COMP 40 Laboratory: Testing assembler macro instructions

December 2, 2011

1 Introduction

In this lab, you'll create simple test programs for your assembler macros.

1. You'll start by implementing this C function:

   ```
   bool test_passes(const char *test, Um_Word n1, Um_Word n2, Um_word result);
     // tell if the test produces 'result' in r3 with operands n1 and n2
   ```

   Here are some tests you can run with such a function:

   ```
   test_passes("r3 := r1 + r2", 2, 2, 4); // 2 + 2 == 4
   test_passes("r3 := ~r1", -1, 0, 0);    // ~(-1) == 0
   test_passes("r3 := r1; r3 := r3 | r2", 0xdead0000, 0xbeef, 0xdeadbeef);
      // 0xdead0000 | 0xbeef == 0xdeadbeef
   ```

   The implementation of `test_passes` should look something like this:

   (a) Write assembly code to file `test.ums`. The assembly code should look like the template in Figure 1 on the next page, but you'll fill in the test and the first and second numbers based on the arguments to `test_passes`.

   (b) Call `system("./umasm test.ums > test.um");`

   (c) Call `system("./um test.um > test.1");`

   (d) Read in four bytes from `test.1`, treating them as a 32-bit unsigned integer stored in little-endian byte order. Call the resulting integer `from_disk`.

   (e) Return `from_disk == result`.

2. Write a main program that runs `test_passes` in a loop. You can implement multiple tests, and for each test, you can try multiple operands. You can use hand-picked operands, random operands, or a combination. (You can get random operands by reading from `/dev/urandom`; look at the man page for `fread()`.) If a test fails, issue a suitable error message. If all instances of a given test pass, you might want to write out a message, e.g.,

   ```
   Passed 83 tests of the form "r3 := r1 + r2".
   ```

We recommend that you get your test infrastructure working by testing the add, divide, and multiply instructions, which are built in. After that, you can test your macro instructions.

This is almost exactly the way Professor Ramsey tests his implementations of the macro instructions.

```
.zero r0
.temps r6, r7

// load two numbers into r1 and r2
r1 := m[r0][first_number]
r2 := m[r0][second_number]

// compute test result into r3
r3 := r1 + r2 // this string is argument 'test' to function test_passed()

// output the results byte by byte, little-endian format
r4 := r3 & 0xff
output r4

r3 := r3 / 256
r4 := r3 & 0xff
output r4

r3 := r3 / 256
r4 := r3 & 0xff
output r4

r3 := r3 / 256
r4 := r3 & 0xff
output r4

halt

first_number:
.data 0x7c716e0d  // this number is argument n1 to function test_passed()
second_number:
.data 0xbf6d2eca  // this number is argument n2 to function test_passed()
```

Figure 1: Template for testing macro instructions, filled in with an add test and two operands