CPSC 213: Assignment 3

Due: Wednesday, February 1, 2012 at 6pm.

Late assignments are accepted until Friday, February 3 at 6pm with a 25% penalty per day (or fraction of a day) past the due date. This rule is strictly applied and there are no exceptions.

Goal

There are three goals for this assignment. The first goal is to implement all the remaining instructions from Assignment 2 (if you have not done so) and fully test all of the SM213 instructions you implemented. If you already did all of this as part of Assignment 2, there is nothing to do here for you other than to turn in again what you did last week.

The second goal is to examine how access to struct/instance variables is implemented in SM213 assembly. You use the simulator to carefully observe the execution of Snippet 4. Then you will synthesize what you have learned from your careful examination of Snippets 1-4 to write a small assembly language program and analyze it.

Finally, the last goal is to examine dynamic allocation and de-allocation of memory in C. You will see the danger of dangling pointers and get a taste for how to avoid them.

Testing the SM213 Implementation

In Assignment 2, you implemented a subset of the SM213 ISA, but you were not required to test all of these instructions. Now you are.

Use a text editor to create a plain text file whose name ends with an extension ".s", which contains a set of SM213 instructions that provides complete test coverage for these instructions. You must test each of the instructions you implemented for Assignment 2 and consider what edge conditions exist and test these cases as well. Turn in this test file and a careful explanation of your overall test plan, how you handled edge cases, and what you saw when you performed the tests (i.e., does your implementation pass the tests). This explanation can generalize across groups of instructions that are tested in a similar way; it is not necessary (or good) to describe every single test instruction.

Implementing Struct/Instance Variable Access

There is one snippet for this week.

- S4-instance-var

As you did last week, load this snippet into the SM213 and single step through its execution. Turn on animation and run it slowly. Slow the animation by hitting the `Slower` button. Hit the `Pause` button when you want to take a longer look. Carefully summarize what you observe.

Now, combine your understanding of S1, S2 and S4 to answer the following questions about this piece of C code.

```
            struct S {
                  int          x[2];
                  int*         y;
                  struct S*   z;
            };

            int         i;
            int         v;
            struct S   s;

            void foo () {
                  v = s.x[i];
                  v = s.y[i];
                  v = s.z->x[i];
            }
```

1. Implement this code in SM213 assembly, by following these steps:

   a. Create a new SM213 assembly code file with three sections, each with its own `.pos`: one for code, one for the static data, and one for the "heap". Something like this:

```
.pos 0x1000
            code:

            .pos 0x2000
            static:

            .pos 0x3000
            heap:
```

   b. Using labels and `.long` directives allocate the variables `i`, `v`, and `s` in the static data section. Something like this (the ellipsis indicates more lines like the previous one) :

```
.pos 0x2000
            static:
            i:          .long 0
            v:                .long 0
            s:                .long 0
                        .long 0
                        ...
```

   c. Implement the three statements of the procedure `foo` (not any other part of the procedure) in SM213 assembly in the code section of your file. Comment every line carefully.

   d. Initialize the variable `s.y` to store a pointer to the beginning of the `heap` section, as if the program had called `malloc` to allocate an array of 3 integers. What you are doing here is modeling some of the dynamic calculation of the program (the `malloc` and initialization of `s.y`) so that you can test the code you have written. Something like this:

```
.pos 0x3000
            heap0:      .long 0
```

```
                .long 0
                .long 0
```

e. Initialize the variable `s.z` to store a pointer to the next available part of the "heap" section (i.e., right after the three `.long` elements of heap0). This part of the heap should have one `.long` for every element of `struct S`. Something like this:

```
heap1:          .long 0
                .long 0
                ...
```

f. Test your code for a few different values of `i`, `s.x[0...1]`, `s.y[0…2]`, and `s.z->x[0…2]`.

2. Use the simulator to help you answer these questions about this code.

   a. How many memory reads are required to implement the first line of `foo()`?

   b. How many more memory reads are required to implement the second line of `foo()`?

   c. How many more memory reads are required to implement the third line of `foo()`?

Dangling Pointers in C

Included with the assignment is a C program called `dangling-pointers.c`. This program implements a stack and consists of 6 tests that can be performed on it. Compile the program and run it from the command line. For example, these two lines typed at the command line compile the program and execute Test 1.

```
gcc -o dangling-pointers dangling-pointers.c
./dangling-pointers 1
```

Odd numbered tests appear to work while even numbered tests demonstrate symptoms of one or more bugs. For Tests 2 and 4 the bug is in the test itself. Test 6, however, demonstrates a bug in the stack implementation. Tests 1-4 are really a warm up for this one. The key observation you should make is that even though Test 5 appears to work, there really is a serious bug lurking. If your test suite included only tests like Test 5, you might miss this bug. Then when your customer ran something along the lines of Test 6 and saw crazy behaviour, they might stop payment on their cheque.

Take the tests in groups of two to identify and correct each of the bugs in tests 2, 4 and 6. Clearly explain the cause of the bug and how you fixed it. Fixing the bug in the stack implementation, the bug illustrated by test 6, is not easy. The solution involves considering the key C de-allocation issues we discussed in class. Consider these issues and make a decision about how to solve the problem. You will likely need to change the interface to the stack procedures. The basic structure of the tests should remain the same, however. And, importantly, you must eliminate the bug. That is, ANY test that conforms to your new interface must work without showing the symptoms that test 6 does.

Material Provided

In the file code.zip:

1. `S4-instance-var.{java, c, s}`
2. `dangling-pointers.c`

What to Hand In

Use the **handin** program. The assignment directory is **a3**. Please hand in exactly the following files with the specified names. Do not hand in class files, or your entire Eclipse project, or a README in formats like .doc or .rtf.

1. `CPU.java`.
2. `dangling-pointers.c` with the bugs fixed.
3. `test-lab3.s` testing all the instructions you implement.
4. `struct.s` for the assembly implementation of the C code above.
5. `README.txt` that contains:
    - header with your name, student number, four-digit cs-department undergraduate id (e.g., the one that's something like a0b1)
    - statement that the work you are handing in is your own, and that you read through and understood the course policies on plagiarism, cheating, and academic misconduct posted at http://www.ugrad.cs.ubc.ca/~cs213/winter11t2/policies.html
    - Test description for `test-lab3.s`. Did all of the tests succeed? Does your implementation work?
    - Observations from running snippet `S4-instance-var.s`.
    - Test result for `struct.s` program and answers to Question 2 above.
    - Description of bug in each of dangling-pointers tests 2, 4, and 6, and whether your fixes work, and how you test to your fixes.