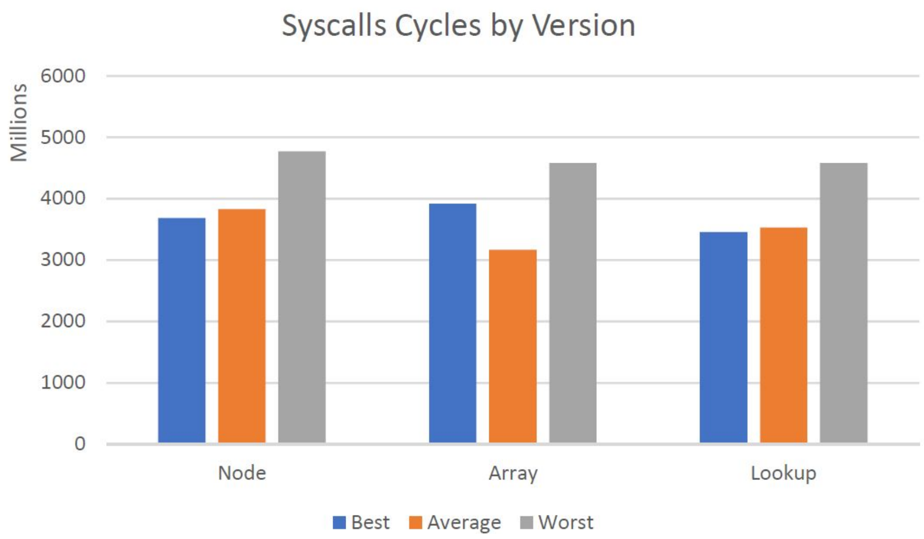


Figure 9: Syscall performance across versions

Assembly Performance

Assembly optimization gains were evaluated using expected run-time analysis, shown in Table 6. Both GCC's and the hand optimized functions were converted into polynomials representing their expected run times for n input elements. Because the functions iterate on characters encoded or decoded, some n quantities are multiplied or divided by five to represent the expected number of encoded characters which will fit in a uint32_t. Both assembly function optimizations show an expected 20% improvement over the GCC functions. The clock cycle estimates used to approximate these polynomials were constructed from the estimated behavior of the function's on an ARM920T microprocessor (see Appendices A, B, C, and D). Clock cycle statistics and pipeline behavior was gathered from the processor reference manual [8].

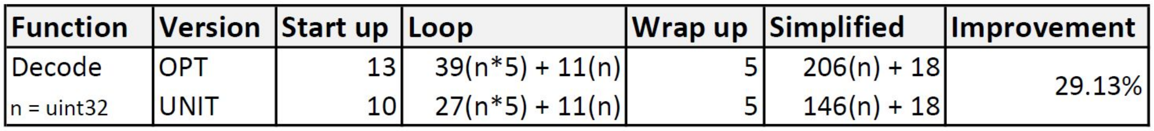
Table 6: Expected run-time analysis of assembly optimized function

Function	Version	Start up	Loop	Wrap up	Simplified	Improvement
Encode	GCC	22	32(n) + 12(n/5)	8	34(n) + 30	20.59%
n = char	OPT	17	27(n)	6	27(n) + 23	
Decode	GCC	11	49(n*5) + 12(n)	0	257(n) + 11	19.84%
n = uint32	OPT	13	39(n*5) + 11(n)	5	206(n) + 18	

Functional Unit Performance

The lookup table functional unit is a cache holding the static lookup tables which are allocated in the data section of the lookup table implementation. The benefit of a hardware unit to perform the lookup operation is that indexing the table costs no more processing time than a single memory access operation and it eliminates the intermediate bookkeeping operations between lookup table accesses. Because the functional unit acts as a supplementary cache, the call to the functional unit guarantees no page faults while eliminating two memory access operations. The benefits of the functional unit in comparison to the optimized assembly function are shown in Table 7. The assembly code GCC -O3 generated for the functional unit decode was hand optimized like the optimized lookup table decode assembly function, and the resulting function's expected behavior on an ARM920T microprocessor was evaluated (Appendix F contains the pipeline evaluation). The functional unit generates a 30% improvement over the optimized assembly function without the hardware unit and a 43% improvement over the unoptimized lookup table decode function. Using the tested performance of the unoptimized lookup table decode function, the functional unit decode would run in 1130million cycles: a nearly equivalent run time to the encode function.

Table 7: Expected run-time analysis of the functional unit decode.



Function	Version	Start up	Loop	Wrap up	Simplified	Improvement
Decode	OPT	13	39(n*5) + 11(n)	5	206(n) + 18	29.13%
n = uint32	UNIT	10	27(n*5) + 11(n)	5	146(n) + 18	

Considering the syscall performance evaluation, an additional area where functional units would be beneficial is when accessing the function input and output. Hypothetically, a functional unit which acted as a buffer to the input and output arrays could be used to absorb page fault latency so the core process could always have immediate access to the input and output. The input buffer would have a certain number of the input elements preloaded and would asynchronously load new elements. The output buffer would hold a queue of elements to write to output and would asynchronously write to the output array. One significant complication with these functional units compared with the lookup table cache is that in a practical setting the input and output locations will not be known at compile time (unlike the lookup table locations). There would have to be a software method to change their data sources at runtime.

