1 Pointers and Structures

So far we have talked about arrays, which are a collection of objects of the same type stored in a contiguous chunk of memory. A structure is also a collection of objects stored in a contiguous chunk of memory, but these objects are not necessarily all of the same type. Consider the following structure in C:

```
typedef struct node_s {
  int * data;   // pointer to an integer
  struct node_s * next;  // pointer to another struct node_s
} node_t;   // node_t is shorthand for "struct node_s"
```

The above structure can be used to implement a singly-linked list of nodes (where each node has type node t) as follows: the head node is the first node of the list, its next field contains a pointer to the next node in the list, etc. The list is terminated with a pointer to NULL. Recall from lecture that pointers are just memory addresses at the machine level. Below is C code for a function that increments each element of a singly-linked list by a given value. The variable head contains a pointer to the first element of the list.

```
void increment(node_t * head, int value)
{
  for(node_t * trav = head; trav != NULL; trav = trav->next)
  {
      *(trav->data) += value;
  }
}
```

Write increment as a MIPS funtcion.

2 Endian-ness

There are two possible ways to store multi-byte data words (*e.g.*, 32-bit integers) in memory; in both cases when a word is stored to address N, the bytes are stored in byte addresses N, N+1, N+2, and N+3. *Little endian* stores the least significant byte (LSB) in the first byte (N, *i.e.*, little end first), while *big endian* stores the most significant byte (MSB) in the first byte (N, *i.e.*, big end first). For example, if storing the value `0xdeadbeef` to address `0x10000000`, you get the following:
Both conventions are in active use by popular microprocessors: Alpha and x86 are little endian,

ADDRESS:	**0x10000000**	**0x10000001**	**0x10000002**	**0x10000003**
Little Endian	`0xEF`	`0xBE`	`0xAD`	`0xDE`
Big Endian	`0xDE`	`0xAD`	`0xBE`	`0xEF`

SPARC and PowerPC are big endian. MIPS is actually bi-endian (*i.e.*, it can be run in either mode) and SPIM (somewhat sadistically) uses the endianness of its host. That is, SPIM is big endian on a SPARC machine and little endian on an x86.
As long as you don't cast one data type to another, you shouldn't have to worry about whether you are running on a big-endian or little-endian machine. That is unless you need to communicate over the network.
When two computers communicate, they must ensure that numbers are correctly interpreted even if one machine is little endian and the other is big endian. For example, remote procedure call (RPC) defines big endian as the canonical form (not surprising since Sun defined it and SPARC is big endian). Thus, little endian machines need to "swizzle" their bytes to big endian before sending them across the network.

```
unsigned swizzle_word(unsigned in) {
   unsigned temp = in >> 24;
   temp |= ((in >> 16) & 0xff) << 8;
   temp |= ((in >> 8) & 0xff) << 16;
   temp |= (in & 0xff) << 24;
   return temp;
}
```

For your reference, here is the same function in MIPS assembly:

```
swizzle:
       srl   $t0, $a0, 24      # v0 = (in >> 24);
       and   $v0, $t0, 0xff

       srl   $t0, $a0, 16      # v0 |= ((in >> 16) & 0xff) << 8;
       and   $t0, $t0, 0xff
       sll   $t0, $t0, 8
       or    $v0, $v0, $t0

       srl   $t0, $a0, 8       # v0 |= ((in >> 8) & 0xff) << 16;
       and   $t0, $t0, 0xff
       sll   $t0, $t0, 16
       or    $v0, $v0, $t0
       and    $t0, $a0, 0xff    # v0 |= (in & 0xff) << 24;
       sll    $t0, $t0, 24
       or     $v0, $v0, $t0

       jr     $ra
```

2.1 Endian Detection

Write a MIPS function that returns the value 1 on a big endian machine and 0 on a little endian machine.

2.2 memset

In lecture, we discussed pointers and how they are just memory addresses at the machine level. Below is C code for a function that writes an integer value repeatedly `num_words` times starting from address `dst`.

```
void memset(int *dst, int value, int num_words) {
   for (int i = 0 ; i < num_words ; ++ i) {
     *dst = value;
      dst ++;
   }
}
```

Write it as a MIPS function: