In the past two lectures, we discussed MIPS operations on integers. Today we will consider a few data structures that you are familiar with, namely arrays and strings, and discuss how to implement them in MIPS.

Arrays

Suppose we wish to declare and use an array of `N` integers. How would we represent this array in MIPS? If we were programming in C, we might let the array be `a[ ]`. Let the address of the first element of the array 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:

$$\texttt{a}[12] = \texttt{a}[10];$$

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:

$$\texttt{lw} \quad \$t1, 40(\$s0)$$

$$\texttt{sw} \quad \$t1, 48(\$s0)$$

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,

$$\texttt{m} = \texttt{a}[\texttt{i}];$$

Suppose the integer variable `m` is stored in register `$s1` and integer index `i` is stored in register `$s2`. We multiply the contents of `$s2` by 4 to get the offset, and then we add the offset to the base address which we'll assume is in `$s0` as in the above example. Note that we compute the offset by multiplying `i` by 4 by using the `sll` instruction, 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

In the C programming language (COMP 206), a variable of type "char" uses one byte (`http://www.asciitable.com`). Thus, each character in a string is stored in one byte.

Previously, we saw the instructions `lw` and `sw` which load a word from Memory or store a word to Memory, respectively. There are similar instructions for single bytes. `lb` loads a byte, and `sb` stores a byte. These functions are of I-format. Again, we specify the address of the byte with a base address plus an offset. The offset is a signed number.

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

`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 compute the length of a string. In C, a string is a sequence of bytes, terminated by the ASCII NULL character which is denoted '\0' and which has byte value 0x00.

```
char  *str;   //  Declare a pointer to a string.
              //  str is an address (a 32 bit number).
int  ct;
str = "I love COMP 273";

while ( *(str + ct) != '\0' ){
    ct++;
}
```

The variable `str` is a pointer (or "reference") and so its value is an address, which is just an unsigned integer (32 bits). The fancy instruction above then adds an offset `ct` to this address. The * operator *dereferences* that address, which means that it returns the content that is stored at that address. If you are taking COMP 206 now, then you will learn about dereferencing soon. (Joseph told me so.)

Here's is MIPS code that does roughly the same thing as the `while` loop part of the above code. (Let's not deal with the initialization part of the code that comes before the while loop. For now, just accept that the first instruction `la` "loads the address" of the string `str` into register `$s0`. Also, assume `ct` is in register `$s1`.)

```
        la     $s0, str                # pseudoinstruction (load address)
        add    $s1, $zero, $zero       # initialize ct,    $s1 = 0.
loop:   add    $t0, $s0,   $s1         # address of byte to examine next
        lb     $t1, 0( $t0 )           # load that byte to get *(s + ct)
        beq    $t1, $zero, exit        # branch if  *(s + ct) ==  '\0'
        addi   $s1, $s1, 1             # increment ct
        j      loop
exit:
```

Register $t1 holds the ct^{th} char (a byte) and this byte is stored in the lower 8 bits of the register. Note also that I have used `$t` registers for temporary values that do not get assigned to variables in the C program, and I have used `$s` registers for variables. I didn't need to do this, but I find it cleaner to do it.

Assembler directives

We have seen many MIPS instructions. We now would like to put them together and write 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 on what (data) to put where (address).

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 the string got there. Below is some code program that defines a string in MIPS Memory and that counts the number of characters in the string.

The program begins with several assembler directives (`.data, .asciiz`, ...) followed by a main program for computing string length. The `.data` directive says that the program is about to declare data that should be put into the data segment of Memory. Next, the instruction label `str` just defines an address that you can use by name when you are programming. It allows you to refer a data item (in this case, a string). We'll see how this is done later. In particular, the line has an `.asciiz` directive which declares a string, "COMP 273". When the program below uses the label `str`, the assembler translates this label into a number. This number can be either an offset value which is stored in the immediate field (16 bits) of I-format instruction or a 26 bit offset which in a J format instruction.

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 "COMP 273"       #  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*. MARS translates this instruction into

```
lui  $s0,  4097
```

Note that $(4097)_{10} = 0x1001$, 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 MARS.)

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.

```
tmp = y0;
y0  = y1;
y1  = tmp;
```

Suppose `y0` and `y1` are in registers `$s0` and `$s1` respectively. You could use the pseudoinstruction `move`, or

```
move    $t0,  $s0        #  same as  "add  $t0, $s0,  $zero"
move    $s0,  $s1
move    $s1,  $t0
```

However, if the variables are in Memory rather than in registers, then you would first need to load them into Memory, do the swap, and then store them back in Memory. Alternatively, you could just do it like this.

```
la      $t0,  y0              #   load addresses of variables to swap
la      $t1,  y1              #   assuming they are labelled y0 and y1

lw      $s0,  0($t0)          #   load contents into registers
lw      $s1,  0($t1)
sw      $s1,  0($t0)          #   store in opposite position
sw      $s0,  0($t1)
```

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 MIPS kernel/OS that you want to do I/O or some other operation that requires the operating system, such as exiting 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. When you are reading or printing, you need to specify which register(s) hold the values/addresses you are reading from/to. *There is no need to memorize this stuff. Consult the documentation on MARS.*

Here is an example of how to print a string to the console.

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

[MODIFIED Feb 17. The notes I originally posted contained a mistake.]

Here is an example of how to read a string from console. You need to specify that you want to read a string. You need to specify where the string will be put in Memory. You need to specify a buffer size for the kernel to use to read the string. (See Assignment 2 Part 4.) Suppose the maximum buffer size for reading a string is specified by a integer value which is stored in the data segment at label `sizeBuffer` as in Assignment 2. Here is the code for reading a string.

```
li    $v0, 8              #  code for reading string is 8.
add   $a0, $zero, $s1     #  suppose $s1 specifies address
                          #      where string will start

la    $t0, sizeBuffer     #  specifies a buffer size for reading string.
lw    $a1, 0($t0)         #  load that buffer size.
syscall
```

In the slides I then discussed Assignment 2.