Conclusion

This project was an exercise in optimization. Optimization was explored at four levels: algorithms and data structures, high level code, low level code, and hardware. Algorithm and data structure shifts between versions realized significant performance improvements. Simply switching the storage method of the Huffman tree resulted in doubling the performance of the decode function. From our testing, it is difficult to gauge the performance gains from high and low level code optimizations, because these optimizations were tested in tandem with the version changes. However, the lower level the software optimizations went the smaller the performance return. Consider the expected performance gain from the assembly optimizations, 20% improvement from the unoptimized lookup table run time. The unoptimized lookup table run time in turn gave a 33% improvement from the version before it, which itself was a 50% improvement from its predecessor. Optimizing exclusively at the software level can only go so far. This motivates exploring hardware assisted optimizations. Functional units and hardware level optimization has the unique ability to thwart inefficiencies caused by hardware level resources. Functional units proposed for optimizing Huffman coding exclusively addressed memory usage and eliminating page fault latency. Software level optimizations can only go so far in addressing page faults. For example page faults incurred by input and output access are unavoidable at the software level, but can be obfuscated by additional hardware. Embedded system development exposes the programmer directly to hardware level concerns enabling optimization outside the reach of pure software.

Bibliography

[1] M. Benes, S. Nowick and A. Wolfe, "A Fast Asynchronous Huffman Decoder for Compressed-Code Embedded Processors", *1998 IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1998. [Accessed 29 July 2019].

[2] K. Sailunaz, M. Kotwal and M. Huda, "International Journal of Computer Applications", 2014. [Accessed 29 July 2019].

[3] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098-1101, 1952. Available: 10.1109/jrproc.1952.273898.

[4] D. Galles and Y. Lucet, *Data Structure Visualization*. UBC Okanagan, 2011.

[5] *Project Gutenberg*, 2019. [Online]. Available: https://www.gutenberg.org/. [Accessed: 29-Jul- 2019].

[6] M. Sima, "SENG 440 Embedded Systems: Huffman Coding", 2019.

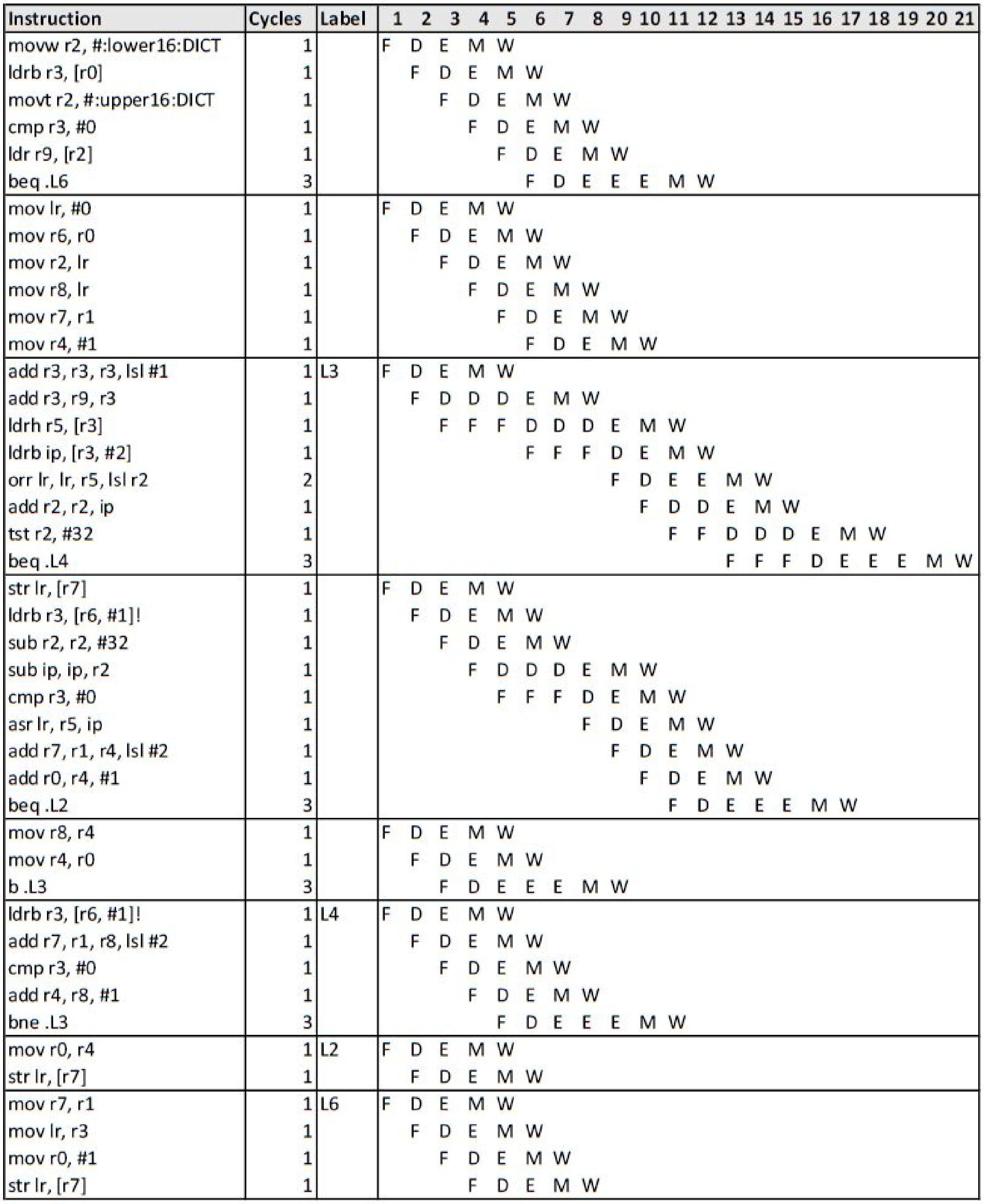
[7] *S3C2440A (Rev 0.14) User's Manua*l, 2004, pp. 1-1

