LSU EE 4720 **Homework 1** **Due: 24 January 2019**

Problem 1: Follow the instructions for class account setup and for homework workflow in `https://www.ece.lsu.edu/ee4720/proc.html`. Review the comments in `hw01.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/ee4720/2019/hw01.s.html` and the C++11 code at `https://www.ece.lsu.edu/ee4720/2019/hw01-equiv.cc.html`. A good reference for C++ is `https://en.cppreference.com/`. 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`.

File `hw01.s` in the assignment package is where your solution should go. Briefly, complete routine `get_index` so that it finds the position of a word in a list. It will use a hash table to speed things up if the same word is used twice. (Yes, you're going to code all that in assembler!) File `hw01-equiv.cc` contains C++11 code that does the same thing. In particular, routine `App::lookup` does much of what `get_index` is supposed to do.

Routine `get_index` is called with four arguments. The first, `a0`, is the address of a C-style string, call this string the *lookup word*. Argument `a1` is the address of a table of C-style strings, call this the *word list*. (See `word_list_start` in the code.) Argument `a2` is the address of the end of the word list, and `a3` holds the address of a 1024-byte area of memory called the *hash table*.

The routine is supposed to check if the lookup word is in the word list. When the routine returns register `v0` should be set to `0` if the lookup word was not found in the word list, otherwise `v0` should be set to the position in the word list. The first word (`aardvark`) is at position 1, the second is at position 2, etc. The word list contains one string after another. That is, the character after the null terminating `aardvark` is `a`, the first letter of the second word, `ark`. The word list layout used in the C++ code, `App::lookup`, is the same as the layout used in assembler, so inspecting the code in `App::lookup` might help in understanding the layout of the word list.

(*a*) Add code to `get_index` so that it sets `v0` as described above. The testbench should show zero Pos errors.

(*b*) Complete `get_index` so that it uses a hash table to speed lookups. Suppose that the number of words in the word list is large (even if it's small for this assignment), and also do not assume that the list is alphabetized. *Note: That was the original plan.*

Let n denote the number of bytes in the word list. The value of n is `a2-a1` when `get_index` starts and always is `word_list_end - word_list_start`. The amount of work needed to determine whether the lookup word is in the word list is $O(n)$. Remember, we don't like $O(n)$, we much prefer $O(\log n)$ or better yet $O(1)$. As the alert students suspect, this is where the hash table comes in.

When first called, `get_index` should check whether there is an entry in the hash table for the lookup word. It should do so by computing a hash of the first three characters of the lookup word using the same hash function as `App::hash_func` (shown below). Note that `App::hash_func` returns a value in the range 0-255, call that the *hash index*.

```
int hash_func(const char* str)
  {
    return
      ( str[0] ^ ( str[1] >> 2 ) ^ ( str[1] << 6 ) ^
      ( str[2] << 4 ) ^ ( str[2] >> 4 ) ) & hash_mask;
```

```
}
```

Each entry in the hash table consists of two 16-bit values, see the `Hash_Elt` structure. The first 16 bits is an offset, the second is the pos.

```
struct Hash_Elt {
  int16_t offset;
  int16_t pos;
};
```

The `pos` member indicates the position of the word in the word list, or 0 if that hash entry has not been initialized. Member `offset` is the character offset in the word list. That is, `a1 + offset` is the address of the first character of the word.

The C++11 code below first computes the hash for the lookup word (`word`), retrieves the hash table entry, then checks to see if the entry is a hit. To be a hit the `pos` member must be non-zero and the word at offset must match `word`. (This is necessary because two different words can generate the same hash index.)

```
Lookup_Info lookup(const char *word)
{
  // Look up word in hash table.
  //
  const int hash = hash_func(word);
  Hash_Elt& elt = hash_table[ hash ];

  Lookup_Info li;
  li.hash = hash;

  // Return if hash table hit.
  //
  if ( elt.pos && streq( &words[elt.offset], word ) )
    {
      li.pos = elt.pos;
      li.in_hash = true;
      return li;
    }
```

The `li` object has information about the match. In the assembler code the same information is provided in `v0` and `v1`.

If there is a miss in the hash table, scan for the word. If it is in the word list then update the hash table so that the next time there will be a hash table hit. That is performed by the `App::lookup` by writing the hash table element reference, `elt`, that it had retrieved.

```
  // Scan word list to find word.
  //
  int offset = 0, pos = 0;
  while ( words[offset] )
    {
      if ( streq( &words[offset], word ) )
        {
```

```
        // Word has been found, add to hash table and return index.

        elt.offset = offset;
        li.pos = elt.pos = pos;
        return li;
      }

    // Word not found, advance to next word. Assembler can be faster.
    offset += strlen( &words[offset] ) + 1;
    pos++;
  }

// Word not in word list, return 0.
li.pos = -1;
return li;
```

When the the code in `hw01.s` is run a testbench will call `get_index` for multiple words. The console will show the position and hash returned for the word, an `X` to the right of the position indicates that it is wrong, and underscore indicates that it is correct. Two characters are shown to the left of the hash index (labeled `Hash`). The first character, `X` or `_`, indicates whether the hash table lookup result is correct, the second character indicates whether the hash index itself is correct. At the end there will be a tally of the total number of errors.