Behram Mistree & Dmitry Kashlev
6.375 Final Project Report
GZIP Encoding and Decoding in Hardware

GZIP Introduction

GZIP is a software application used for file compression. It is widely used by many UNIX systems. GZIP provides an effectively lossless data compression. GZIP is based on DEFLATE algorithm which is a combination of LZ77 and Huffman coding. The DEFLATE algorithm is specified in RFC1951, while GZIP file format is specified by RFC1952.

Figure 1 below describes the general architecture of GZIP. GZIP contains an encoder and a decoder, each of which have Huffman and LZ77 modules. The text input is fed into LZ77 encoder, and output of LZ77 encoder is connected to the input of Huffman encoder. For the purpose of our final project we connected the output of Huffman encoder to input of Huffman decoder. This is where the data becomes compressed. The output of Huffman decoder, in turn, is connected to input of LZ77 decoder. The output of LZ77 decoder is the uncompressed text.

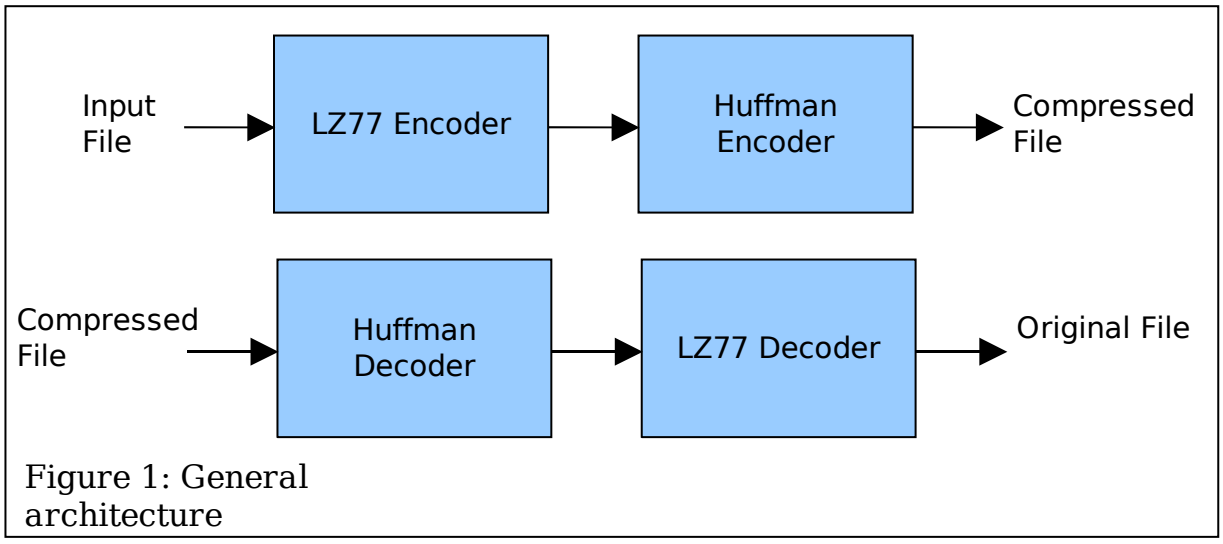


Figure 1: General architecture

LZ77 Overview:

Suppose that you wanted to encode a file that consisted of the sentence "I really, really want to encode a sentence," repeated 1000 times. You could determine the optimal way of encoding the sentence and then repeat that optimal encoding 1000 times. LZ77 eschews this method and instead encodes the first appearance of the sentence and then replaces all subsequent repetitions of the original sentence with "pointers" to the first sentence. For instance:

I really, really want to encode a sentence
I really, really want to encode a sentence
I really, really want to encode a sentence
I really, really want to encode a sentence
I really, really want to encode a sentence
I really, really want to encode a sentence

would be encoded as:

I really, really want to encode a sentence
<Pointer to sentence 1>
<Pointer to sentence 1>
<Pointer to sentence 1>

<Pointer to sentence 1>
<Pointer to sentence 1>

A pointer has two parts to its structure: a distance and a length. The distance is a value ranging from 1 to 32,768 and corresponds to how many spaces prior to the current position the repeated snippet of text is located. The length is a value ranging from 3 to 258 bytes that represents how long the string to insert is. Revisiting our example, because the length of the "I really, really want to encode a sentence" is 42, we would get:

I really, really want to encode a sentence
<Distance: 42; Length: 42>
<Distance: 42; Length: 42>
<Distance: 42; Length: 42>
<Distance: 42; Length: 42>
<Distance: 42; Length: 42>

The basic principle behind LZ77 encoding is that one can replace a phrase that appears multiple times in a text with a pointer to the previous occurrence of that same piece of text.

The LZ77 specification requires that the encoder have access to either 2KB, 4KB, 8KB, or 32KB of data prior to the current character or word being encoded. Because the default option for most software implemented GZIP encoders requires access to 32KB of data, we built our hardware using a 32KB window of accessible values. (However, a single trivial parameter change in our code would allow for the 2KB, 4KB, and 8KB cases.)

We need some way to keep track of the previous 32KB of data. To accomplish this goal, we built a separate RAM module that allows reads and writes. To assure a good model of real-world memory, this

decoupled memory module does not allow combinational reads.

We built a controller module for the RAM module called Memory Checker. Its primary purpose is to check whether valid locations in memory contain characters equal to those passed into it by calls to its methods. In addition, Memory Checker is responsible for ensuring that, during writes, characters are inserted into the correct positions in the RAM memory module.

The interface with comments for the Memory Checker module is provided below.

```
method Action       write_next( Char data );
```

write_next takes in a character of data and puts it in the appropriate space in memory.

```
method Action check_equals (Rindx rindx, Char val);
method ActionValue#(Bool) get_equals();
```

check_equals takes in an memory index position (rindx) and a character. The corresponding call get_equals returns whether the character value passed in was equal to the element stored in memory at position rindx.

```
method Action set_find_triplet( Char data1, Char data2, Char data3 );
method ActionValue#(Maybe#(TripletReturner)) get_find_triplet();
```

set_find_triplet takes in three characters. The corresponding call to get_find_triplet returns an invalid data type if all three characters were not found consecutively in the memory. If all three characters were found consecutively in the memory, get_find_triplet returns the index of the memory where the first of the three characters were found tagged valid.

The set_find_triplet and get_find_triplet methods are the crux of the Memory Check module. Figure 2 presents a high level block diagram of the inner workings of Memory Check when set_find_triplet is called. Briefly:

1. **set_find_triplet:** A call to set_find_triplet initializes register search_ptr to the earliest memory index written to.

2. **memory request:** The Memory Check module requests data from the locations in memory with indices search_ptr, search_ptr +1, and search_ptr +2.
3. **Memory Response:** Memory Response returns values.
    a) If the memory response contains values equal all three of the values passed in to set_find_triplet, then we set get_find_triplet to return search_ptr tagged valid.
    b) **Check search_ptr:** If the memory response contains values that do not equal all three of the values passed in to set_find_triplet, then we increment search_ptr.
        i. If we already checked the character value stored in the RAM index search_ptr +2, then we set get_find_triplet to return invalid data.
        ii. If we have not already checked the character value stored in the RAM index search_ptr +2, then we go back to step 2, memory request.