[8] *ARM920T (Rev 1) Technical Reference Manual*, 1st ed. 2019, pp. 249-258.

Appendices

Appendix A

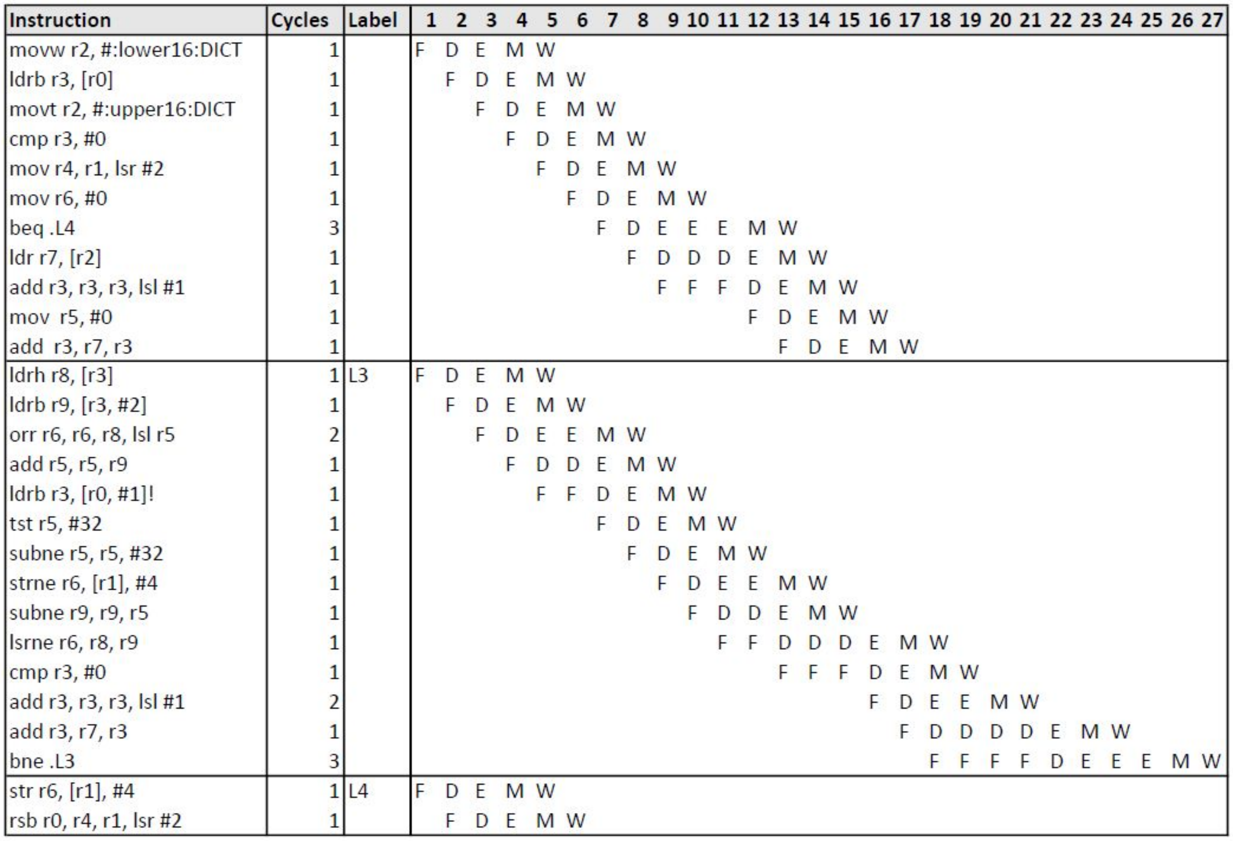
Simulated Processor Pipeline of GCC Encode Function



