MSpec: An RSpec-like unit test framework for MIPS assembly

Zachary Kurmas
School of Computing and Information Systems
Grand Valley State University
Allendale, MI 49401
Phone: 616-331-8688
Fax:616-331-2106
kurmasz@gvsu.edu

Abstract

`MSpec` is a Ruby program that takes `RSpec`-like examples as input and generates `MIPS` assembly-language unit tests. These "spec files" are considerably easier to write than assembly language driver programs. Both instructors and students can run the `MSpec`-generated test files to quickly and easily determine the correctness of assembly code.

`MSpec` improves grading by reducing the time needed to determine correctness, thereby leaving more time for qualitative feedback. More importantly, `MSpec` encourages students to find and correct mistakes while doing the assignment, thereby improving their ability to write and understand assembly. (In contrast, many students who don't discover a mistake until receiving a grade never take the time to understand and correct the mistake.)

1 Motivation

We have found that students hesitate to submit projects with known bugs [2]. If they can quickly and easily find a bug, they will usually take the time to fix it (and, in the process, correct the misconception that led to the bug in the first place). In contrast, by the time students receive formal feedback (e.g., their graded assignment), they have moved on to other topics and rarely go back to correct the misconception that caused the bug in the first place. Thus, we believe that providing assembly language students a simple mechanism of verifying the correctness of their code will improve their learning.

A mechanism of quickly verifying code correctness is also helpful to instructors: Reducing the amount of time spent verifying code correctness leaves more time for qualitative evaluation (e.g., offering the students suggestions on how to write better code).

2 Basic Operation

`MSpec` takes as input an `RSpec`-like file containing test cases and generates a `MIPS` assembly file containing unit tests [1]. When run, the `MSpec`-generated assembly file either prints "All tests pass" on the standard output, or prints the name of the first test that fails. `MSpec` was designed to work with the `Mars MIPS` simulator; but, the generated files should work with `SPIM` as well [3].

`MSpec` input files are considerably easier to write than an entire assembly language driver program. Figure 1 shows some of the tests for an `indexOf` procedure. The `describe` block identifies the procedure under test. Each `it` block defines one test. Each test (1) specifies some input data for the `.data` segment, (2) calls the procedure with the desired parameters, and (3) verifies the state of the CPU after the procedure returns. (This example only verifies the procedure's return value. Instructors may also choose to check for side-effects.)

`MSpec` automates the tedious aspects of writing a driver program including (1) initializing the registers and memory, (2) initializing the "`$a`" registers with parameters prior to a procedure call, (3) verifying the processor state

```
describe :indexOf do
 it "finds value when it occurs once only" do
    data :array1, :word, 5, 4, 9, 8, 2, 1, -1
    call 2, :array1
    verify :v0 => 4
  end

  it "finds value at head of array" do
    data :array2, :word, 5, 4, 7, 6, -1
    call 5, :array1
    verify :v0 => 0
  end
```

Figure 1: Simple `MSpec` input

```
describe :nCk do
 def nCk(n, k)
    return 1 if k == 0 || n == k
    return n if k == 1
   nCk(n-1, k-1) + nCk(n-1, k)
  end

  (0..20).each do |n|
    (0..n).each do |k|
      it "calculates #{n}C#{k}" do
        call n, k
        verify :v0 => nCk(n, k)
      end
    end
  end
end
```

Figure 2: Intermediate `MSpec` input

after the procedure call, and, most importantly, (4) printing helpful error messages in the event of a failure. (Writing assembly code that generates a helpful error message is much more tedious than writing a helpful error message in a high-level language like Java.)

The use of Ruby to define the test cases provides considerable flexibility. For example, the spec shown in Figure 2 generates 231 tests: one for each possible input to "n choose k" for values of n less than 20. (Users who are not familiar with Ruby can generate tests using the simple syntax shown in Figure 1.)

3 Poster content

The poster will focus on how to use `MSpec`, including

- simple, straightforward ways of specifying tests (like Figure 1),
- more advanced techniques of specifying tests (like Figure 2), and
- how to easily run the tests (e.g., using the `Mars` command line).

As space allows, we will also discuss

- Benefits of making it easy for students to test their own code
- Different ways to use `MSpec`:
  - Provide students with a complete set of test cases
  - Provide students a few sample test cases and have the students write the rest
  - Encourage students to share and discuss tests cases with each other
  - Have the students write all the test cases
- `MSpec` design decisions. `MSpec` uses Ruby to generate an assembly language test driver. We also considered
  - Using JRuby and having the program interact directly with `Mars`.
  - Writing a Java program that uses JUnit syntax to produce an assembly language driver program.
  - Writing a Java program that uses JUnit syntax and interacts directly with `Mars`.

References

[1] http://www.rspec.info.

[2] Z. Kurmas. Improving student performance using automated testing of simulated digital logic circuits. In *ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 265–270, New York, NY, USA, 2008. ACM.

[3] K. Vollmar and P. Sanderson. Mars: an education-oriented mips assembly language simulator. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 239–243. ACM, 2006.