Problem 0: Follow the instructions for class account setup and for homework workflow in `https://www.ece.lsu.edu/ee4720/proc.html`. Review the comments in `hw02.s` and look for the area labeled "Problem 1."

Those who want to start before getting to the lab can find the assembler for the entire assignment at `https://www.ece.lsu.edu/ee4720/2021/hw02.s.html`. For MIPS references see the course references page,
`https://www.ece.lsu.edu/ee4720/reference.html`. Easy MIPS practice problems can be found in the practice directory, see MIPS Homework and Practice Workflow in
`https://www.ece.lsu.edu/ee4720/proc.html`.

This Assignment

One goal of this assignment is to build assembly language proficiency by working with data at different sizes and by traversing a tree. The sizes are bits for the compressed text, words for the array of compressed text, half (2-byte) for the tree, and bytes for the dictionary. Another goal is to provide a starting point for architectural improvements. That is, ISA and hardware changes to make code go faster.

Huffman Compression Background

One way to compress data is to divide it up into pieces, compute a *Huffman* coding for the pieces, then replace each piece with its Huffman code. The size of a Huffman code can vary from 1 bit (yes, just one), to an arbitrarily long bit vector. Pieces that appear more frequently in the original text will have short codes and pieces that appear less frequently will have longer codes. Consider a file containing English text, such as the source file for the Homework 1 handout. One way of dividing it to pieces is to make each character a piece. For Homework 1 a space was the most frequent piece (258 times) followed by the letter "e" (174 times). They received codes 100_2 and 1110_2, each of which is shorter than the eight bits used to encode each in the original file. The character "8" appears just once and gets a long encoding, $110\,1100\,0001_2$. The compressed data consists of a concatenation of all of the codes. So "e e" would be encoded $111\,0100\,1110_2$. The encoded data does not contain any separators between the pieces. To decode it one needs to first assume the code is one bit long, see if such a code exists, if not try two bits, and so on. So for the example one would first look for a code for 1_2. If it didn't exist (and it shouldn't) one checks for 11_2, which also shouldn't exist, neither does 111_2 (the third try). One the fourth try we look for 1110_2 and find that this is a code and the value is "e". The de-coding can continue by trying 1_2, 10_2, and finally 100_2 which is the code for a space.

Huffman *Huff Tree* Format for This Assignment

This assignment will use a format in which text is compressed into three arrays, the compressed text, starting at `huff_compressed_text_start`, a dictionary of strings, starting at `huff_dictionary`, and the Huff Tree (a lookup tree), at `huff_tree`.

The compressed text is a long bit vector. As with Homework 1, bits are numbered in big-endian order. The compressed text is specified using words but of course can be read using other sizes. The dictionary of strings consists of a bunch of null-terminated strings. The Huff Tree is used to decode compressed pieces. It is traversed using bits of the compressed text (`0` for left child, `1` for right child) and a leaf provides either an index into the dictionary or a character.

Consider the following excerpt from the homework file:

```
huff_compressed_text_start:
        .word 0xd9ac96d8, 0x10b75d4f, 0xa06510d1, 0x7d9961e3, 0xeb6f31f1
```

```
# Encoding: .word BIT_START, BIT_END, TREE_POS, DICT_POS, FRAG_LENGTH
huff_debug_samples:
#   0: 0  11011 -> "\n"
        .word 0, 5, 0x2ee, 0xa, 1;
#   0: 5  001101 -> "    ."
        .word 5, 11, 0x32, 0x16, 9;
#   0:11  0110010010 -> "text"
        .word 11, 21, 0x110, 0x27b, 4;
#   0:21  11011 -> "\n"
        .word 21, 26, 0x2ee, 0xa, 1;
#   0:26  011000000 -> "histo"
        .word 26, 35, 0xea, 0xce, 5;
#   1: 3  1000010 -> ":\n"
        .word 35, 42, 0x1d6, 0x2e, 2;
```

The compressed text is shown as 32-bit words, under `huff_compressed_text_start` and as an aid in debugging, the start of the same text is shown under `huff_debug_samples`. The first piece, 11011_2, encodes a line feed character (we can see that by looking at the comment). The second piece, 001101_2, encodes " ." (spaces followed by a period). The first piece is in bits 0 to 4 (inclusive) of the compressed text, and the second piece is in bits 5 to 10. The hexadecimal digits of the compressed text can be found by concatenating the compressed pieces and then grouping them into four-bit hex digits: `11011 001101 0110010010` → `1101 1001 1010 1100 1001 0` → `d 9 a c 9 ?`. That matches the start of the compressed text shown under `huff_compressed_text_start`.