Instruction	Cycles	Label	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
movw r2, #:lower16:DICT	1		F	D	E	M	W																
ldrb r3, [r0]	1			F	D	E	M	W															
movt r2, #:upper16:DICT	1				F	D	E	M	W														
cmp r3, #0	1					F	D	E	M	W													
ldr r9, [r2]	1						F	D	E	M	W												
beq .L6	3							F	D	E	E	E	M	W									
mov lr, #0	1		F	D	E	M	W																
mov r6, r0	1			F	D	E	M	W															
mov r2, lr	1				F	D	E	M	W														
mov r8, lr	1					F	D	E	M	W													
mov r7, r1	1						F	D	E	M	W												
mov r4, #1	1							F	D	E	M	W											
add r3, r3, r3, lsl #1	1	L3	F	D	E	M	W																
add r3, r9, r3	1			F	D	D	D	E	M	W													
ldrh r5, [r3]	1				F	F	F	D	D	D	E	M	W										
ldrb ip, [r3, #2]	1							F	F	F	D	E	M	W									
orr lr, lr, r5, lsl r2	2										F	D	E	E	M	W							
add r2, r2, ip	1											F	D	D	E	M	W						
tst r2, #32	1												F	F	D	D	D	E	M	W			
beq .L4	3														F	F	F	D	E	E	E	M	W
str lr, [r7]	1		F	D	E	M	W																
ldrb r3, [r6, #1]!	1			F	D	E	M	W															
sub r2, r2, #32	1				F	D	E	M	W														
sub ip, ip, r2	1					F	D	D	D	E	M	W											
cmp r3, #0	1						F	F	F	D	E	M	W										
asr lr, r5, ip	1									F	D	E	M	W									
add r7, r1, r4, lsl #2	1										F	D	E	M	W								
add r0, r4, #1	1											F	D	E	M	W							
beq .L2	3												F	D	E	E	E	M	W				
mov r8, r4	1		F	D	E	M	W																
mov r4, r0	1			F	D	E	M	W															
b .L3	3				F	D	E	E	E	M	W												
ldrb r3, [r6, #1]!	1	L4	F	D	E	M	W																
add r7, r1, r8, lsl #2	1			F	D	E	M	W															
cmp r3, #0	1				F	D	E	M	W														
add r4, r8, #1	1					F	D	E	M	W													
bne .L3	3						F	D	E	E	E	M	W										
mov r0, r4	1	L2	F	D	E	M	W																
str lr, [r7]	1			F	D	E	M	W															
mov r7, r1	1	L6	F	D	E	M	W																
mov lr, r3	1			F	D	E	M	W															
mov r0, #1	1				F	D	E	M	W														
str lr, [r7]	1					F	D	E	M	W													

Appendix B

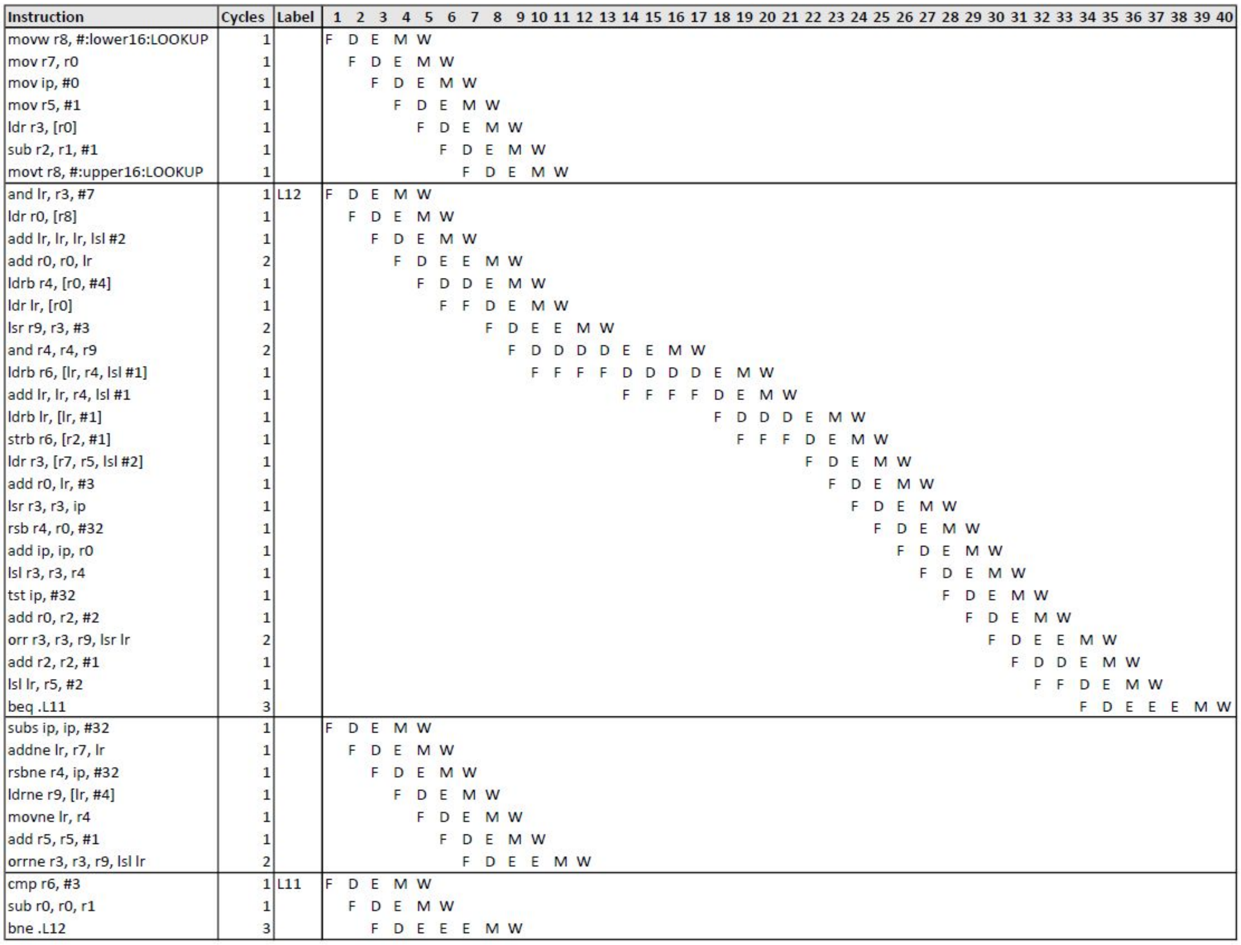
Simulated Processor Pipeline of Optimized Encode Function



