CS 252 (Spring 17): Computer Organization

Project #12

Recursion

due at 5pm, **Wed** 3 May 2017 - **Last Day of Classes!**

NO LATE DAYS

1 Overview

It's the last project of the semester - and we'll be working with recursion. As with Project 10, we'll have two parts: writing MIPS, and converting MIPS code to C.

Both parts of this project will use recursion in their solutions.

1.1 Warning: Bugs Ahead!

I have intentionally left one or more bugs in the MIPS code (for part 2, the MIPS-to-C conversion). They are there on purpose; I'm checking to see if you are **actually** translating the assembly (not just making a guess). Your C code should faithfully reflect exactly what the MIPS code does. Don't fix the bugs.[1]

1.2 Required Filenames to Turn in

You will turn in three files:

- Name your assembly language file `proj12_part1.s`.
- Add comments to the file I provide (`proj12_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 `proj12_part2.c`.

  (This time, I've provided a C file to get you started. Rename it, and modify it.)

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`

[1] Of course, if you find a problem with **my** C code - that is, the output expected doesn't match the actual MIPS code - then be sure to tell me about it on Piazza!

- `and, andi, or, ori, xor, xori, nor`
- `sll, srl, sra`
- `lw, lh, lb, sw, sh, sb`
- `mul, mult, div, mfhi, mflo`
  NOTE: See the description below about the `mul` instruction - which is **not** the same as `mult`, which I've mentioned before!
- `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.

1.4 `mul` - A Simpler Multiply

We've talked about the `mult,div` instructions in the past - which have 64 bits of output each, and so write to the `lo,hi` special registers. However, during this semester, a student pointed out a **second** multiply instruction to me:

```
mul $s0, $s1, $s2
```

This is an R-format instruction (although it uses opcode 0x1c, not 0x00) which performs multiplication, but **only stores the bottom 32 bits of the answer.**

Please use this instruction to perform multiplication in your code.

2 C → MIPS Tasks

I have provided C code (or descriptions) for a few functions below. You will implement each function in MIPS assembly, in the file `proj12_part1.s`.

2.1 `printWithLen()`

This is a simple utility function (non-recursive), which will be useful for your implementation of the larger program. It takes two parameters: a pointer to a string, and a count of bytes. Print out exactly that many characters; typically this function will stop printing **before** the null terminator.

Do not modify the input string; loop over it, and print each character individually. Do **not** add any extra characters at the beginning or end; the calling code is responsible for adding a newline after you are done.

I will not provide C code for this function; you can write it on your own.

2.2 `formulaRecurse()`

This function takes a single parameter: a pointer to a string. The string contains a mathematical formula, which you will break down into sub-strings, and then calculate the value of.

To simplify parsing, the input string must only be made up of single digits, plus and minus signs, and parentheses. No negative signs, and no multi-digit numbers are allowed. (Also no whitespace!) In addition, parens must be used to ensure that every expression has **exactly** two operands. That is,

```
1+2*3
```

is illegal because it has three operands, joined by two operators. Instead, the valid form would be:

```
1+(2*3)
```

Note that the top-level expression must also have exactly two operands; so the following expression is also invalid:

```
(1+(2*3))
```

Your function will parse the characters of the input string, and find the values of two sub-expressions, `val1,val2`. To do so, it will either interpret simple digits - or it will recurse to calcualte the value of parenthesized expressions.

There is one critical way in which the code below is **not valid C code:** your function must return **two** values. The first, stored in `$v0`, is the **length** of the sub-expression you just parsed; this will often be less than the entire string. The second, stored in `$v1`, is the **value** of that expression.

Finally, in order to help debug (and to show what you're doing), `formulaRecurse()` must print out the sub-expression that it evalauated (often, less than the entire string), and the value that it is returning.

For example, here is the input string from testcase 8:

```
5+(6*7)
```

and here is the expected output:

```
6*7
42
5+(6*7)
47
len=7
retval=47
```

The first two lines of output come from two different calls to `formulaRecurse()`; the last two come from the testcase.

The C code for this task is as follows:

```
(int,int) formulaRecurse(char *buf)
{
    int pos = 0;
    int val1,val2;
    int len;

    if (buf[0] == '(')
    {
        pos++;
        (len,val1) = formulaRecurse(buf+pos);
        pos += len+1;
    }
    else
    {
        val1 = buf[pos]-'0';
        pos++;
    }

    int operator = buf[pos];
    pos++;

    if (buf[pos] == '(')
    {
        pos++;
        (len,val2) = formulaRecurse(buf+pos);
        pos += len+1;
    }
    else
    {
        val2 = buf[pos]-'0';
        pos++;
    }

    printWithLen(buf,pos);
    printf("\n");

    int retval;
    if (operator == '+')
        retval = val1+val2
    else
        retval = val1*val2

    printf("%d\n", retval);

    return (pos, ret);
}
```

3 MIPS → C Task

I have provided some MIPS assembly in the file `proj12_part2.s`. This code has three recursive functions; each takes one or two parameters - and the first argument is always a pointer to a struct. To make your life a little easier, I have provided a few comments in the MIPS - but I've also provided a C file to get you started.

First, extensively comment the code. (**Do not skip this step, or you will lose points.**) 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?

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):

- 35% for passing the testcases
- 15% for comments, indentation, and style of your MIPS code

Part 2 (MIPS to C):

- 35% for passing the testcases
- 10% for the quality of the comments that you added to `proj12_part2.s`
- 5% for the clarity and readability of your C code

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/proj12/`

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_proj12_part1` is used to grade the MIPS assembly that you write (`proj12_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.