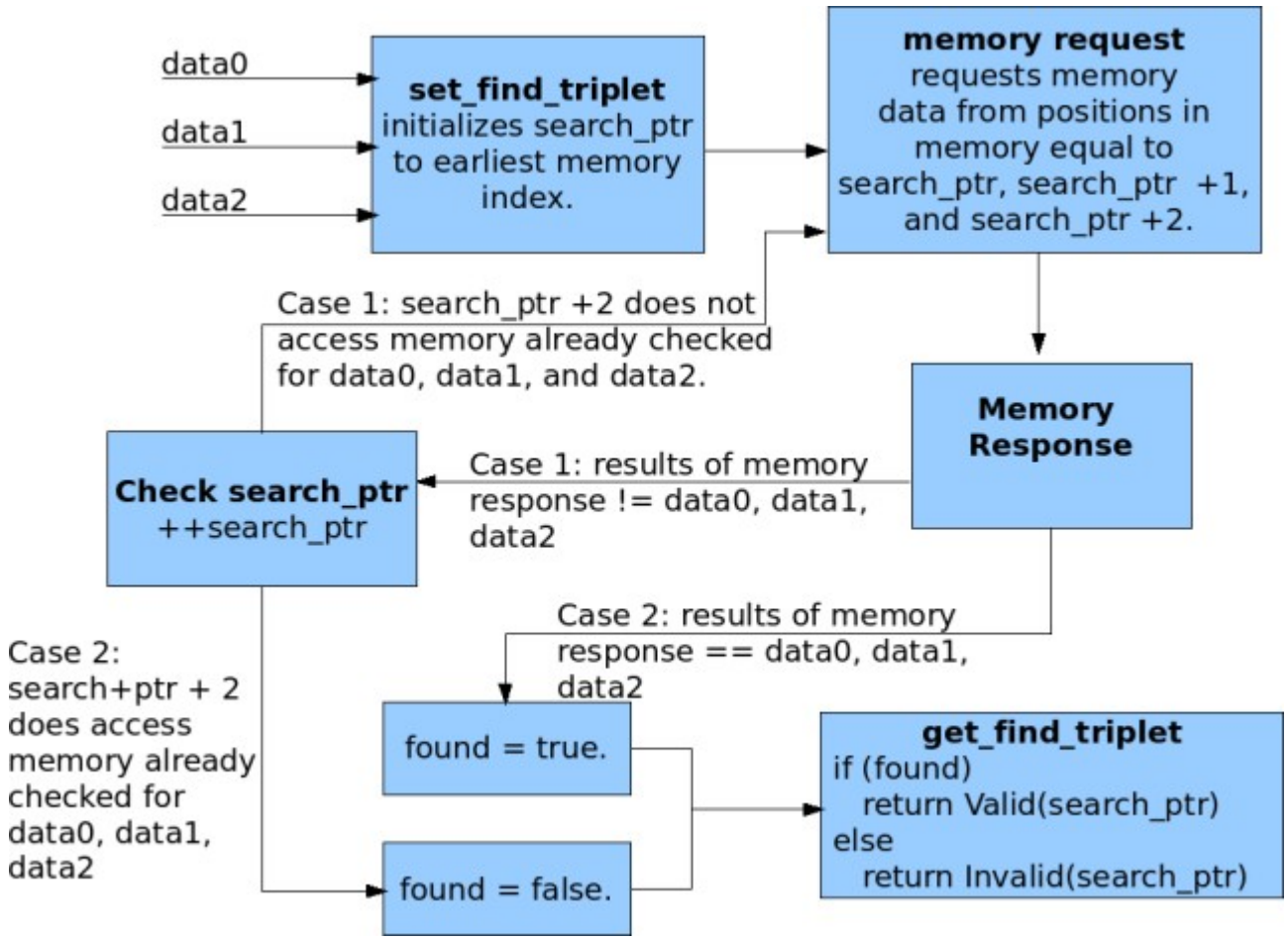


Figure 2: High level block diagram of logic for set_find_triplet and get_find_triplet methods of Memory Checker.

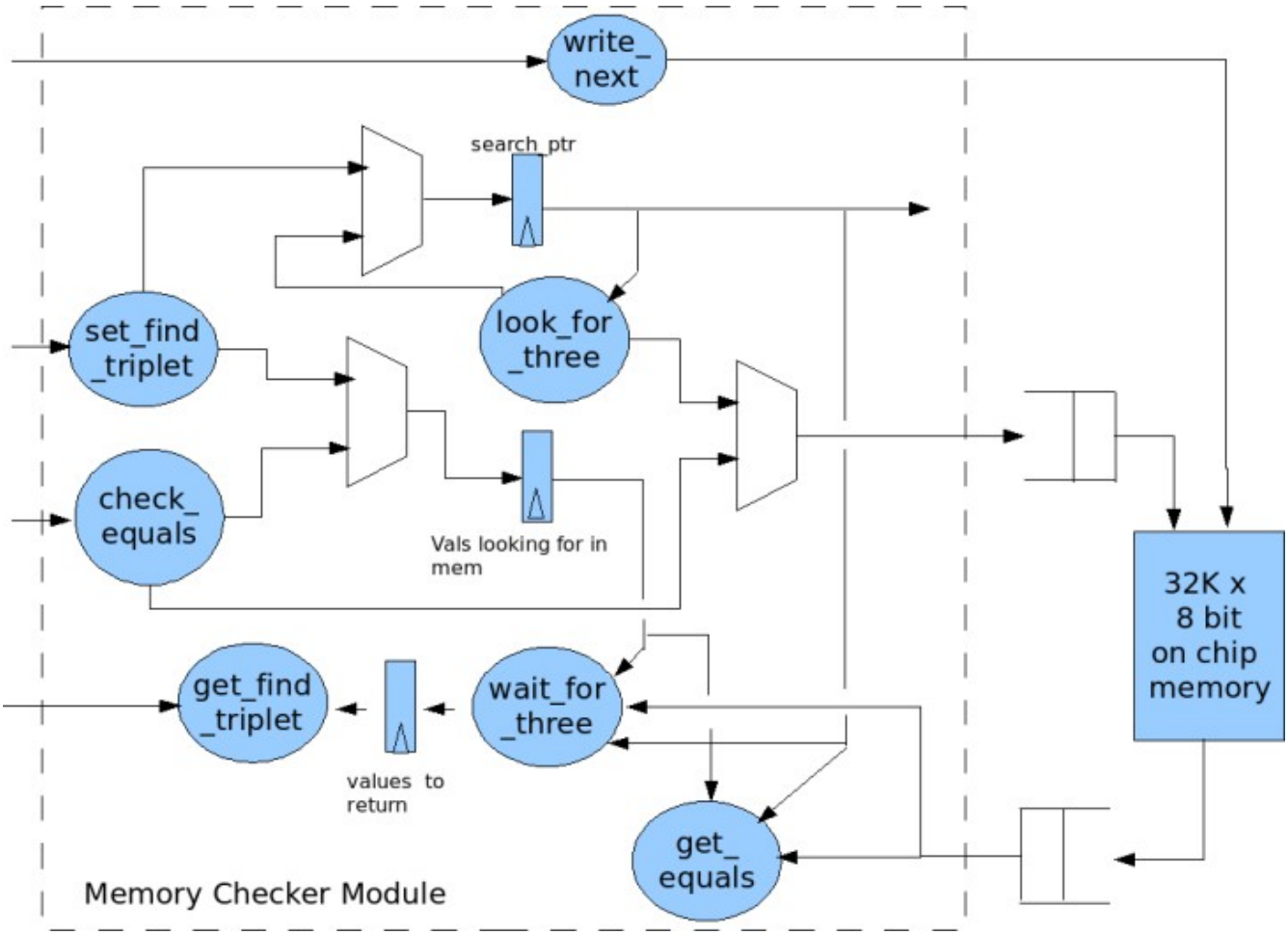


Figure 3: Semi-hardware level diagram of Memory Checker module.

We have created complete drawings by hand of the hardware generated by the LZ77 encoder (minus compiler optimizations). They provide a very detailed and specific understanding of how the hardware for the LZ77 encoder works. Such detail in this report is not necessary, therefore, you can access the drawings at http://bmistre.mit.edu/6.375/lz77Hardware.pdf. Of particular interest however may be the hardware for the state variable (which provides the primary guard on most of these rules pictured above). Therefore, we have added the hardware fore the LZ77 encoder's Memory Checker module in Appendix A.

The third and last encoding module is titled Lz77_Encode. Lz77_Encode provides a get/put interface to an external source. The get element of the interface for Lz77_Encode reads in a character at a time from some external source, while the put element outputs the encoded data. The logic behind the Lz77 encoding is presented as a high level block diagram in Figure 4. Below is the enumerated explanation of the encoding:

1. **Initialization:** Reads three values from external file into first[0], first[1], and first[2].
2. **check_equals:** Sends a request to Memory Check using the set_find_triplet method described above. data0, data1, and data2 from set_find_triplet are equal to first[0], first[1], and first[2] respectively.
    a) If Memory Checker returns an invalid memory index, then we cannot generate a pointer that points to any previous values of first[0], first[1], and first[2] because first[0], first[1], and first[2] do not appear consecutively in the previous 32Kb of data passed into the RAM.
        i. **Write first[0]:** Because we know that the consecutive characters first[0], first[1], and first[2] cannot be replaced by a pointer to a previous position in memory, we write first[0] to our 32Kb window of data through a call to Memory Checker's write_next function. We also write first[0] through the put interface indicating that there is no compression that occurs on the character in first[0].
        ii. **Shift and Read new value:** We shift the values in first and load a new character in from the get interface so that we can look for a new set of three characters in the previous 32Kb

of memory.

b) If Memory Checker returns a valid memory index, then the characters represented by first[0], first[1], and first[2] can be replaced by a pointer to a previous occurrence of these three characters.

   i. **look_for_more:** Instead of just writing the characters contained in first to the output as a pointer straightaway, we first want to check whether we can "grow" the pointer by matching additional characters coming from the get interface with those that follow the previous occurrence of first in the 32Kb of data.

   ii. **check_equals:** We peak at the next value coming from the external file and send this value to Memory Checker along with potential index that would grow the pointer.

      I. **Write:** If Memory Checker returns that the potential index that would grow the pointer is not populated with the next value that we are receiving from the get interface, then we write out our pointer through the put interface.

      II. If Memory Checker returns that the potential index that would grow the pointer is populated with the next value that we are receiving from the get interface, then we dequeue from get. In addition, we increment the memory index that we are sampling to grow the pointer. We then go back to check_equals with a new value in our get interface and a new position to check in memory.

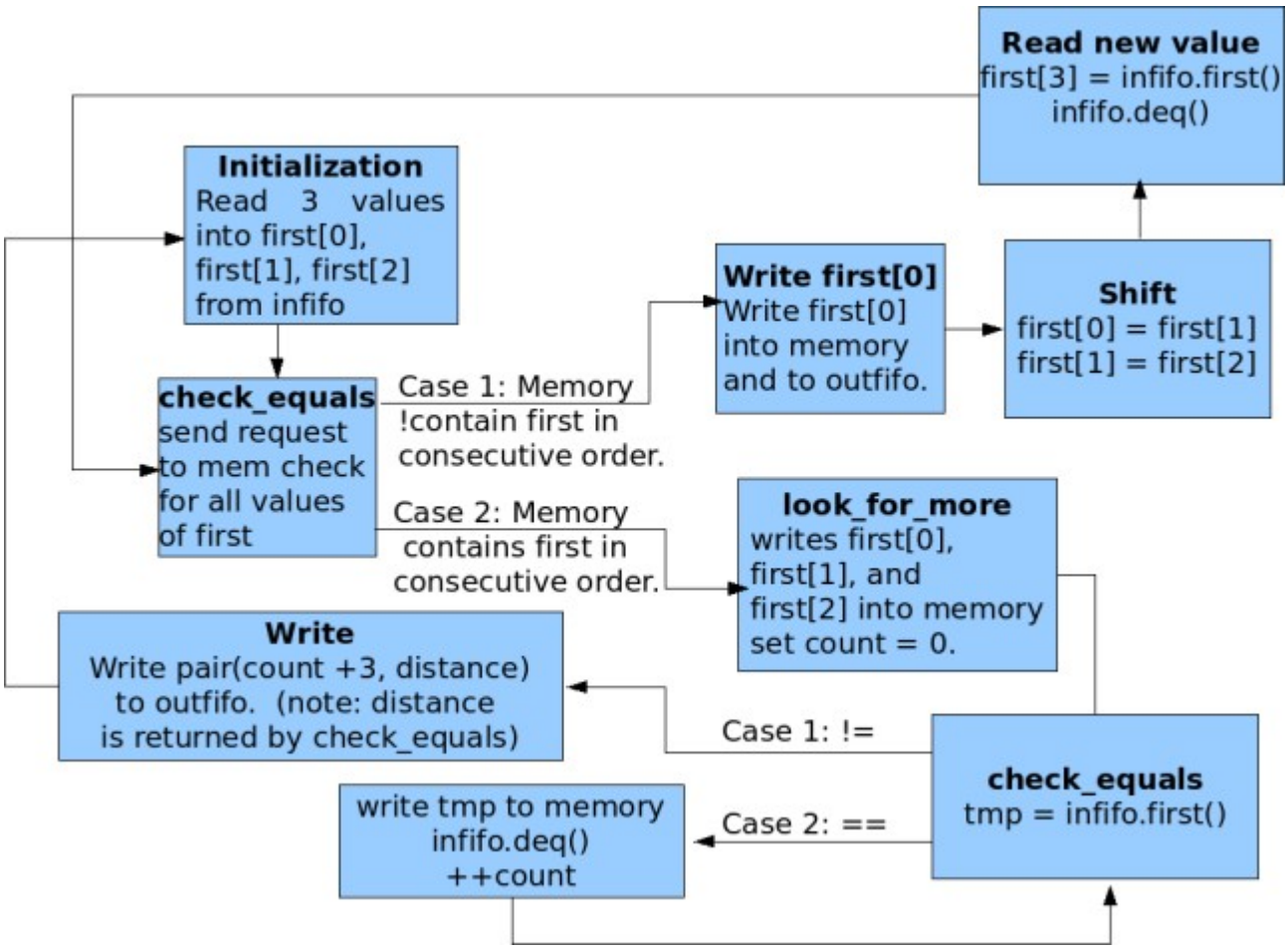


Figure 4: High level block diagram for functioning of Lz77_Encode module which encodes character data passed into it using the get/put interface.

A state variable provides an explicit guard that controls the transitions between these states. We have included the specific hardware of these state transitions in Appendix B.

LZ77 Decode:

Compared to encoding, decoding is fairly straightforward. The decoding aspect of LZ77 is split into two modules. One module, Decompressed Window, keeps track of the last 32Kb of decoded data. Specifically, Decompressed Window provides methods for writing new decoded data into its 32Kb

window and returning characters from pointers for the other module, Lz77_Decode to write out. The logic for writing a character for module Decompressed Window is trivial – we simply write the new character into the next available spot in the 32Kb window. The complexity of Decompressed Window comes from decoding pointers. Figure 5 provides a high level block diagram for the logic necessary to decode a pointer.

1. **set_return_string:** set_return_string is called by Lz77_Decode. The two arguments that it is passed correspond to the distance and length values of a particular pointer. Decode Window keeps track of the next available position in memory to write a value into (called window_end). Therefore, the first character that a pointer points to (labeled start_position in the diagram) should be equal to window_end – dist. Count is a register that keeps track of the number of characters that should match the pointer. That is, we know to exit when count is equal to length.
2. **filling_string:** filling_string simply makes a memory request. Note that because count is incremented each time, filling_string effectively makes a memory request once for each character that is represented by the pointer.
3. **get_return_string:** get_return_string is an ActionValue method.
    a) If count + start_pos is equal to runUntil, get_return_string returns an invalid character and sets the system state so that Lz77_Decode can call set_return_string again.
    b) If count + start_pos is not equal to runUntil, get_return_string returns a valid character. In addition, we increment counter so that future memory requests will look for the next value in

memory. We then return to the filling_string state.

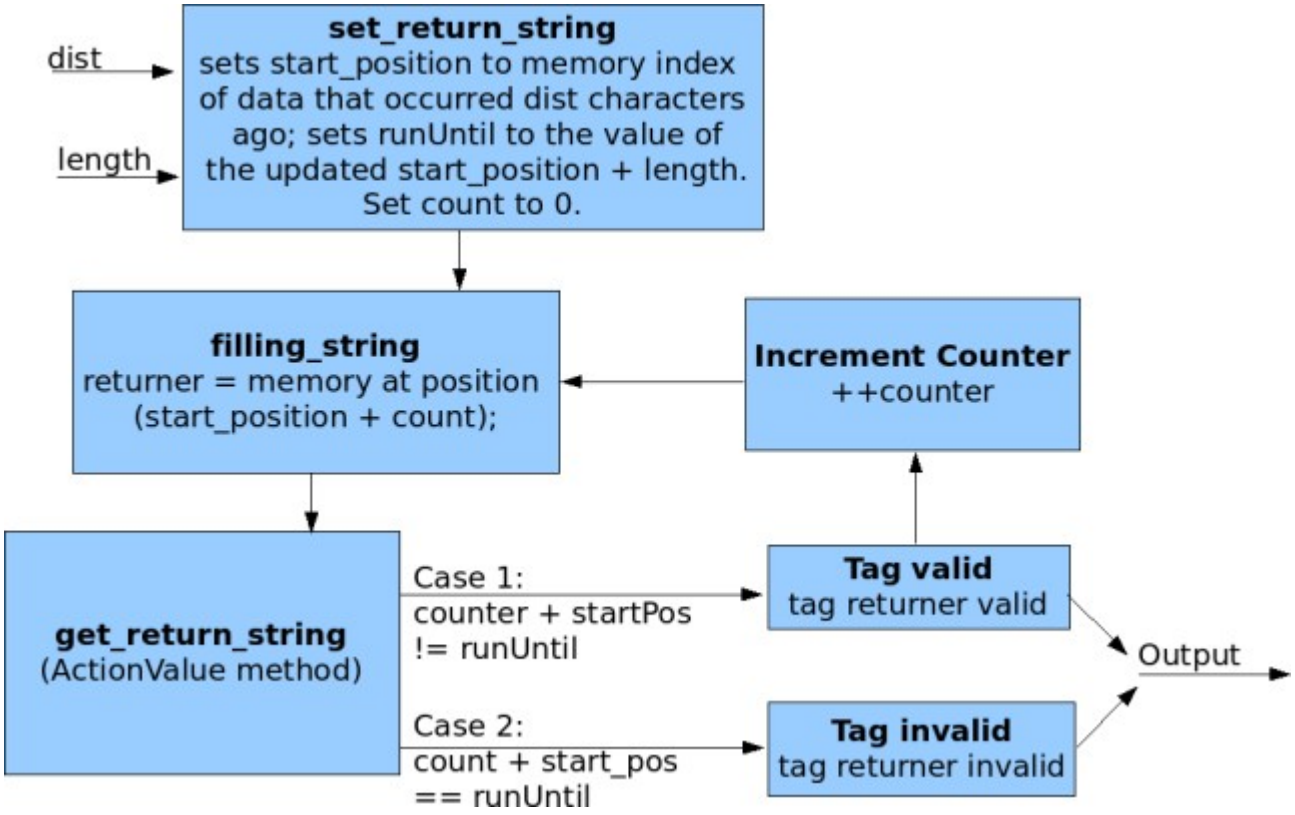


Figure 5: High level block diagram for Decode Window's set_return_string and get_return_string.