Instruction	Cycles	Label	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
movw r2, #:lower16:DICT	1		F	D	E	M	W																						
ldrb r3, [r0]	1			F	D	E	M	W																					
movt r2, #:upper16:DICT	1				F	D	E	M	W																				
cmp r3, #0	1					F	D	E	M	W																			
mov r4, r1, lsr #2	1						F	D	E	M	W																		
mov r6, #0	1							F	D	E	M	W																	
beq .L4	3								F	D	E	E	E	M	W														
ldr r7, [r2]	1									F	D	D	D	E	M	W													
add r3, r3, r3, lsl #1	1										F	F	F	D	E	M	W												
mov r5, #0	1													F	D	E	M	W											
add r3, r7, r3	1														F	D	E	M	W										
ldrh r8, [r3]	1	L3	F	D	E	M	W																						
ldrb r9, [r3, #2]	1			F	D	E	M	W																					
orr r6, r6, r8, lsl r5	2				F	D	E	E	M	W																			
add r5, r5, r9	1					F	D	D	E	M	W																		
ldrb r3, [r0, #1]!	1						F	F	D	E	M	W																	
tst r5, #32	1								F	D	E	M	W																
subne r5, r5, #32	1									F	D	E	M	W															
strne r6, [r1], #4	1										F	D	E	E	M	W													
subne r9, r9, r5	1											F	D	D	E	M	W												
lsrne r6, r8, r9	1												F	F	D	D	D	E	M	W									
cmp r3, #0	1														F	F	F	D	E	M	W								
add r3, r3, r3, lsl #1	2																	F	D	E	E	M	W						
add r3, r7, r3	1																		F	D	D	D	D	E	M	W			
bne .L3	3																			F	F	F	F	D	E	E	E	M	W
str r6, [r1], #4	1	L4	F	D	E	M	W																						
rsb r0, r4, r1, lsr #2	1			F	D	E	M	W																					

