CPSC 213: Assignment 4

Last Modified: October 2, 10:55am (minor typographical change)

Due: Sunday October 5, 2014 at 11:59pm

After an eight-hour grace period, no late assignments accepted.

Goal

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

Then you will extend the SM213 implementation to support static control flow, including static procedure calls. You will use this expanded machine to examine the compiler implementation of for-loops, if-then-else statements, procedure calls and return. One goal is to understand the role of pc-relative addressing, conditional and unconditional branch statements in support of high-level language constructs such as loops and if, connecting your existing understanding of these language features to what you now know about the execution of programs in hardware. Another goal is to understand the role of the PC register in implementing jumps to see that a jump is really just an assignment of a new value to this register. The final goal is to contrast the use of static and dynamic jumps in implement procedure call and return and to understand how the return instruction knows to which instruction it should jump.

Groups

To encourage you to be more interactive with each other in labs and to give you an additional way to get help with the learning goals of assignments — helping each other — you may do this assignment in groups of two if you like. If you do, be sure that you both contribute equally to the assignment and that you each work on every part of the assignment. Do ***not*** split up the assignment in such a way that one of you does one part and the other does another part. Keep in mind that the core benefit to you of doing the assignment is the learning that happens while you do it. Each assignment is worth only around 1.8% of your grade, but what you learn while doing the assignment goes a long way to determining the other 85%.

If you choose to do the entire assignment with a partner, submit one solution via **`handin`** and list both of your names, student numbers, and computer-science login ids in the **`README`** file.

Alternatively, you can also choose to collaborate with another student for a portion of the assignment and have your work marked individually. Do do this, simply submit your own version via **`handin`** and list the other student as a collaborator in your **`README`** file. Just don't do this if you and your partner are turning in identical solutions, as we would like to realize marking efficiency where possible. You may

have at most one collaborator for the assignment; i.e., you can not collaborate with one student for one part of the assignment and another student for another part.

Download

Download the file www.ugrad.cs.ubc.ca/~cs213/cur/Assignments/a4/code.zip, containing:

1. dangling-pointers.c
2. S5-loop.{java,c,s}
3. S5a-loop-unrolled.{c,s}
4. S6-if.{java,c,s}
5. S7-static-call.c
6. S7-static-call-regs.s

Part I: Dangling Pointers in C

The program **`dangling-pointers.c`** that you downloaded implements a stack data structure and consists of 6 tests that can be performed on it. A stack has two operations: push and pop. Push adds strings to the stack and pop removes them, in last-in-first-out order. So, if you push "one", "two", and then "three", in that order, three pops will give you "three", "two", "one", in that order.

Compile the program and run it from the command line. The program takes one parameter, the test number to run. 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.

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 deallocation 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, including those that push an arbitrary number of items onto the stack, must work without showing the symptoms that test 6 does.

Part II: Static Control Flow

Extending the ISA

Implement these six control-flow instructions in CPU.java. Note that in the "Format" column, capital letters are hex digit literals and lower-case letters represent hex values referenced in the "Assembly" and "Semantics" columns.

Instruction	Assembly	Format	Semantics
branch	`br A`	`8-pp`	pc ← (A = pc + pp*2)
branch if equal	`beq rc, A`	`9cpp`	pc ← (A = pc + pp*2) if r[c]==0
branch if greater	`bgt rc, A`	`Acpp`	pc ← (A = pc + pp*2) if r[c]>0
jump	`j A`	`B--- AAAAAAAA`	pc ← A
get pc	`gpc $o, rd`	`6Fpd`	r[d] ← pc + (o == p*2)
indirect jump	`j o(rt)`	`Ctpp`	pc ← r[t] + (o == pp*2)

Do the Following

Here are the problems to solve for this week's assignment.

1. Implement the six control-flow instructions one at a time.
2. Extend your **`test.s`** test program to include tests for each of the new instructions.
3. Compare the C versions of snippets S5 and S5a, then compare their assembly code by running them in the simulator, S5a first. Document the changes you see in memory and registers for the first few iterations of the loop and the last one.
4. Examine S6 and S7. Step the assembly through the simulator. Document the changes you see in memory and registers.

What to Hand In

Use the **`handin`** program. The assignment directory is **`a4`**.

1. A single file called "**`README.txt`**" that includes your name, student number, four- or five-digit cs-department undergraduate id (e.g., the one that's something like a0b1).
2. If you had a partner for the entire assignment, turn in only one copy of your joint solution under one of your ids and list both student's complete information in the **`README.txt`** file and include the text "**`PARTNER - MARK JOINTLY`**".
3. If, on the other hand, you collaborated with another student for a portion of the assignment, but your solutions are not identical and you want them marked individually, each of you should include the other student's complete information in your **`README.txt`** file, include the text "**`COLLABORATOR - MARK SEPARATELY`**", and turn in copies separately via **`handin`**.

4. The **`README.txt`** file must also include all written material required by the assignment as listed below.
5. In **`README.txt`**: description of the bugs in dangling-pointers.c tests 2, 4, and 6 and how you fixed them.
6. You modified dangling-pointers.c with the bugs fixed.
7. Your modified CPU.java
8. Your extended test program, **`test.s`**.
9. In **`README.txt`**: your test description. Did all of the tests succeed? Does your implementation work?
10. In **`README.txt`**: a written description of the key things you noted about the machine execution while running snippets (S5, S6, and S7), including observations required by Questions 3 and 4. Keep it brief, but point out what each instruction read and wrote etc.