CPSC 213: Assignment 3

Due: Sunday, October 3, 2010 at 6:00pm

Goal

There are three goals for this assignment. The first is to evaluate a snippet of code that performs pointer arithmetic in C.

The second is to continue the SM213 implementation to add support for instance variables and to add the ALU instructions, a halt instruction and an instruction that does nothing, called NOP.

The third is to examine dynamic allocation and de-allocation in C and Java. For C, the goal is to see the danger of dangling pointers and to get a taste for how to avoid them. For Java, the goal is to see how ignoring deallocation can lead to a serious bug called a "memory leak", to see the consequences of this bug, and to see how to use *reference objects* to avoid the bug.

The ISA to Implement

In this assignment, you change your implementation of the load/store base + offset instructions from Assignment 2 to now include a small static offset as part of the instruction. This change is helpful for implementing access to instance variables, which are at a static position within their containing object/struct, but where the address of the object/struct itself is usually a dynamic value.

Here are the revised semantics of these two instructions.

Instruction	Assembly	Format	Semantics
load base + offset	ld o(rs), rd	1isd	r[rd] ← m[i*4+r[rs]]
store base + offset	st rs, o(rd)	3sid	m[i*4+r[rd]] ← r[rs]

You will note that the assembly-language template uses the symbol "o" for the static offset while the machine code template uses the symbol "i". The difference is that the assembly language specifies a byte offset that we require be divisible by 4. The machine code stores that value divided by 4 for compactness of instructions. And thus, o = i * 4, as indicated in the Semantics column.

You will also implement the ALU, halt, and nop instructions this week so that they are available for subsequent assignments. Here is a summary of their semantics.

Instruction	Assembly	Format	Semantics
rr move	mov rs, rd	60sd	r[d] ← r[s]

Instruction	Assembly	Format	Semantics
add	add rs, rd	61sd	r[d] ← r[d] + r[s]
and	and rs, rd	62sd	r[d] ← r[d] & r[s]
inc	inc rd	63-d	r[d] ← r[d] + 1
inc addr	inca, rd	64-d	r[d] ← r[d] + 4
dec	dec rd	65-d	r[d] ← r[d] - 1
dec addr	deca, rd	66-d	r[d] ← r[d] - 4
not	not rd	67-d	r[d] ← ~r[d]
shift	shl $v, rd shr $v, rd	7dss	r[d] ← r[d] << s *s = v for left and -v for right*
halt	halt	f000	throw halt exception
nop	nop	ff00	do nothing (nop)

Code Snippets You Will Evaluate

As explained in detail below, you will use the following code snippets this week. There are C, Java and SM213 Assembly versions of each of these (except the C-pointer-math file, for which there is no Java).

- S3-C-pointer-math
- S4-instance-var

Your Simulator Implementation

You are implementing two methods of the CPU class in the Arch.SM213.Machine.Student package of the SM213 Simulator.

1. **fetch ()** loads instructions from memory into the **instruction** register, determines their length and adds this number to the **pc** register so that it points to the *next* instruction, and then loads the various pieces of the instruction into the registers **insOpCode**, **insOp0**, **insOp1**, **insOp2**, **insOpImm** and **insOpExt** (for 6 byte instructions). The meaning of each of these registers and a primmer on the Java syntax for accessing them was given in class and is part of the online lecture slides and Companion notes. *The only change you*

will make to your fetch method is to add insOpImm (the second byte of the instruction word) to the set of registers you fetch.

2. **execute ()** uses the register values stored by the fetch stage to execute the instructions, transforming the register file (i.e,. **reg**) and main memory (i.e., **mem**) appropriately.

Using the Simulator to Test and Debug Your Code

The simulator displays the current value of the register file, main memory and the internal registers such as the pc and instruction registers.

You will use the simulator GUI to test and debug your ISA implementation and to analyze the code snippets. In each case you should single-step through the machine code and carefully observe the state changes that occur in the process registers, register file and main memory.

Here are a few quick things that you will find helpful.

1. You can edit instructions and data values directly in the simulator (including adding new lines or deleting them).
2. The simulator allows you to place "labels" on code and data lines. This label can then be used as a substitute for the address of those lines. For example, the variable's **a** and **b** are at addresses 0x1000 and 0x2000 respectively, but can just be referred to using the labels **a** and **b**. Keep in mind, however, that this is just an simulator/assembly-code trick, the machine instructions still have the address hardcoded in them. You can see the machine code of each instruction to the left of the instructions in the *memory image* portion of the instruction pane.
3. You can change the program counter (i.e., **pc**) value by double-clicking on an instruction. And so, if you want to execute a particular instruction, double click it and then press the "Step" button. The instruction pointed to by the **pc** is coloured green.
4. Memory locations and registers read by the execution of an instruction are coloured blue and those written are coloured red. With each step of the machine the colours from previous steps fade so that you can see locations read/written by the past few instructions while distinguishing the cycle in which they were accessed.

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 terminal on a **CS department Sparc server, such as galiano.ugrad.cs.ubc.ca**. 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.

Run the tests in pairs (1 & 2), (3 & 4), (5 & 6) 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 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 must work without showing the symptoms that test 6 does.