Appendix C

Simulated Processor Pipeline of GCC Decode Function



Instruction	Cycles	Label	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
movw r8, #:lower16:LOOKUP	1		F	D	E	M	W																																			
mov r7, r0	1			F	D	E	M	W																																		
mov ip, #0	1				F	D	E	M	W																																	
mov r5, #1	1					F	D	E	M	W																																
ldr r3, [r0]	1						F	D	E	M	W																															
sub r2, r1, #1	1							F	D	E	M	W																														
movt r8, #:upper16:LOOKUP	1								F	D	E	M	W																													
and lr, r3, #7	1	L12	F	D	E	M	W																																			
ldr r0, [r8]	1			F	D	E	M	W																																		
add lr, lr, lr, lsl #2	1				F	D	E	M	W																																	
add r0, r0, lr	2					F	D	E	E	M	W																															
ldrb r4, [r0, #4]	1						F	D	D	E	M	W																														
ldr lr, [r0]	1							F	F	D	E	M	W																													
lsr r9, r3, #3	2									F	D	E	E	M	W																											
and r4, r4, r9	2										F	D	D	D	D	E	E	M	W																							
ldrb r6, [lr, r4, lsl #1]	1											F	F	F	F	D	D	D	D	E	M	W																				
add lr, lr, r4, lsl #1	1															F	F	F	F	D	E	M	W																			
ldrb lr, [lr, #1]	1																			F	D	D	D	E	M	W																
strb r6, [r2, #1]	1																				F	F	F	D	E	M	W															
ldr r3, [r7, r5, lsl #2]	1																							F	D	E	M	W														
add r0, lr, #3	1																								F	D	E	M	W													
lsr r3, r3, ip	1																									F	D	E	M	W												
rsb r4, r0, #32	1																										F	D	E	M	W											
add ip, ip, r0	1																											F	D	E	M	W										
lsl r3, r3, r4	1																												F	D	E	M	W									
tst ip, #32	1																													F	D	E	M	W								
add r0, r2, #2	1																														F	D	E	M	W							
orr r3, r3, r9, lsr lr	2																															F	D	E	E	M	W					
add r2, r2, #1	1																																F	D	D	E	M	W				
lsl lr, r5, #2	1																																	F	F	D	E	M	W			
beq .L11	3																																			F	D	E	E	E	M	W
subs ip, ip, #32	1		F	D	E	M	W																																			
addne lr, r7, lr	1			F	D	E	M	W																																		
rsbne r4, ip, #32	1				F	D	E	M	W																																	
ldrne r9, [lr, #4]	1					F	D	E	M	W																																
movne lr, r4	1						F	D	E	M	W																															
add r5, r5, #1	1							F	D	E	M	W																														
orrne r3, r3, r9, lsl lr	2								F	D	E	E	M	W																												
cmp r6, #3	1	L11	F	D	E	M	W																																			
sub r0, r0, r1	1			F	D	E	M	W																																		
bne .L12	3				F	D	E	E	E	M	W																															

