Debugging notes

Here are Professor Zilles' notes on debugging. While these are applicable generally, they'll be especially handy during the MIPS labs.

Error rendering macro 'toc'


Test Cases

For test cases, you generally want to test a wide variety of cases. Unfortunately, complex test cases are generally hard to debug. So one thing to do after finding a test case that fails, is to simplify the test case as much as possible, while maintaining the failure.

Simplification in this case can mean restricting the cases that you code has to deal with. For example, in Lab 8, simplification might mean testing with small matrices to make it easier to step through, or passing lists in which the initial rows correspond to the rows of the solution, and so on.

The other thing you can aim for when trying to simplify test cases is to simplify things such that you can mentally (or on paper) figure out exactly what your function will do during its execution. At this point, finding the bug becomes a matter of simply watching the execution of your function and determining when it diverges from your expectations.

You can save a lot of time by testing each function independently. If you write a bunch of code and try to debug it all at once, then you have a lot of places the bug can hide. If you exhaustively test each subcomponent as it is developed, then you can focus your testing on a new function and not on any previously tested functions that it calls.

Debugger

Learn to use the debugger, and get comfortable with breakpoints. You'll be using them a lot.

The basic idea is, set a breakpoint so you can get as close to the bug as possible, without going past it. After that, step, check that everything seems to be correct, and continue, until you run into the bug. You should have a pretty good idea as to what the correct behavior of your program is, so you can tell when your code diverges from it.

If you don't have the bug localized to a particular area, your best bet is to stick breakpoints at strategic areas in the code. Good places are the beginning and ends of functions - make sure that functions are being called with the correct arguments and returning the correct values. If your loops are not iterating too many times, it may be worthwhile to stick a breakpoint at the beginning of major loops - make sure that everything is as you expect it to be before continuing each iteration. If your loops are iterating too many times, you might be better off going for print statements (see the next section).

A key idea which is useful in debugging is "divide-and-conquer". In the context of debugging, this means that if you have a test case which produces an incorrect final result, set a breakpoint somewhere (hopefully close to halfway through the execution) and check at that point whether the state of the execution is correct. If it is, then it means the bug is after the breakpoint; if it isn't, then the bug is before the breakpoint. Either way you have reduced your remaining search space by half. Repeat by subdividing the buggy half with another breakpoint. Keep repeating until you find exactly where the bug manifests.

Finally, one trick to remember is the if-nop setup. If you've determined that the bug happens on the 1000th iteration of some loop, you can make setting up a breakpoint simpler, by sticking in a if (i == 1000) { nop; } or some assembly equivalent into the beginning of the loop. Then, just add a breakpoint on the nop operation in the if statement, and run until you hit the breakpoint. This can save a lot of frustration involved in walking through a loop multiple times.

Print Statements

If you don't have access to a debugger, or you don't have an easily reproducible test case, or you have a long stream of execution and very little idea of where the bug is, you may want to try to debug via print statements. Basically, modify your program to print out some status information every so often. Two things to watch out for: 1) you are changing the execution of the function - some bugs may disappear or be affected by this (mostly timing related bugs) and 2) make sure you do not insert new bugs when you code in the print statement. With assembly, make sure whatever registers you use to print things out do not hold important values, or at least do not hold values that you are going to use in the future.

The most obvious (and usually fairly useful) place to insert print statements is at the start of functions; that way, you can trace function calls. You might also want to place print statements at returns. The next obvious place is at the beginning of loops. Most loops should have a set of loop constraints - basically, a set of constraints that you expect to be true at the beginning of each loop.

Print statements can be used to verify these loop constraints and figure out exactly where they're violated, although there can be a lot of output to sift through.