Memory Leaks in Java

Also included with the assignment is a Java class in the file MemoryLeak.java. Add this to your Eclipse environment so that you can edit and compile. You will run it from the UNIX command line.

The class has a public static main that can be run from the command line with a test number argument. You should run this class by prefacing the "time" command to get its running time. So, to run test number 1, you would type this.

```
time java MemoryLeak 1
```

The MemoryLeak includes two inner classes called Thing and ThingMogrifier. They have java-doc comments that you should read. Thing's have a unique integer ID and a 128-byte value. The ThingMogrifier creates Things when its client calls "receive". Think of this as modeling a process like a computer network where the ThingMogrifier generates new Things that might be coming from the network and provides them to clients when they ask for them. The ThingMogrifier also catalogues the Things it creates so that clients can later find them with their IDs, if they want. This catalogue is a convenience for the client and it is not meant to interfere with garbage collecting Things when the client is done with them, but as you will see there is a bug here.

There are four tests. Each test has the identical inner loop that calls ThingMogrifier receive 100 times, keeping a local cache of the Things received. It then uses the ThingMogrifier's catalogue to lookup these things by their ID and to verify that the Thing that ThingMogrifier finds is indeed the Thing that the client thinks matches that ID. This is a simple interaction that is meant to model the client performing local operations on a set of Things before moving on to another task. Each test then performs this inner loop multiple times to model a program that performs multiple independent tasks that each interact with the ThingMogrifier. The number of tasks increases by a decimal order of magnitude with each test, starting with test number 1 performing 100 tasks.

You should attempt to run each test and look at their running time. The running time is the left-hand-side number printed when the program terminates. For example, the follow test took 265 ms (the u stands for "user" time, which is all we care about).

```
time java MemoryLeak 1
0.265u 0.043s 0:00.28 107.1%    0+0k 0+1io 0pf+0w
```

You will find that as the number of tasks increases the running time increases, as expected, until a catastrophe occurs, probably on test number 3. You will get an error like this.

```
Exception in thread "main" java.lang.OutOfMemoryError: Java
heap space
 at java.util.HashMap.resize(HashMap.java:462)
 at java.util.HashMap.addEntry(HashMap.java:755)
 at java.util.HashMap.put(HashMap.java:385)
 at MemoryLeak$ThingMogrifier.receive(MemoryLeak.java:34)
 at MemoryLeak.test(MemoryLeak.java:54)
 at MemoryLeak.main(MemoryLeak.java:80)
1.474u 0.136s 0:01.34 119.4%   0+0k 0+0io 0pf+0w
```

Your job is to explain what bug causes this error and then to fix the bug in two different ways by creating new versions of ThingMogrifier called ThingMogrifier1 and ThingMogrifier2. The first of these should fix the bug by adding a method that must be called by clients to flush objects from the catalogue. The second of these should instead fix the bug by using WeakReference objects; this version of the ThingMogrifier should have exactly the same interface as the original. Create 8 new tests, four for each of the new ThingMogrifiers. Run them and determine their running time.

Requirements

Here are the requirements for this week's assignment.

1. Run S3-C-pointer-math.s through your simulator. Note what happens in each step.
2. Carefully explain why the C statement *c = c[3] is equivalent to c[3]=c[6] in S3-C-pointer-math using the original value of c by noting precisely what happens in the simulation for the relevant assembly instructions. In the quiz associated with this lab, you may be asked to provide a similar explanation for a similar program (but without the aid of the simulator) and so take your time here to be sure you really understand this.
3. Implement the required changes to load and store base-plus-offset instructions.
4. Extend your test program to test these changes.
5. Run S4-instance-var through your simulator. Single-step the machine, instruction by instruction, and carefully note what happens in each step.
6. Implement the ALU, halt and nop instructions.
7. Extend your test program to test these instructions.
8. Run dangling-pointer tests 1 and 2. Identify the bug in test 2. Describe it carefully. Fix it. Rerun it.
9. Run dangling-pointer tests 3 and 4. Identify the bug in test 4. Describe it carefully. Fix it. Rerun it.
10. Run dangling-pointer tests 5 and 6. Identify the bug in test 6. Describe it carefully. Fix it. Rerun it.
11. Run MemoryLeak tests 1-4. Examine the code to explain the MemoryLeak bug.
12. Implement ThingMogrifer1 and tests 5-8. Run these tests and record their running time.
13. Implement ThingMogrifer2 and tests 9-12. Run these tests and record their running time.

Material Provided

The Simulator code was provided as part of Assignment 1.

You extended the simulator in Assignments 1 and 2. You may use the solutions to Assignments 1 and 2 as the starting point for this assignment if you like. They will be available on the course website following their due dates.

This assignment includes snippets 3 and 4 in the file snippets-3-4.zip. Dangling-pointers.c and MemoryLeak.java in the file code.zip

What to Hand In

Use the handin program. The assignment directory is **a3**.

1. CPU.java
2. Your test program
3. Your test description. Did all of the tests succeed? Does your implementation work?
4. A written description of the key things you noted about the machine execution while running snippets S3 and S4. Keep it brief, but point out what each instruction read and wrote etc.
5. Your answer to Requirement 2 above.
6. Your modified dangling-pointers.c
7. The explanations required in Questions 8-10.
8. Your modified MemoryLeak.java
9. The explanations required in Questions 11-13.