The Lz77_Decode module provides an interface for putting encoded values through the Decompressed Window module.

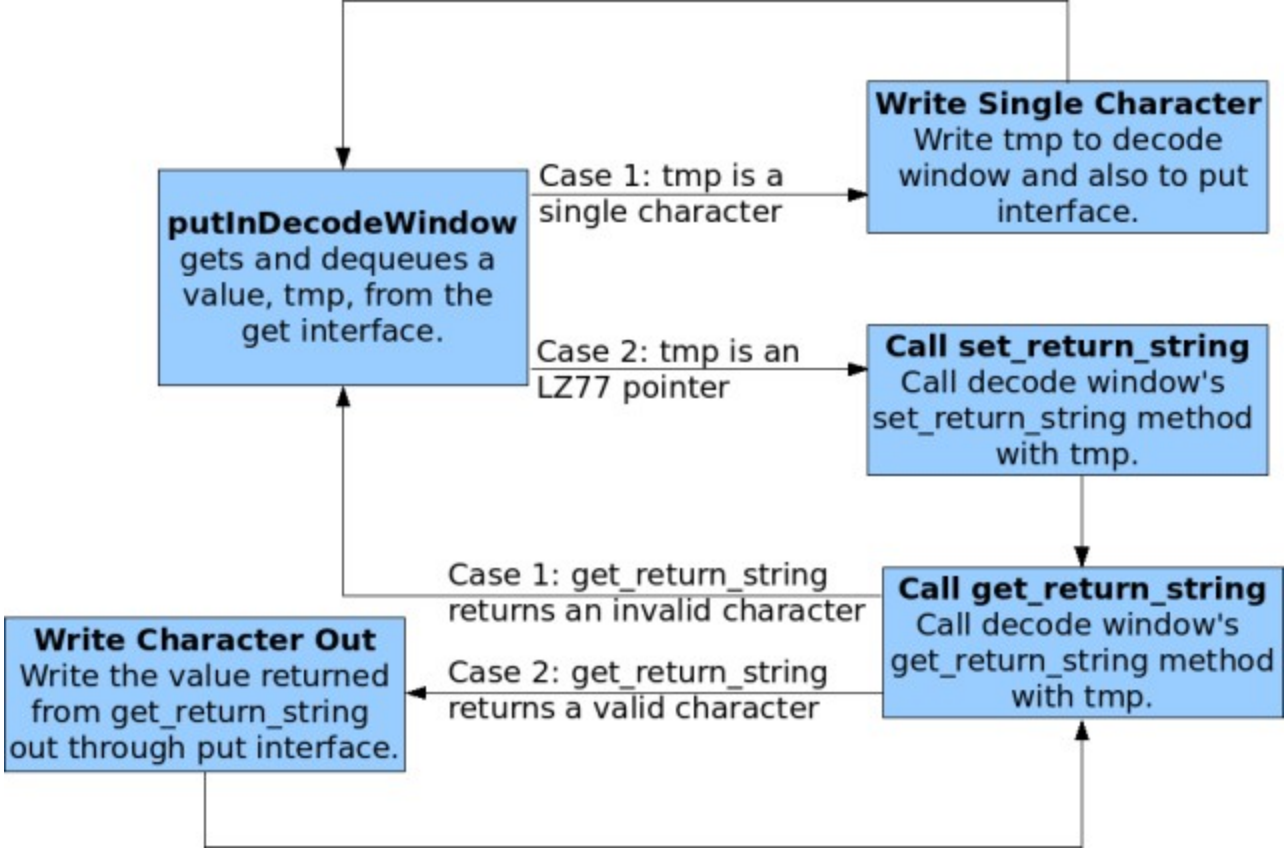
1. **putInDecodedWindow:** reads and dequeues a value from the Lz77_Decode module's get interface. The value may either be a single character, or a pointer, representing a string of characters.

   a) **Write Single Character:** The case that tmp is a single character is uninteresting – there is

nothing to decode, we simply write the character into the Decode Window module and also write its value through the put interface.

b) **Call set_return_string:** When tmp is an LZ77 pointer, we need to decode the pointer. Fortunately, the Decode Window module provides logic that does this decoding for us. We simply call Decode Window's set_return_string method and wait for a series of character responses from Decode Window's get_return_string.

Figure 6: High level block diagram for Lz77_Decode.



Again, for a thorough treatment of the hardware necessary for decoding LZ77 please see http://bmistre.mit.edu/6.375/lz77Hardware.pdf.

LZ77 Exploration:

We went through several iterations of LZ77 design. Our emphasis was on making the encoding and decoding faster through algorithmic improvements. Because our encoder was our primary bottleneck (taking the maximum number of clock ticks out of all the elements to process input), we focused on streamlining the encoder.

As we learned in previous labs, some of the greatest slow down occurs from reading from memory. Initially, we read only single characters from memory, progressively comparing these values to those to be encoded. However, because memory hardware should be able to support more than

a single character read per clock cycle, we changed our rules to read more memory values and compare them against our values.

LZ77 Results:

We put the *Communist Manifesto*, a file consisting of 108,673 characters, into our LZ77 encoder. Our LZ77 encoder returned an output that consisted of 27,052 character or pointer values (3204 single, unencoded characters; and 23848 pointer values).

If we assume that each pointer pair is 30 bits (15 bits to represent 32K potential distance values and 15 bits to represent 32K potential length values), then our LZ77 encoded file is now 92,634 bytes long. Therefore, if we just use LZ77 compression with no other techniques, our encoded file is 85% the size of our unencoded file. These results are quite promising as Huffman encoding should reduce this number still further.

After performing our optimizations, the total area of our LZ77 encoder was 4051.00 μm^2. Please note that because we were not able to determine how to use the Encounter tool to place memory on our chip, this figure excludes the size of our memory 32KB on-chip memory.

The critical path for our LZ77 encoder was also 3.93 ns. This critical path corresponded to dequeuing from the input fifo that is sending unencoded characters and sending those characters to the memory manager.

The total area of our LZ77 decoder (again excluding memory) was 2015.25 μm^2. Strangely, the

decoder also had a critical path of 3.93 ns. This critical path corresponds to the path from reading a value from the fifo outputting encoded values, and passing that value to the decoding window of 32KB described above.

Huffman Modules:

The Huffman part of the GZIP algorithm consists of 2 major parts – encoding and decoding. Encoder takes ascii characters as input and produces a sequence of bits'0's and '1's as output. Decoder reads the sequence of bits and determines the characters based on the binary tree representation. Both Decoder and Encoder require prior knowledge of bit-sequence to character mappings and vice versa. This is why a static Huffman table is generated before encoding and static Huffman binary tree is generated before decoding.

Before we delve deeper into the subject of Huffman decoding in gzip, I will describe what Huffman codeword actually is. A Huffman codeword is a sequence of bits, or "0"s and "1"s that have variable length. The variable length of Huffman code is one most important reason why Huffman coding has been so successful in the field of data compression. The characters that appear most frequently have shortest Huffman codeword, while characters that are used rarely get longer Huffman codeword. Instead of representing each character in a binary format, we can also add another degree of freedom, which is a bit length. While to a normal human, 0011 and 11 seem to mean the same number, 3, in Huffman context they actually are two different Huffman codewords. To sum it up, Huffman codeword is a sequence of bits

that is of variable length.

Huffman code is a prefix-free code. This means that there is no fixed length dictated for every Huffman codeword. If given a Huffman string, it is necessary to start at the beginning of the stream to read the Huffman code. Starting in the middle will not work because the location of the end of previous Huffman code in this stream is not known.

Huffman trees are used in decoding Huffman streams. The usage of the binary tree means that the character is obtained when the tree iteration algorithm hits a leaf. Every leaf in the tree contains the ascii character that is represented by a Huffman codeword that can be determined by going from the root node to the leaf. Since every node in a binary tree can have only 2 branches, each branch taken is either a "0" or a "1". See Figure 7 for the illustration of Huffman tree.