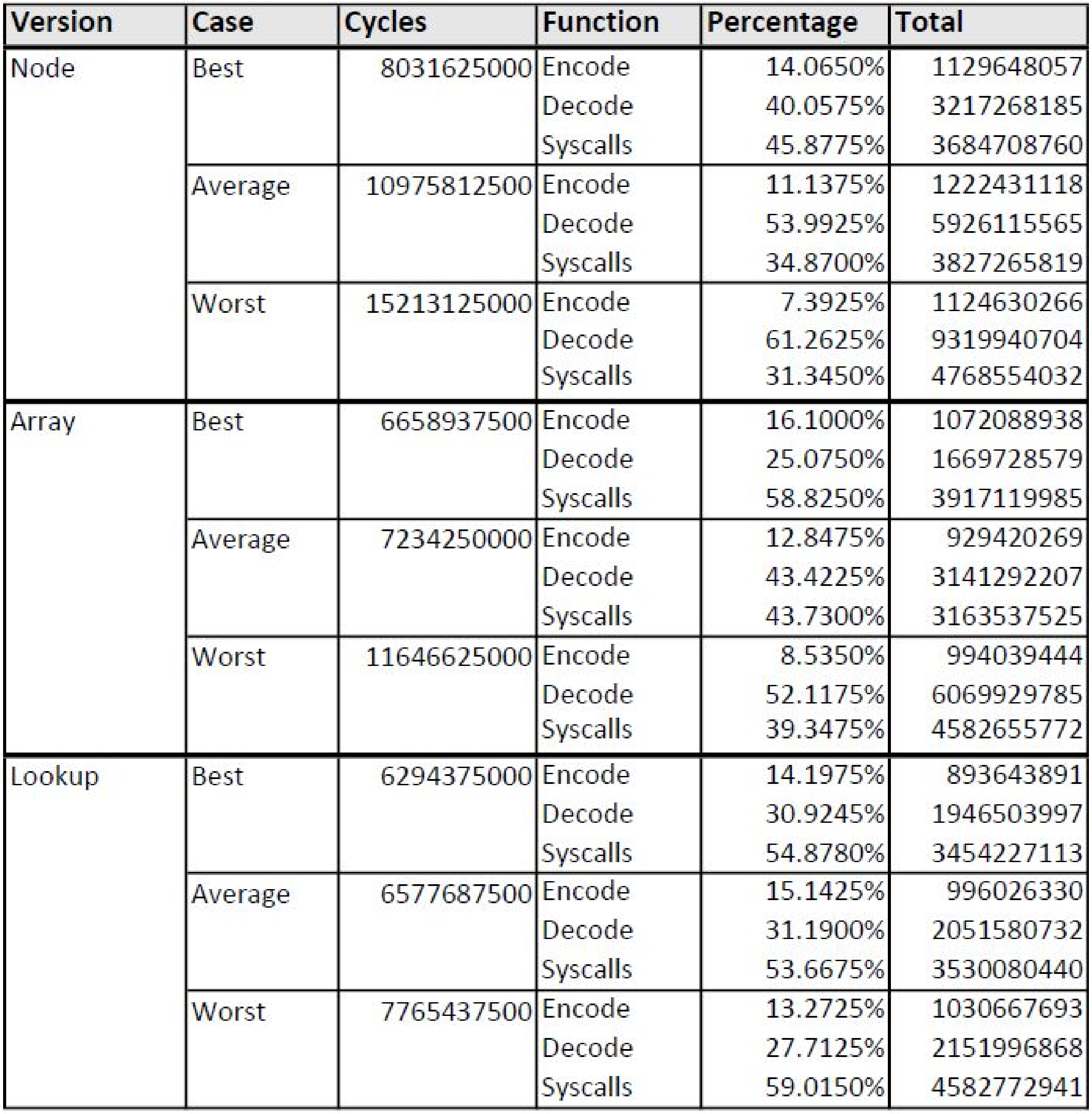
Appendix D

Simulated Processor Pipeline of Optimized Decode Function

Instruction	Cycles	Labels	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
movw r8, #:lower16:LOOKUP	1		F	D	E	M	W																							
mov r5, #0	1			F	D	E	M	W																						
ldr r9, [r0]	1				F	D	E	M	W																					
ldr r3, [r0, #4]!	1					F	D	E	M	W																				
mov r2, r1	1						F	D	E	M	W																			
movt r8, #:upper16:LOOKUP	1							F	D	E	M	W																		
and lr, r9, #7	1								F	D	E	M	W																	
ldr r7, [r8]	1									F	D	E	M	W																
add lr, lr, lr, lsl #2	1	L12									F	D	E	M	W															
add r7, r7, lr	1		F	D	E	M	W																							
lsr r9, r9, #3	1			F	D	E	M	W																						
ldrb r4, [r7, #4]	1				F	D	E	M	W																					
ldr lr, [r7]	1					F	D	E	M	W																				
and r4, r4, r9	1						F	D	E	M	W																			
add lr, lr, r4, lsl #1	1							F	D	D	D	E	M	W																
ldrb r6, [lr]	1								F	F	F	D	D	D	E	M	W													
ldrb lr, [lr, #1]	1											F	F	F	D	E	M	W												
strb r6, [r1], #1	1														F	D	E	M	W											
lsr r9, r9, lr	1															F	D	E	M	W										
add r7, lr, #3	1																F	D	E	M	W									
lsr lr, r3, r5	1																	F	D	E	M	W								
add r5, r5, r7	1																		F	D	E	M	W							
rsb r4, r7, #32	1																			F	D	E	M	W						
tst r5, #32	1																				F	D	E	M	W					
orr r9, r9, lr, lsl r4	2																					F	D	E	E	M	W			
beq .L11	3																						F	D	D	E	E	E	M	W
subs r5, r5, #32	1		F	D	E	M	W																							
ldr r3, [r0, #4]!	1			F	D	E	M	W																						
rsbne r4, r5, #32	1				F	D	E	M	W																					
orrne r9, r9, r3, lsl r4	2					F	D	D	D	E	E	M	W																	
cmp r6, #3	1	L11	F	D	E	M	W																							
and lr, r9, #7	1			F	D	E	M	W																						
ldr r7, [r8]	1				F	D	E	M	W																					
add lr, lr, lr, lsl #2	1					F	D	E	M	W																				
bne .L12	3						F	D	E	E	E	M	W																	
sub r0, r1, r2	1		F	D	E	M	W																							

Appendix E

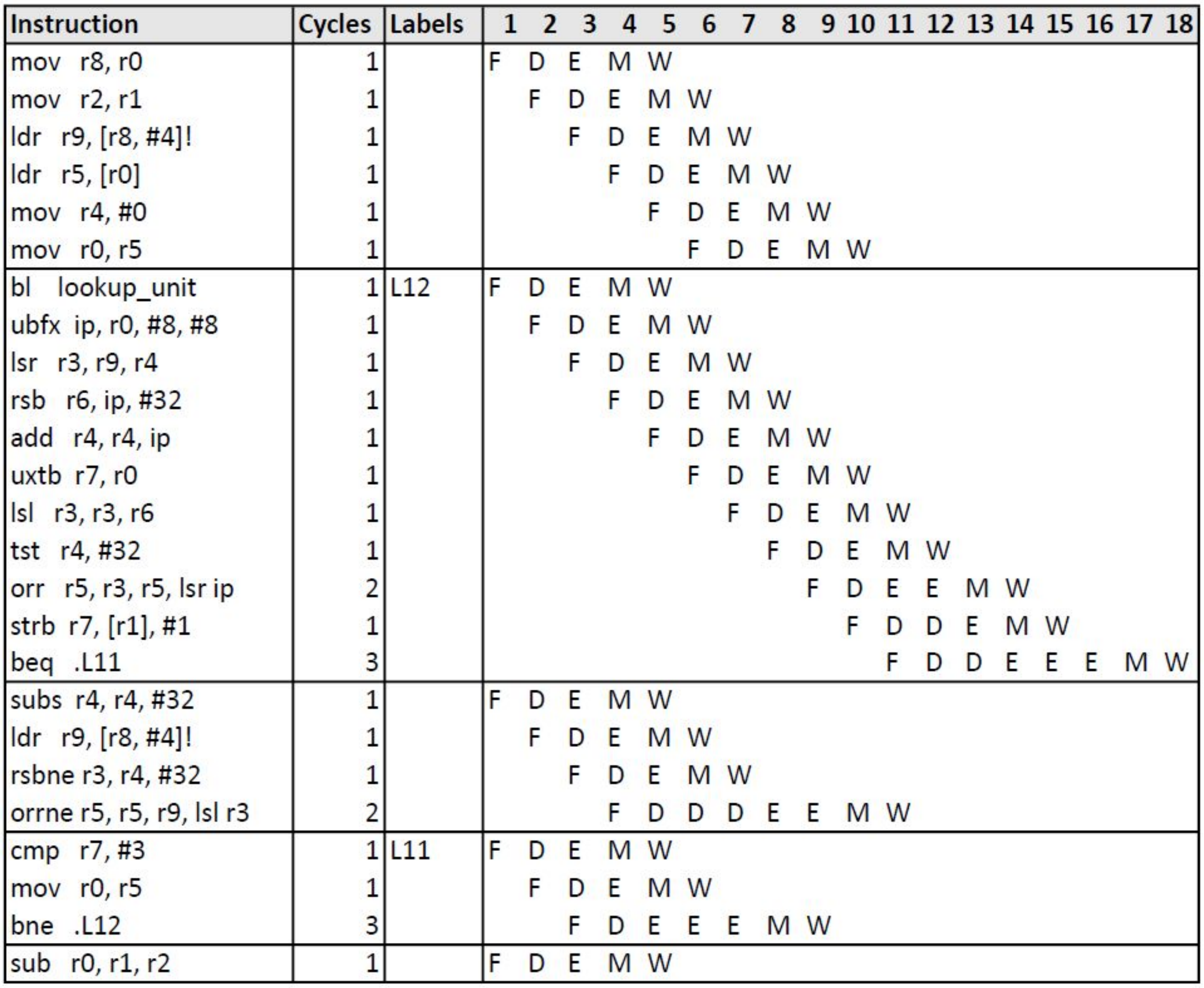
Aggregated Performance Statistics for the Three Executable Versions



