CS 252 (Spring 17): Computer Organization

Project #10

Arrays of Strings; MIPS to C

due at 5pm, Fri 14 Apr 2017

1 Overview

In this project, we'll be going a little further with arrays. We'll also be experimenting with a new variant on Projects - converting MIPS back to C.

As with previous MIPS projects, I will provide a few functions which you must implement; write a `.s` file which includes these two functions. I have provided a set of testcases, which we will link with your code to see if it works properly.

For the C code, I will provide a `.s` file with **no comments**. You will have two jobs: first, to comment the code so that the MIPS is more readable; second, to convert it to C. We will also provide a set of testcases to test this code - but most of your points (for this part) will come from the quality of your comments in the assembly. Some points (but less) will come from the quality of your C code; some points (but even less) will come from correctly matching the testcase output.

Note that this function, since it has two very different types of tasks, has two different grading scripts. Use both of them!

1.1 Arrays of Pointers

In previous assignments, you've seen arrays in MIPS - but they have been arrays of bytes and words. In this project, you'll go a step further, seeing arrays of pointers. Of course, a pointer in MIPS32 is a 32-bit value; this means that reading a pointer works the same way as reading an `int`. But, instead of using the value that you read as a number, you will use it as the address of something else.

A simple example is an array of strings. In MIPS, we can declare a pointer to a string using a label and `.asciiz` :

```
str1:    .asciiz   "The quick brown fox jumps over the lazy dog."
str2:    .asciiz   "This is a different string.\n"
str3:    .asciiz   "Badger Badger Badger Badger Mushroom Mushroom"
```

You can declare an array of strings just like any other array of words - but the **value** of each word is a label!

```
array:
        .word    str1
        .word    str2
        .word    str3
```

This array could be null-terminated (a zero value after the last pointer), or we could have a counter.

In C, single strings are pointers to characters - and so an array of pointers to strings is represented by a pointer-pointer to characters:

```
char   *str1   = "The quick brown fox jumps over the lazy dog.";
char   *str2   = "This is a different string.\n";
char   *str3   = "Badger Badger Badger Badger Mushroom Mushroom";

char  *array[] = {str1, str2, str3};    // array of pointers

char  **arrPtr = array;   // this points to the first element of the array
```

(In this project, you won't need to write the C code above, but I provide it for reference. All you will need to use are pointer-pointers as function parameters.)

1.2 Required Filenames to Turn in

You will turn in three files:

- Name your assembly language file `proj10_part1.s`.
- Add comments to the file I provide (`proj10_part2.s`), and then turn in the updated file. **Do not change the code** - only add comments.
- Write a C file which implements the same function as in the part 2 file that I provided; name this file `proj10_part2.c`.

1.3 Allowable Instructions

When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- `add, addi, sub, addu, addiu, subu`
- `beq, bne, j, jal, jr`
- `slt, slti`
- `and, andi, or, ori, xor, xori, nor`
- `sll, srl, sra`
- `lw, lh, lb, sw, sh, sb`
- `mult, div, mfhi, mflo`
  There's no real need for multiply and divice. But I'll let you use them if you think that it would be fun.
- `la, lui`

- `syscall`

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), **do not use them!** We want you to learn the fundamentals of how assembly language works - you can use fancy tricks after this class is over.

2 C → MIPS Tasks

I have provided C code for a few functions below. You will implement each function in MIPS assembly, in the file `proj10_part1.s`.

2.1 `printManyStrings()`

This function helps you get your feet wet with the idea that we can have arrays of pointers. In this case, we'll use a NULL-terminated array of strings. Just like a null-terminated string is an array of characters that ends with a zero character, a NULL-terminated array of strings is an array of strings that ends with a `NULL` pointer.

```
void printManyStrings(char **strings)
{
    for (int i=0; strings[i] != NULL; i++)
        printf("%d: %s\n", i, strings[i]);
}
```

2.2 `swapTwoStrings()`

This function takes a pointer-pointer, and two indices into that array. You must swap the two strings in the array. **Do not move the strings themselves; only move the pointers.**

```
void swapTwoStrings(char **strings, int x, int y)
{
    char *tmp1 = strings[x];
    char *tmp2 = strings[y];

    strings[x] = tmp2;
    strings[y] = tmp1;
}
```

2.3 printSeveralStrlen()

This function takes a pointer-pointer to characters (remember, this is an array of pointers). It also takes a length, as an integer. It must iterate through the strings in the array, and print out each length.