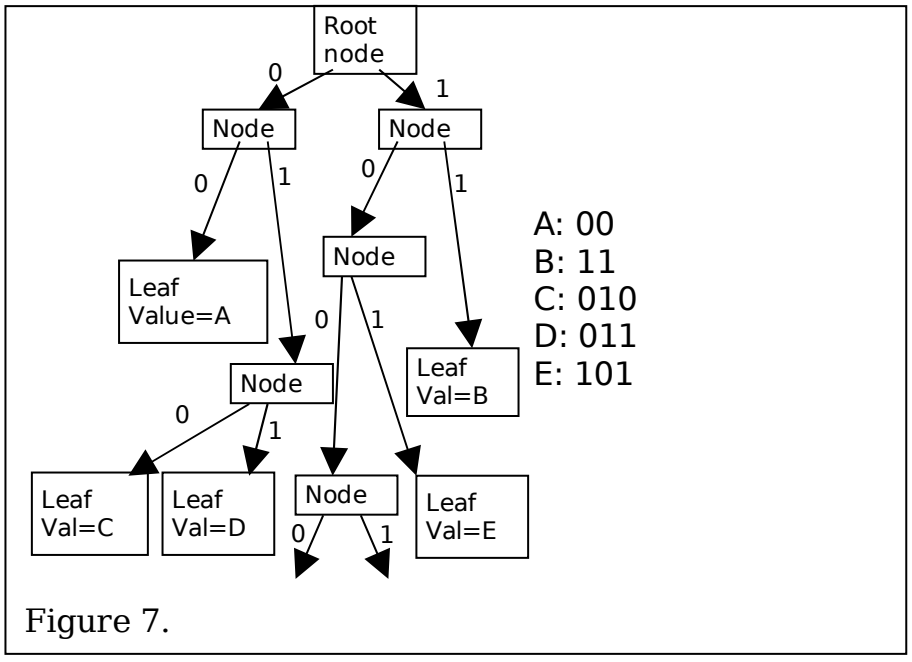


Figure 7.

There are two different Huffman tree generation algorithms, dynamic (adaptive) Huffman and static Huffman. GZIP uses adaptive Huffman to generate the tree based on pre-determined probabilities

for each character [1]. However, there have been no past attempts to integrate adaptive Huffman into hardware, and this has belonged to a software realm for a while. Because adaptive Huffman tree is generated by moving various nodes of the tree around the tree, and each node may have a variable number descendants, it's very hard, if possible at all to store a node in a fixed register in hardware. Storing value of a variable length in a register would involve resizing the register.

For the purposes of this final project we used static Huffman algorithm which means that a static binary tree is generated before decoding and a static Huffman table is generated before encoding. The table or the tree does not get altered in the process of decoding or encoding. This, of course, meant that we could no longer compare our output to that of the commercial gzip software.

Current Implementation of Huffman Alphabet:

Since we are using static Huffman coding, we generate Huffman table and tree in advance. In the case of encoder, we generate a Huffman table that maps ascii character to a Huffman codeword. Ascii characters are represented by indices of the registers in the register file, and their equivalent Huffman codewords – as 256-bit entries to these registers. Please see Figures 8 and 9 below.

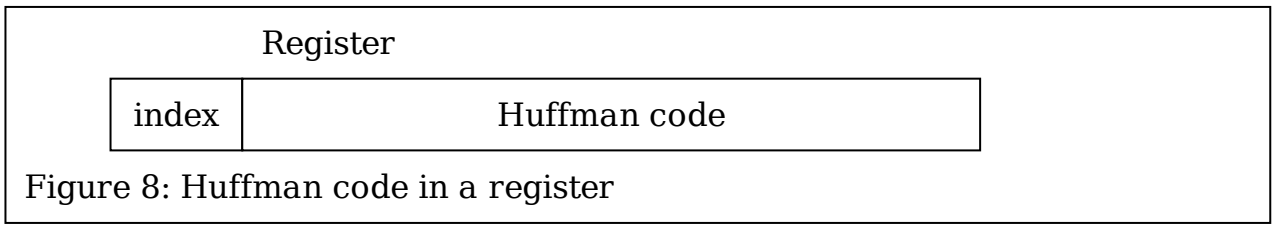


Figure 8: Huffman code in a register

Indices represent ascii characters. For example, letter 'a' if converted to decimal, would be equal to 97. The Equivalent 256-bit Huffman codeword for character 'a' would thus be placed in register 97. For

safety reasons we are using the 256-bit register to store the Huffman codeword for each ascii character, as it is the longest Huffman codeword ever possible for an 8-bit ascii character.

Registers

‘a’	97	1111111111111111.... 1111111111111110
‘b’	98	1111111111111111.... 11111111111111110
‘c’	99	1111111111111111.... 111111111111111110
‘d’	100	1111111111111111.... 1111111111111111110
‘e’	101	1111111111111111.... 11111111111111111110

Figure 9: Huffman table in a register file

Since Huffman code is a binary number of variable length, and registers in hardware cannot store elements of variable width, a different approach in storing Huffman code was taken. Instead of Huffman code in the register, we used a struct of both Huffman code and the bit length of that code. This way, a Huffman code can be stored in a 256-bit register (actually longer than 256 bits because struct information and the bit length must also be stored). See figure 10 for precise representation of Huffman code.

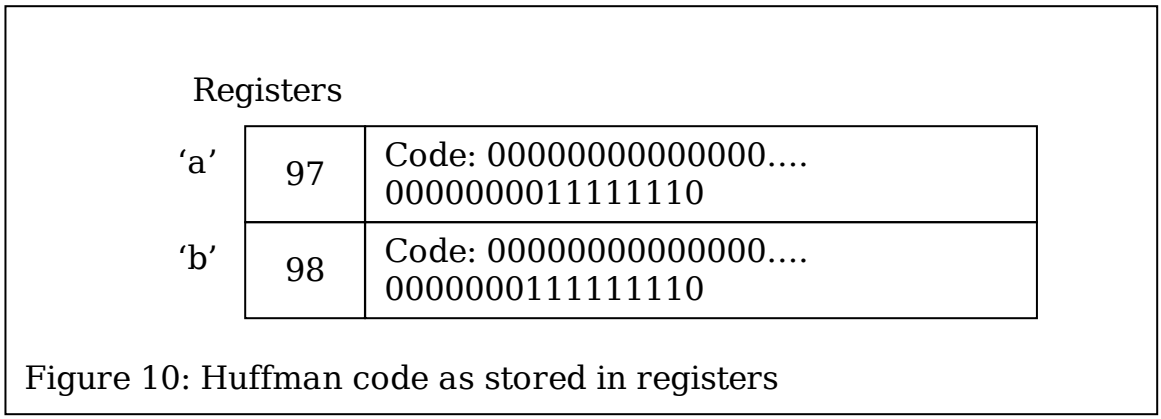


Figure 10: Huffman code as stored in registers

As you notice in Figure 9, the Huffman code is a sequence of 1's and '0's, but it is mostly 1's. This was decided because we are not sure what zero-terminated string means. It was assumed that any two zeros in a sequence would mean the zero-termination symbol. Therefore, each consecutive ascii character is one bit longer than the other, up to 256 bits (there are 256 ascii characters). This is a very inefficient Huffman tree, and can be improved by actually trying to build a huffman tree by hand. Since we are using a static Huffman tree in this project, we wrote an algorithm that generates the Huffman table and tree by giving certain ascii groups higher priority, and thus lower-length Huffman code. The order of importance was a-z, 0-9, A-Z, symbols, and other characters, with a-z receiving shortest Huffman codeword. There is a glitch in a register file in bluespec that for some reason does not write into an appropriate register. Whatever we pass to rfile.upd() ends up in a different register than specified In the input to upd() method.

In the case of decoder, instead of a Huffman table, we are using a Huffman tree. Since Huffman codes have no prefix and can be of a variable length, binary tree is the only way to determine which

character the code belongs to. The 256-bit implementation of Huffman code makes this binary tree a very long branch with tiny leaves coming out of this branch at every node. Every node of the branch has a connection to another node and a leaf. In current implementation, a leaf would mean zero, and node would mean one. Please see Figure 11 below.

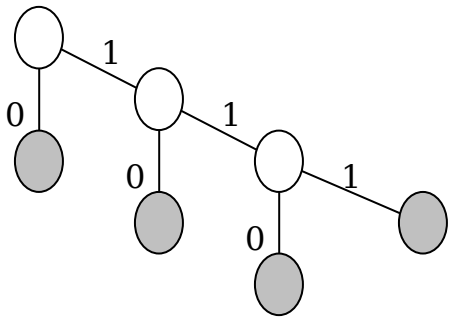


Figure 11: Nodes are white, and leaves are grey

As seen from figure above, every node has two branches, one pointing to a leaf, and another to next node in the long branch. Every Huffman code ends with a zero bit with the exception of the bottom branch. So, for example, if the stream of bits is 1111101101001111110, it means there are 5 characters (every character has the zero bit at the end). This is accomplished by iterating through the Huffman binary tree. As I said above, this algorithm is very inefficient, and can be greatly improved by hand-coding the most optimal static Huffman tree, which we have not had a chance to do due to time constraints.