Version	Case	Cycles	Function	Percentage	Total
Node	Best	8031625000	Encode	14.0650%	1129648057
			Decode	40.0575%	3217268185
			Syscalls	45.8775%	3684708760
	Average	10975812500	Encode	11.1375%	1222431118
			Decode	53.9925%	5926115565
			Syscalls	34.8700%	3827265819
	Worst	15213125000	Encode	7.3925%	1124630266
			Decode	61.2625%	9319940704
			Syscalls	31.3450%	4768554032
Array	Best	6658937500	Encode	16.1000%	1072088938
			Decode	25.0750%	1669728579
			Syscalls	58.8250%	3917119985
	Average	7234250000	Encode	12.8475%	929420269
			Decode	43.4225%	3141292207
			Syscalls	43.7300%	3163537525
	Worst	11646625000	Encode	8.5350%	994039444
			Decode	52.1175%	6069929785
			Syscalls	39.3475%	4582655772
Lookup	Best	6294375000	Encode	14.1975%	893643891
			Decode	30.9245%	1946503997
			Syscalls	54.8780%	3454227113
	Average	6577687500	Encode	15.1425%	996026330
			Decode	31.1900%	2051580732
			Syscalls	53.6675%	3530080440
	Worst	7765437500	Encode	13.2725%	1030667693
			Decode	27.7125%	2151996868
			Syscalls	59.0150%	4582772941

Appendix F

Simulated Processor Pipeline of Optimized Decode Function Using the Lookup Table Cache Functional Unit



Instruction	Cycles	Labels	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
mov r8, r0	1		F	D	E	M	W													
mov r2, r1	1			F	D	E	M	W												
ldr r9, [r8, #4]!	1				F	D	E	M	W											
ldr r5, [r0]	1					F	D	E	M	W										
mov r4, #0	1						F	D	E	M	W									
mov r0, r5	1							F	D	E	M	W								
bl lookup_unit	1	L12	F	D	E	M	W													
ubfx ip, r0, #8, #8	1			F	D	E	M	W												
lsr r3, r9, r4	1				F	D	E	M	W											
rsb r6, ip, #32	1					F	D	E	M	W										
add r4, r4, ip	1						F	D	E	M	W									
uxtb r7, r0	1							F	D	E	M	W								
lsl r3, r3, r6	1								F	D	E	M	W							
tst r4, #32	1									F	D	E	M	W						
orr r5, r3, r5, lsr ip	2										F	D	E	E	M	W				
strb r7, [r1], #1	1											F	D	D	E	M	W			
beq .L11	3												F	D	D	E	E	E	M	W
subs r4, r4, #32	1		F	D	E	M	W													
ldr r9, [r8, #4]!	1			F	D	E	M	W												
rsbne r3, r4, #32	1				F	D	E	M	W											
orrne r5, r5, r9, lsl r3	2					F	D	D	D	E	E	M	W							
cmp r7, #3	1	L11	F	D	E	M	W													
mov r0, r5	1			F	D	E	M	W												
bne .L12	3				F	D	E	E	E	M	W									
sub r0, r1, r2	1		F	D	E	M	W													