For this assignment a piece, for example `11011`, is decoded by traversing the `huff_tree`. Each node in the `huff_tree` is 16 bits and can be one of three possible kinds: A leaf encoding a character, a leaf encoding a dictionary entry, or an internal node (with a left and right child). If the value of a node is `<128` it is a leaf encoding a character. Otherwise if the value of a node is `>=0x7000` it is a leaf encoding a dictionary entry. Otherwise it is an internal node.

```
huff_tree:
# Huffman Lookup Tree
#
huff_tree:  # Note: Most entries omitted.
        .half 0x01fa  # Tree Idx   0               Pointer to right child.
        .half 0x011d  # Tree Idx   1  0            Pointer to right child.
# [Many entries not shown.]
        .half 0x028c  # Tree Idx 378  1            Pointer to right child
# [Many entries not shown.]
        .half 0x02e2  # Tree Idx 524  11           Pointer to right child.
        .half 0x02c7  # Tree Idx 525  110          Pointer to right child.
# [Many entries not shown.]
        .half 0x02e1  # Tree Idx 583  1101         Pointer to right child.
# [Many entries not shown.]
        .half 0x000a  # Tree Idx 609  11011        Literal "\n"
```

The Huff Tree is an array of nodes, each a 16-bit value. Let T denote such an array. The root is $T[0]$. Let i indicate some position in the tree and $n = T[i]$ denote the node at position i. The assembler data above shows some elements of a Huff Tree. (The entire tree can be found in `hw02.s`.) The numbers in binary (following the Tree Idx) show the path to that node.

If $n < 128$ it is a leaf node encoding a character, and the ASCII value is n. If $n \geq 7000_{16}$ then the node is a leaf encoding a dictionary entry. The address of the first character of the dictionary entry is `huff_dictionary + n - 0x7000`. The strings in the dictionary are null-terminated.

Let $n = T[i]$ be a non-leaf node, so that $n \geq 128$ and $n < 7000_{16}$. Its left child is at $T[i+1]$ and its right child is at $T[n-128]$.

Here is how piece `11011` of the compressed text would be decoded based on the data in the example above. Start at the root, retrieving $T[0]$. The value is $1fa_{16}$, which is an internal node. The first bit of `11011` is 1 so we traverse the right child which is at $1fa_{16} - 80_{16} = 17a_{16} = 378$. The entry at tree index 378 (based on the table) is $28c_{16}$ which again is an internal node. The second bit of the piece is 1 so we compute the index of the right child: $28c_{16} - 80_{16} = 20c_{16} = 524$. The next compressed bit is zero so we proceed to the left child, at index $524 + 1$. The tree excerpt above includes the entry leading to the leaf node.

The routine below (which can be found in `huff-decode.cc` in the homework package) decodes the piece starting at bit `bit_offset` and writes the decoded piece at `dcd_ptr`.

```
void
hdecode(HData& hd, int& bit_offset, char*& dcd_ptr)
{
  // Decode one piece, starting at bit position bit_offset and
  // write decoded piece starting at dcd_ptr.

  // hd.huff_compressed: Compressed text. An array of 32-bit values.
  // hd.huff_tree: A tree used to decode the compressed pieces.
  // hd.huff_dictionary: Decompressed pieces.

  // Start lookup at root of Huffman tree (tree_idx = 0).
  //
  int tree_idx = 0;

  while ( true )
    {
      // Retrieve node.
      uint16_t node = hd.huff_tree[tree_idx];

      if ( node < 128 )
        {
          // Node is a leaf encoding a character.

          char c = node;  // Node value is an ASCII character.
          *dcd_ptr++ = c; // Write character to decoded text pointer ..
          return;         // .. and return.
        }
      else if ( node >= 0x7000 )
        {
          // Node is a leaf holding an index into the dictionary.

          // Compute dictionary index.
          int idx = node - 0x7000;
```

```
            // Compute address of first character of dictionary entry.
            char* str = hd.huff_dictionary + idx;

            // Copy the dictionary entry.
            while ( *str ) *dcd_ptr++ = *str++;
            return;
          }
        else
          {
            // Node is not a leaf, need to set tree_idx to the index of
            // either the left or right child of the node. The left
            // child is used if the next bit of compressed text is zero
            // and the right child is used if the next bit of compressed
            // text is 1.

            // Get the next bit of compressed text.
            //
            int comp_idx = bit_offset / 32;  // Index of word in huff_compressed.
            int bit_idx = bit_offset % 32;  // Index of bit. MSB is 0.

            uint32_t comp_word = hd.huff_compressed[ comp_idx ];

            // Move needed bit to LSB in a way that sets other bits to zero.
            bool bit = comp_word << bit_idx >> 31;

            bit_offset++;

            if ( bit )
              {
                // Set tree_idx to index of the right child.
                tree_idx = node - 128;
              }
            else
              {
                // Set tree_idx to index of the left child.
                tree_idx++;
              }
          }
      }
}
```