Our inefficient implementation can be made efficient easily by modifying the code that builds Huffman table in the case of encoder and Huffman tree in the case of decoder. The Huffman tree creation

algorithm in case of decoder would have to be modified to perform a depth-first search. Currently it considers one of the two branches to always end up as leaf, and the other one to end up as either a leaf or a node.

Hardware Description:

Huffman coding uses a large register file (256 registers, each is 256 bits plus integer). Each entry in the Huffman table is a struct of variable-length Huffman code, and a length of that code. This is the way of keeping track of how many bits are in the Huffman code for a particular character.

For Huffman tree in case of decoding, we use a large register file (with 10-bit index). Each register entry contains a struct of 4 variables: type, value, left and right pointers. Type can only be a Node or a leaf. Value is an ascii character. We regard left pointer to always point to '0', and right pointer to always point to '1'. Unlike the table that encoder uses, decoder does not assign Huffman code to a register that is represented by character's ascii number. Instead it relies on pointers to keep track of position in the Huffman tree. Please see Figure 12 for detailed description.

1	Root Node (leftPointer=2, rightPointer=3)
2	Leaf, (Value='a')
3	Node (leftPointer=4, rightPointer=5, Value=1)
4	Leaf (Value='b')
5	Node (leftPointer=6, rightPointer=7, Value=1)
6	Leaf (Value='c')
7	Leaf (Value='d')

Figure 12: Huffman binary tree implementation

As you can see from the figure, the Huffman tree is a list of nodes, each node entry has 2 pointers, each of which contain the index of the node or leaf connected to that node. Node (represented by index 3) points to a leaf (register 4) and another node (register 5). This implementation only works for our scenario of long branch with leaves coming out of it (so that the left node is always a leaf and right node is either a node or a leaf). In case where the branches all have variable length, we would have to add the back pointer to keep track of parent nodes. For example, in register 5, there should be a back pointer to register 3 because there is a node in register 3 that points to a node in register 5.

General Overview of Our Huffman Algorithm

Encoder

After the Huffman table is generated by encoder, the encoder reads the input stream from the input FIFO (which is the output of LZ77 encoder), character by character, and places Huffman code bit by bit into an output FIFO. The output of LZ77 Encoder can be of one of the two types: an ASCII character, or a length-distance pair, so the input FIFO of Huffman encoder follows the same format. For the purpose of this project we are not encoding the pairs. We are simply passing the length-distance pairs to the output FIFO. Each element in output FIFO is either a bit or a length-distance pair. Each element in output FIFO is mostly one-bit wide because Huffman code is a stream of bits, and a FIFO cannot handle elements of variable bit length. Since ASCII alphabet is very large (128 standard characters, and 128 extended characters), having 128-256 choices for an element in the FIFO is not feasible. Since Huffman codes are of variable length, it would be very hard to utilize a FIFO with elements of variable length. Therefore, we update the FIFO one a bit at a time. The output FIFO would thus contain a stream of bits, one bit per element of a FIFO, and the contents of that FIFO would be saved to a file by test bench. The same output is connected to the input of the Huffman decoder.

Decoder

After the Huffman tree is generated by the decoder, the decoder reads the input stream from the input FIFO on a bit by bit basis. The input FIFO is connected to the output FIFO of Huffman encoder. Each element in the input FIFO is either a length-distance pair or a single bit. If it is a pair, then the pair is passed to the output FIFO. If it is a bit, then the Huffman tree is used to obtain the ascii character. For every bit in a sequence, the Huffman decoder goes down from a node to a child node based on the bit. It

takes a left pointer to a left child node (leaf), or a right pointer to the right child node. Once a leaf is hit, the character that is stored in that leaf as a value is placed in the output FIFO. Then test bench reads characters from the output FIFO and either saves into a text file, or passes the characters to LZ77 algorithm.

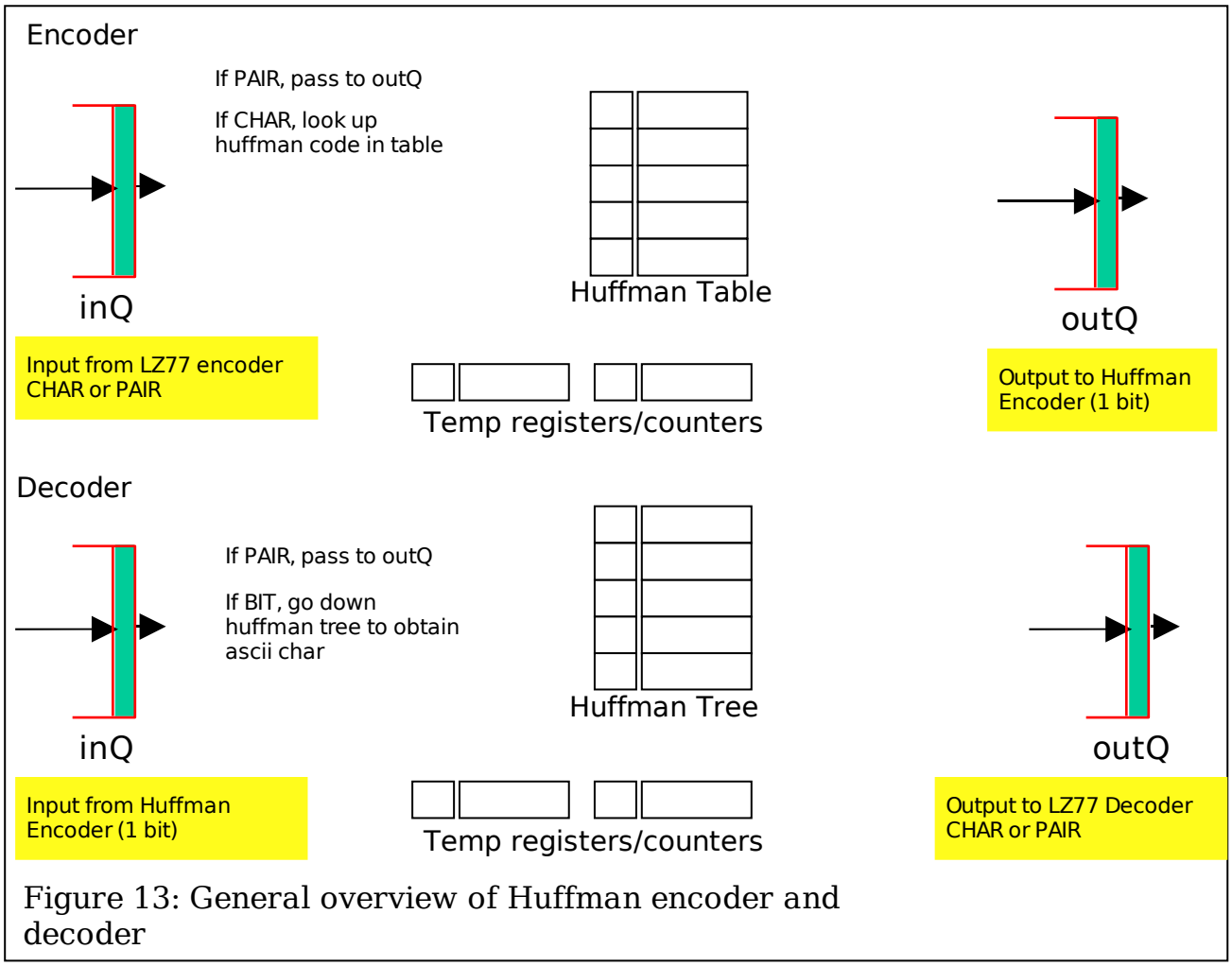


Figure 13: General overview of Huffman encoder and decoder

Detailed overview of Decoder and Encoder

Encoder

There are 5 main stages in the Huffman encoder pipeline: {HUFFMAN_INIT, HUFFMAN_GENERATE_TABLE, HUFFMAN_COMPRESS, HUFFMAN_GENERATE_BITSEQ,

and HUFFMAN_FINISH}. There is an optional HUFFMAN_TREE_CHECK stage for displaying the contents of the regfile. This was used to check if Huffman codewords were assigned to correct registers after the table has been generated (HUFFMAN_GENERATE_TABLE).

The Huffman table is generated during HUFFMAN_GENERATE_TABLE. HUFFMAN_COMPRESS and HUFFMAN_GENERATE_BITSEQ work together. The latter is called from inside the former. HUFFMAN_COMPRESS looks at one character at a time and saves the 256-bit code along with the bit length of the code in temporary registers. HUFFMAN_GENERATE_BITSEQ looks at those temporary registers and generates the bit sequence based on the length of codeword. This is the stage that inputs bits to the output FIFO bit by bit. Figure 13 shows how stages interoperate with each other.

