Arrays

Suppose we wish to declare and use an array of `N` integers. How would we represent this array in MIPS? Let the array be `a[ ]`. Let the address of the first element of the array (written `&a[0]` in C) be kept in one of the registers, say `$s0`. This address is called the *base address* of the array. To access an element of array, we need to specify the address of that element relative to the base address, i.e. the *offset*.

For example, consider the C instructions:

```
a[12] = a[10];
```

In MIPS, we cannot transfer data directly from one location in Memory to another. Instead, we need to pass the data through a register. Here might be the corresponding MIPS instructions:

```
lw      $s1, 40($s0)

sw      $s1, 48($s0)
```

The offsets from the base address are 40 and 48 for a[10] and a[12] respectively. Why? We need four bytes for each word, so an offset of 10 words is an offset of 40 bytes and an offset of 12 words is an offset of 48 bytes.

Another example is the C instruction,

```
m = a[i];
```

Let integer variable `m` be in register `$s1` and integer index `i` be in register `$s2`. We need to multiply the contents of `$s2` by 4 and then add the result to the contents of `$s0`. Note we multiply `i` by 4 by using the `sll`, which is simpler than having to perform a general multiplication. (We will see how to perform multiplication in an upcoming lecture.)

```
sll   $t0, $s2, 2
add   $t0, $s0, $t0
lw    $s1, 0($t0)
```

Strings

Recall from C programming that each character in a text file is stored in one byte.[1] In C, a variable of type "char" uses one byte. Such text is often represented is using ASCII. (For the table of ASCII characters, see `http://www.asciitable.com`.)

Previously, we saw the instructions `lw` and `sw` which take a load from or store a word to Memory. There are a similar instructions for single bytes. `lb` loads a byte, and `sb` stores a byte. These functions are of I-format.

```
lb $s2, 3($s1)
sb $s2, -2($s1)
```

[1]In Java, each `char` is stored using two bytes, namely UNICODE.

`lb` takes one byte from memory and puts it into a register. The upper 24 bits of the register are sign extended i.e. filled with whatever value is in the most significant bit (MSB) of that byte. There is also an unsigned version of load byte, namely `lbu`, which fills the upper 24 bits with 0's (regardless of the MSB).

`sb` copies the lower 8 bits of a register into some byte in memory. This instruction ignores the upper 24 bits of the word in the register.

Example: How to calculate the length of a character string?

Consider the following C instructions that computes the length of a string. In C, a string is a sequence of bytes, terminated by the ASCII NULL character which in C is denoted '\0' and which has byte value 0x00.

```
/*  Some C code to compute the length of a string */

       char  *str, *p;   //  Declare pointers to strings
                         //  These are 32 bit numbers (addresses).
       int  ct;
          :
       str = "Cary Price";
       ct = 0;
       p = str;                //    ...OR, AS SUGGESTED IN CLASS,
       while (*p != '\0'){     //    DON'T USE p AND INSTEAD JUST USE:
           p++;                //
           ct++;               //   while ( *(str + ct) != '\0' ){
       }                       //      ct++;}
```

Notice that `p` is a number (a 32 bit address), so when we add 1 to it, we are incrementing the address by one byte.

Here's is MIPS code that does the same thing.

```
        addi   $s2, $zero, 0        # $s2  is ct
        addi   $s1, $s0, 0          # $s0  is str, $s1 is p
loop:   lb     $t0, 0($s1)
        beq    $t0, $zero, exit     # branch if  *p ==  '\0'
        addi   $s1, $s1, 1          # increment p
        addi   $s2, $s2, 1          # increment ct
        j      loop
exit:
```

Note: `$t0` holds the ASCII code (a byte) in the lower 8 bytes.

Assembler directives

We have seen many MIPS instructions. We now would like to write entire MIPS programs. To define a program, we need certain special instructions that specify where the program begins, etc. We also need instructions for declaring variables and initializing these variables. Such instructions are called *assembler directives*, since they "direct" the assembler.

Let's look at an example. Recall the MIPS code that computes the length of a string. We said that the string was stored somewhere in Memory and that the address of the string was contained in some register. But we didn't say how. Below is program that defines a string in MIPS Memory and that counts the number of characters in the string. You can download the code from
`http://www.cim.mcgill.ca/~langer/273/strlength.asm`

The program begins with several assembler directives (`.data, .asciiz`, ...) followed by a main program for computing string length. The `.data` directive says that you are about to declare certain data that should be put into the data segment of Memory. The label `str` can be used in the program whenever you want to reference the data element that is defined at that line. In particular, the line has an `.asciiz` directive which declares a string, "Cary Price". When the program below uses the label `str`, the assembler translates this label into a bit representation of the address. This can be either an offset value as in the case of a conditional branch, or it can be an absolute address as in the case of an unconditional branch (jump) instruction. After running the program, look at the data segment of memory and you will indeed see the string there.