Homework Package

The homework package consists of files to help with your solution and to satisfy curiosity. Your solution, of course, goes in `hw02.s`, which is in the usual SPIM assembler format for this class.

The Huffman compression was performed by the `huff` perlscript. To compress MYFILE invoke it using `./huff MYFILE`. With no arguments it compresses itself. It will write two files, `encoded.s` and `encoded.h`. The contents of `encoded.s` could be copied into the `hw02.s` (replacing what's there). Do this if you'd like to run your code on some other input.

File `huff-decode.cc` is a C++ routine that includes `encoded.h` and decodes it. It needs to

be re-built for each new input file. (Sorry, I've already spent too much time on the assignment.) Here is how it might be used on a new file:

```
[koppel@dmk-laptop hw02]$ ./huff ../../hw02.tex
File ../../hw02.tex
Words 366  Codes 366  Resorts 11
[koppel@dmk-laptop hw02]$ gmake -j 4
g++ --std=c++17 -Wall -g huff-decode.cc -o huff-decode
[koppel@dmk-laptop hw02]$ ./huff-decode
Decoded:
\magnification 1095
% TeXize-on
\input r/notes
```

The assignment was created by compressing `histo-bare.s`.

Problem 1: Complete routine `hdecode` so that it decodes the piece of Huffman-compressed text starting at bit number `a1`, writes the decoded text to memory starting at the address in `a2`, and sets registers `v0`, `v1`, `a1`, and `a2` as described below. (Yes, `a0` is unused.)

Use symbol `huff_compressed_text_start` for the address of the start of the compressed text, `huff_tree` for the address of the start of the Huff Tree, and `huff_dictionary` for the address of the start of the dictionary.

When `hdecode` returns set `v0`, `v1`, `a1`, and `a2` as follows. Set `a1` to the next bit position to use. For example, if the compressed piece were 3 bits and `hdecode` were called with `a1=100` then when `hdecode` returns `a1` should be set to `103`. Set `a2` to the address at which to write the next decoded character. For example, if the decoded text is 9 characters (not including the null) and initially `a2=0x1000` then when `hdecode` returns `a2` should be set to `0x1009`. When `hdecode` returns `v0` should be set to the address of the leaf in the Huff Tree that was used and `v1` should be set to either the address of the dictionary entry used or the value of the character.

Note that the return values of `a1` and `a2` are useful because they are at the values needed to call `hdecode` again for the next piece. The return values of `v0` and `v1` are for debugging.

When `hw02.s` is run `hdecode` will be called multiple times, the return values checked, and the results printed on the console. It will be called for the first 200 pieces, or until there are three errors, whichever is sooner. A tally of errors is printed at the end, followed by the decoded text.

Pay attention to the error messages. Once syntax and execution errors are fixed, debug your code by tracing. To trace start the simulator using `Ctrl`-`F9` if running graphically or just `F9` non-graphically. At the spim prompt type `step 50` to execute the next 50 instructions. The trace shows line numbers of source assembly to the right of the semicolon. It also shows changed register values.

Single stepping is most useful when the first piece fails, which is likely to happen at first. But before long it will be correct and so viewing the trace will be a pain. To have the testbench start at your erroneous piece first locate the piece after label `huff_debug_samples`. The first number after `.word` is the Bit Position referred to in the "Decoding of.." message. Copy that line (perhaps with the comment above it) to just below the label `huff_debug_samples`.