I have provided two possible implementations; I recommend the second one, but it will require you to call one function from another.

VERSION 1

```
void printSeveralStrlen(char **strings, int count)
{
    for (int i=0; i<count; i++)
    {
        char *str = strings[i];

        int len = 0;
        while (str[len] != '\0')    // remember, '\0' is a byte that is zero.
            len++;

        printf("%d\n", len);
    }
}
```

VERSION 2

```
void printSeveralStrlen(char **strings, int count)
{
    for (int i=0; i<count; i++)
        printf("%d\n", myStrlen(strings[i]));
}

int myStrlen(char *str)
{
    int len = 0;
    while (str[len] != '\0')    // remember, '\0' is a byte that is zero.
        len++;

    return len;
}
```

3 MIPS → C Task

I have provided some MIPS assembly in the file `proj10_part2.s`. This code has some labels, but no comments. (The labels are **not** intended to trick you. They are good names, so use them as hints.)

First, extensively comment the code. What does each line do? What appear to be the key variables? What do the branches do - can you find loops, `if()` statements, etc? Can you decode the various comparison and branch instructions to create simple, readable C conditions? Which registers appear to represent variables, and which are simply temporaries, used once and then discarded?

I will give you a hint: the function takes two parameters (a `int*` and an `int`). A good name for the first would be "`array`."

Second, once you have deconstructed the assembly, convert it back to C code. Try to write clear C code that is easy to read, so that it is easy to understand what the function does. If you have some parts of the program you don't understand, don't skip them! Instead, include them in the function, matching the assembly as precisely as you can. Hopefully, as you understand more of the program, these parts will become clear - and you can replace them with better C code.

While your C code should be easy to read, **do not** try to improve it. You are pretending to be a very stupid "decompiler" - so do **not** optimize the code, change how it works, or debug it. Simply implement exactly what the assembly does.

Once you have a C function written, the grading script will test your code against a set of C testcases. Can you match the output?

4 Rubric

The points for this project will be divided as follows:

Part 1 (C to MIPS):

- 40% for passing the testcases
- 15% for comments, indentation, and style of your MIPS code

Part 2 (MIPS to C):

- 30% for the quality of the comments that you added
- 10% for the clarity and readability of your C code
- 5% for passing the testcases for part 2 (MIPS to C)

4.1 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

4.2 Testcases

You can find a set of testcases for this project at
`http://www.cs.arizona.edu/classes/cs252/spring17/projects/proj10/`

For assembly language programs, our testcases will be other `.s` files, which we will append to your program. (We do this because our simulator sometimes gets confused by multiple files.) In this Project, the testcase will provide the definition for all of the variables that you need to read.

The testcases will be named `test_*.s`. Each testcase has a matching output file, which ends in `.out`; our grading script needs to have both files available in order to test your code.

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcases of your own, in order to better test your code.**

4.3 Automatic Testing

As we normally do, we have provided grading scripts. However, since we're doing two different types of things (MIPS to C, and C to MIPS) - and especially because I want to keep their score separate - I have provided two grading scripts, instead of one.

Run both. `grade_proj10_part1` is used to grade the MIPS assembly that you write (`proj10_part1.s`). `grade_proj10_part2` is used to grade the C code that you write (`proj10_part2.s`).

4.4 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.

While you normally cannot share code with friends and classmates, **testcases are the exception.** We encourage you to share you testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

5 Turning in Your Solution

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only your program; do not turn in any testcases or other files.