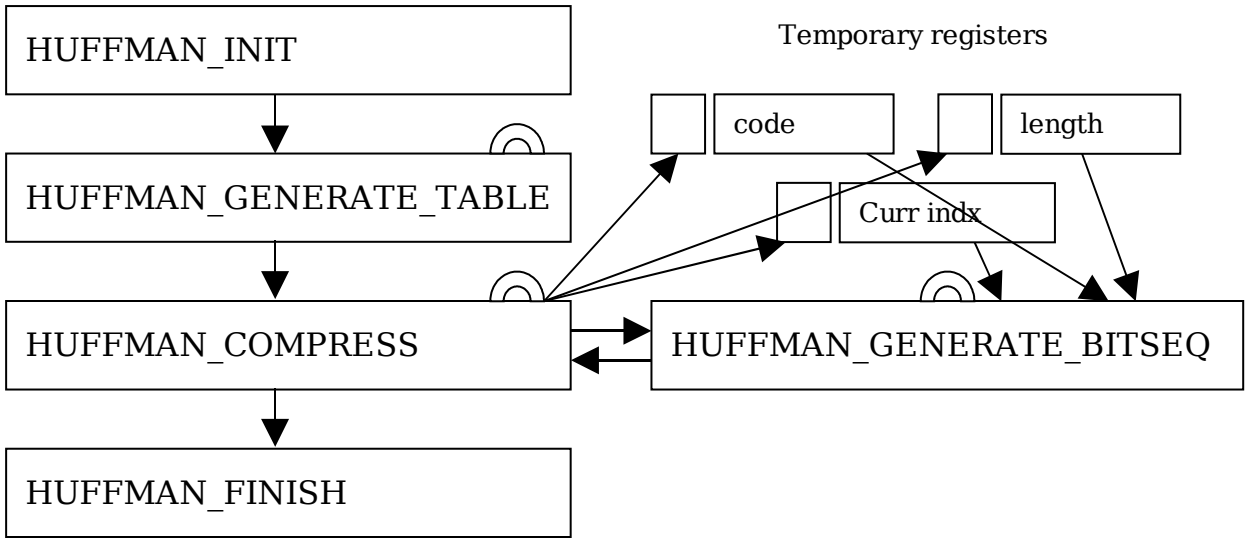


Figure 14: Stages of Huffman encoder

Decoder

There are 5 pipeline stages in Huffman decoder: {HUFFMAN_INIT, HUFFMAN_GENERATE_TREE, HUFFMAN_TABLEADD, HUFFMAN_DECODE, and HUFFMAN_FINISH}.

HUFFMAN_GENERATE_TREE creates 2 children nodes (a node can be of NODE type or LEAF type) and puts them in their corresponding temporary registers. The node structure is explained in Figure 15. HUFFMAN_TABLEADD is needed because there are two concurrent register file accesses for updating the Huffman tree. Each node entry requires a dedicated register file access, and since there are two nodes, two register file accesses are needed. They cannot run concurrently, so HUFFMAN_TABLEADD stage makes sure that the two accesses are executed sequentially.
Once the Huffman tree is generated, HUFFMAN_DECODE stage performs the top-down iteration to obtain the ascii character based on Huffman codeword. HUFFMAN_DECODE goes to the register specified by leftPtr (left pointer) if the bit is zero. If the bit is one, HUFFMAN_DECODE goes to register specified by rightPtr (right pointer). As soon as the leaf is reached, the value of that leaf is extracted and is used as an ASCII character output. Figure 15 shows how stages interoperate with each other.

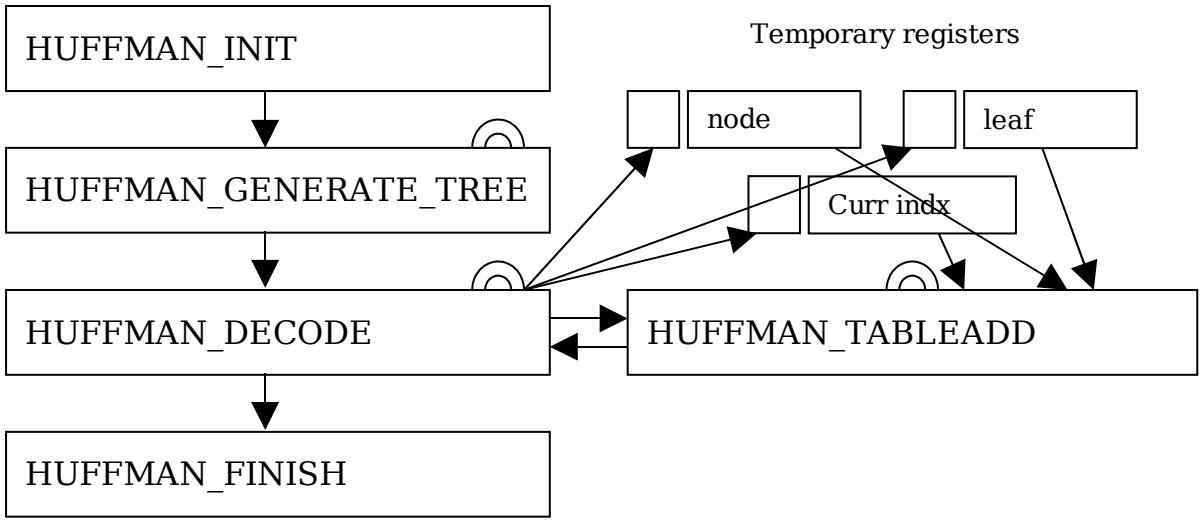


Figure 15: Stages of Huffman decoder

Huffman Node
- → value (ascii character)
- → type {NODE | LEAF}
- → leftPtr pointer to left Node (taken if bit is zero)
- → rightPtr pointer to right Node (taken if bit is one)

Figure 16: Node structure for Huffman tree

Overall Results:

We put the *Communist Manifesto*, a file consisting of 108,673 characters, into our LZ77 and Huffman encoders. The encoded output was 161,177 bytes long. Not only is this result vastly inferior to the default gzip software implementation that we used for testing which produced a 34.4 KB compressed

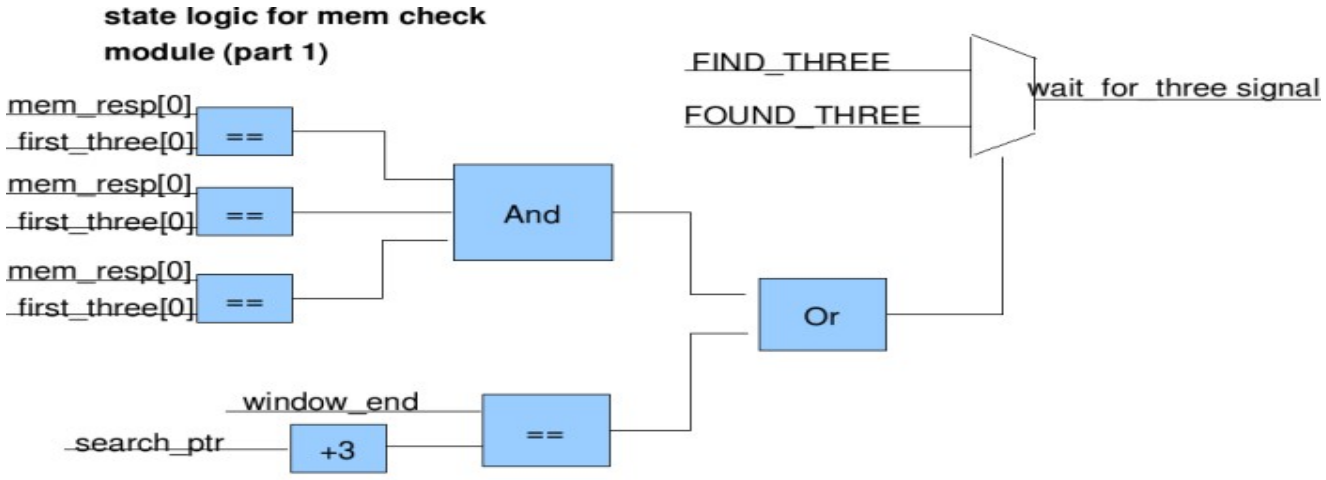
file, but it also actually expanded the file that we were working with.

We speculate that the expansion of these files is due to the naïve way in which we did our Huffman encoding. Specifically, we could not find the actual Huffman code values that gzip uses to do static Huffman encoding. Therefore our poor compression rate can probably be explained by the fact that our mapping between Huffman codes and characters was sub-optimal.

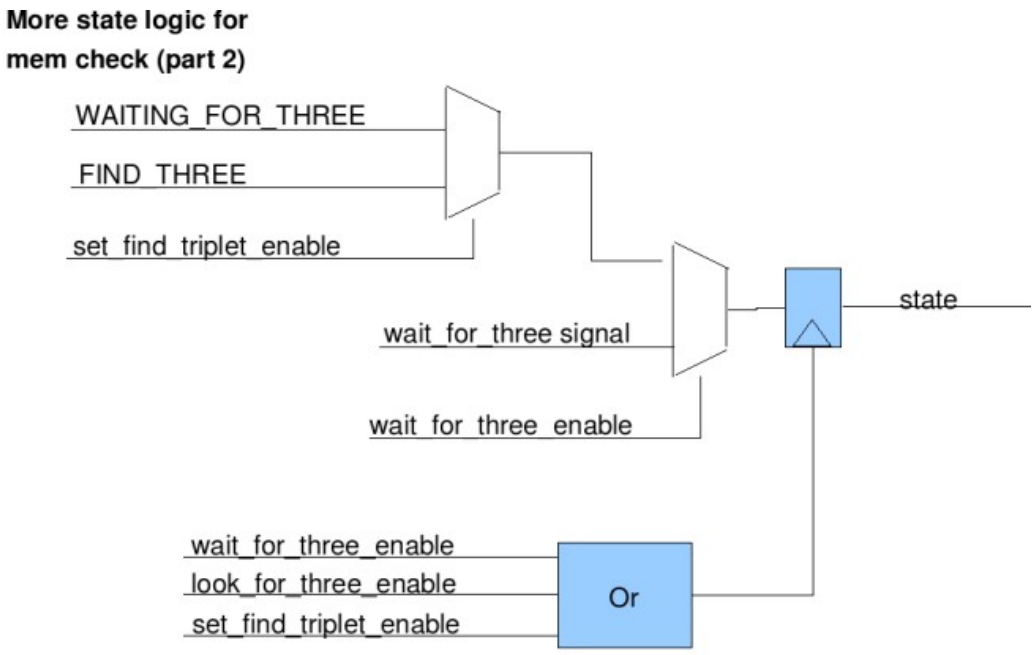
References

[1] http://www.cs.sfu.ca/CC/365/li/squeeze/AdaptiveHuff.html

Appendix A: State transition logic for Memory Checker module

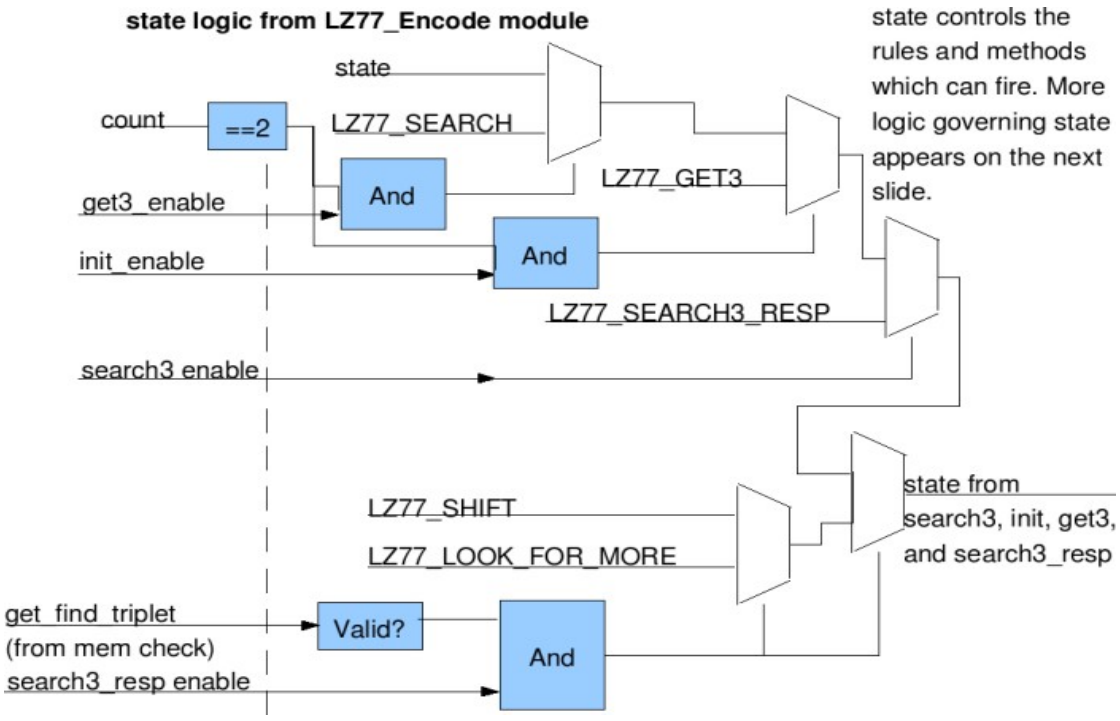


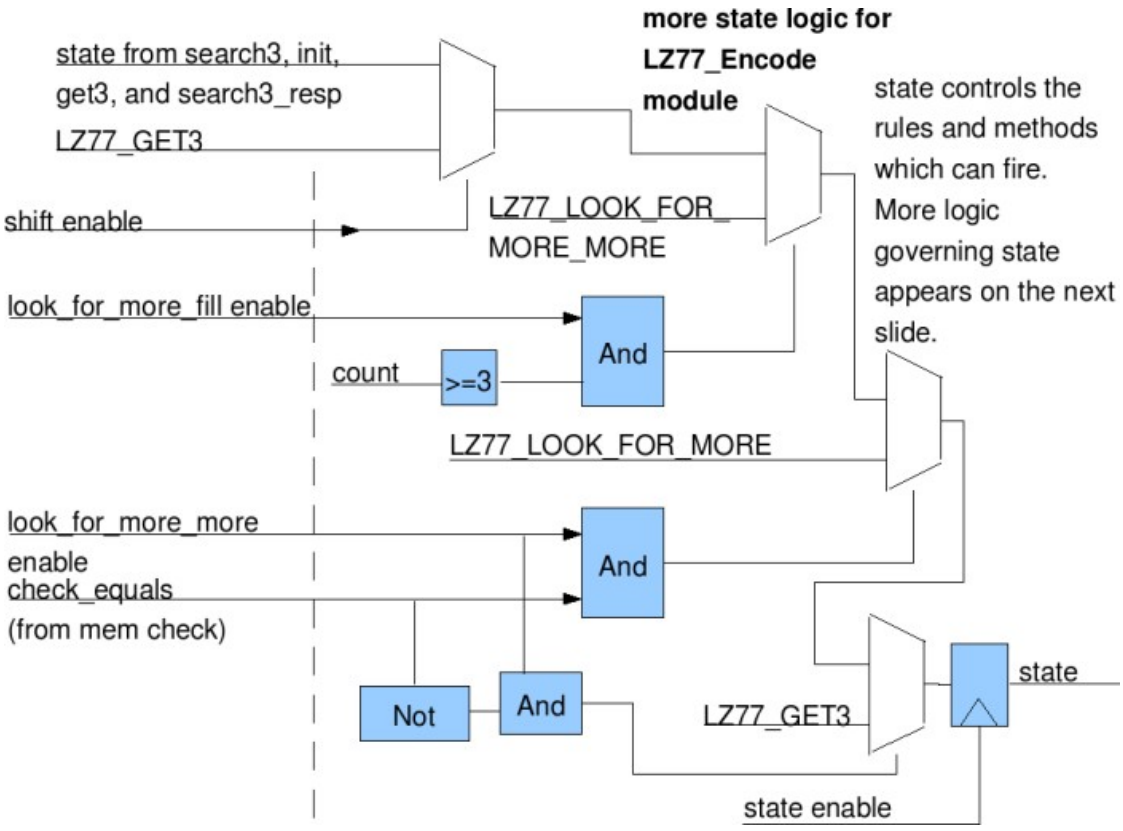
We use a state element titled "state" to determine which rules and methods are allowed to fire. Above is the logic governing the state logic created in the wait_for_three rule. The output of this logic will be used in the next slide to determine the overall state logic.



This logic governs the state of the mem check module. Note, state can only be changed when the wait_for_three, look_for_three, and set_find_triplet methods/rules are enabled.

Appendix B: State Logic for LZ77_Encode Module




more state logic for
LZ77_Encode
module
state from search3, init,
get3, and search3_resp
LZ77_GET3
shift enable
LZ77_LOOK_FOR_
MORE_MORE
state controls the
rules and methods
which can fire.
More logic
governing state
appears on the next
slide.
look_for_more_fill enable
count
>=3
And
LZ77_LOOK_FOR_MORE
look_for_more_more
enable
check_equals
(from mem check)
And
Not
And
LZ77_GET3
state
state enable

more state logic for LZ77_Encode module

state controls the rules and methods which can fire. This is the logic for state enable.