The `.text` directive says that the instructions that follow should be put in the instruction ("text") segment of memory. The `.align 2` directive puts the instruction at an address whose lower 2 bits are 00. The `.globl main` directive declares the instructions that follow are your program. The first line of your program is labelled `main`.

```
             .data                    #  Tells assembler to put the following
                                      #  ABOVE the static data part of Memory.
                                      #  (see remark at bottom of this page)
str:         .asciiz "Cary Price"     #  Terminates string with NULL character.
             .text                    #  Tells the assembler to put following
                                      # in the instruction segment of Memory.
             .align 2                 #  Word align (2^2)
             .globl main              #  Assembler expects program to have
                                      #  a "main" label, where program starts

main:        la     $s0, str          # pseudoinstruction (load address)
        #    (INSERT HERE THE MIPS CODE ON PREVIOUS PAGE)
```

The address of the string is loaded into register using the pseudoinstruction `la` which stands for *load address*. SPIM translates this instruction into

```
            lui  $s0,  4097
```

Note that $(4097)_{10} = (1001)_{16}$, and so the address is `0x1001000` which is the beginning address of the user data segment. (Actually there is a small "static" part of the data segment below this which starts at address `0x10000000`. You can see this in SPIM.)

Here are some more assembler directives that you may need to use:

- `.word` assigns 32-bit values
- `.byte` initializes single bytes
- `.space` reserves a number of bytes

For example,

```
y0:    .word  -14
b:    .byte  24, 62, 1
arr:    .space 60
y1:    .word  17
```

declares labels `y0`, `b`, `arr`, `y1` which stand for addresses that you can use in your program. You would *not* use these labels in branch instructions, since these addresses are in the data segment, not in the text (instruction) segment. Rather, you use them as variables. The variable `y0` is initialized to have value -14. The variable `b` is an array of 3 bytes. The variable `arr` is pointer to the start of 60 consecutive bytes of Memory. This region can be used in a flexible way. For example, it could be used to hold an array of 15 integers.

Suppose we wanted to swap the values that are referenced by `y0` and `y1`. How would you do this? If you are programming in C or in Java, you use a temporary variable. You copy say the value from `y0` to the temporary variable, then copy `y1` to `y0`, and then copy the value from the temporary variable to `y1`. But this assumes you can copy a value directly from one variable to another. In MIPS, you cannot do this if the variables are in Memory (as opposed to being in a register). Here's how you would do it in MIPS.

```
la      $s0,  y0             #   load addresses of variables to swap
la      $s1,  y1
lw      $s2,  0($s0)         #   load contents into registers
lw      $s3,  0($s1)
move    $t0,  $s2            #   swap the values in the registers
move    $s2,  $s3            #   ("move" is a pseudoinstruction)
move    $s3,  $t0
sw      $s2,  0($s0)         #   store the swapped values to Memory
sw      $s3,  0($s1)
```

See `http://www.cim.mcgill.ca/~langer/273/swap.asm`

NOTE: As a student observed in class, the *move* instructions could be avoided entirely. You could drop them and replace the last two `sw` instructions with:

```
sw      $s2,  0($s1)
sw      $s3,  0($s0)
```

The reason for leaving the longer way in the notes is to make sure you see how to swap register values.

System Calls

We have seen how data can be defined using assembler directives. It is often more useful to use input/output (I/O) to read data into a program at runtime, and also to write e.g. to a console that a user can read, or to a file. Here I will describe how to use the console for reading and writing.

There is a MIPS instruction `syscall` which tells the operating system that you want to do I/O (or some other operation that requires the operating system, such as ending the program). You need to specify a parameter to distinguish between several different system calls. You use the register `$2` for this, also known as `$v0`. To print to the console, use values 1,2,3, 4 which are for the case of int, float, double, or string, respectively. To read from the console, use values 5,6,7,8 for int, float, double, or string, respectively. Of course, when you are reading or printing, you need to specify which register(s) hold the values/addresses you are reading from/to. You can easily look up how to do this.

I gave one example in the lecture slides, namely how to read a string.

```
la  $a0,  str   #  la is a pseudoinstruction ("load address")
li  $v0,  4     #  read string
syscall
```

This gives you all you need to start Assignment 1. Happy Valentine's Day!