CS/ECE 347, Spring 1996
Recitation Assignment 2: MIPS assembly language

Assigned: Tue., Jan. 23
Due: Mon., Jan. 29

1. Introduction

In this assignment, you will examine, annotate, and modify MIPS assembly code generated from a few simple functions written in C. The purpose of this assignment is to familiarize you with MIPS assembly language and with how gcc transforms C code into assembly code.

There are three problems. In problem 1, you will use gcc to transform a simple C function into MIPS assembly code, both optimized and nonoptimized; you will then add comments to the assembly code. You will also use gdb to examine the instruction codes in an executable containing code for the same function.

In Problem 2, you will examine and annotate MIPS code for another version of the same function, and constrast it with the code you looked at in Problem 1.

In Problem 3, you will examine and optimize assembly code generated from yet another simple C function, and then compare the performance of a hand-optimized version with that of a compiler-optimized version.

To get started, copy everything in

```
/afs/cs/academic/class/15347-s96/public/recitations/ra2/
```

to your personal directory.

The answers you hand in must be a typewritten or computer–generated document. The graph in problem 3 may be drawn by hand.

2. Problem 1

In the file `idxcopy.c`, you will find a single C function, `idxcopy`. This function duplicates the functionality of `strcpy`, which copies one null-terminated string of characters to another. The first argument is the destination array, and the second argument is the source.

Part A

First, use gcc to generate a MIPS assembly language version by running the following command on a workstation with a MIPS processor:

```
gcc -c -S idxcopy.c
```

This will produce a file, `idxcopy.s`, containing the relevant MIPS code.

For each instruction in the MIPS assembly code in `idxcopy.s`, add a short comment in the style of the annotations used in the slides from the second lecture (1/18/96). Make it clear that you understand which registers correspond to which variables in the code, and which source code lines correspond to which assembly language instructions.

Part B

Recompile `idxcopy.c` into *optimized* MIPS assembly code using the command

```
gcc -c -S -O2 idxcopy.c -o idxcopy.opt.s
```

This will produce another file, `idxcopy.opt.s`.

Annotate this file as you did in Part A. Comment *briefly* on some of the significant differences between this code and the unoptimized version you generated in Part A.

Part C

Use `make` with the supplied makefile to compile an optimized version of `idxcopytest`, a trivial program that uses idxcopy to copy a string before printing it. Now, you will use `gdb` to examine the assembled MIPS code.

Run `gdb` on idxcopytest. (The best way to do this is to run it from Emacs, which you can do by using the [Meta-X] gdb command.) Set a breakpoint inside the idxcopy function (by typing ``break idxcopy'') and start executing the program (by typing ``run''). Now, type ``dissassemble''. This will produce a disassembled "dump" of `idxcopy`'s code.

Turn in an annotation of the disassembled code for `idxcopy` similar to the annotations you generated in parts (A) and (B). Comment *briefly* on some of the differences between (1) the code shown in the dump you just produced and (2) the assembly code in `idxcopy.opt.s` (generated in Part B).

Some gdb commands that can help while disassembling code in the future:

```
disassemble
info reg
nexti
stepi
```

3. Problem 2

Part A

In the file `ptrcopy.c`, you will find another string-copying routine, `ptrcopy`. Compile this file into *optimized* MIPS assembly code as follows:

```
gcc -c -S -O2 ptrcopy.c -o ptrcopy.opt.s
```

Annotate the resulting file, `ptrcopy.opt.s`, as you annotated the assembly code in Parts A and B of Problem 1.

Part B

Use `make` with the provided makefile to compile an optimized version of `ptrcopytest`. Run `ptrcopytest` with `gdb`, and type ``disassemble ptrcopy'' to produce a dissambled version of ptrcopy.

Annotate the resulting dump, as you annotated the dump in Part C of Problem 1.

Part C

Suppose the optimized version of `idxcopy` is given a very long string to duplicate. How many instructions does the computer execute per character copied?

Now, suppose the optimized version of `ptrcopy` is given a very long string to duplicate. How many instructions does the computer execute per character copied? Is this any smaller or larger than the number of instructions per character executed by `idxcopy`? If so, *briefly* note what accounts for the difference.

4. Problem 3

You are on a team developing software to decode and analyze data packets from a space satellite. Unfortunately, one of the other software engineers who is no longer working for your group unwittingly used out-of-date documentation while writing all of his code. His C code is so obfuscated that you have no hope of deciphering and fixing it before the project deadline. Your mission, should you choose to remain gainfully employed, is to write a function that takes an array of data packet structures and tweaks the data in these structures so that it conforms to the up-to-date format specifications – and to make this function reasonably fast, since it's going to be called a lot.

Luckily, the transformation you need to perform on the packets are simple: you need only to swap the first two words in the packet, and multiply the eleventh word by ten.

In `packet.h`, you will find a declaration of the simple data structure being used to represent the packets; in `fixPackets.c`, you will find a correct but somewhat slow version of `fixPackets`, the function you need to implement.

Part A

Produce and annotate a *hand-optimized* MIPS assembly language version of fixPackets. If you want, you can try writing it from scratch, but it would probably be easier to use gcc to produce non-optimized MIPS code like this:

```
gcc -g -c -S prob3.c
```

and then change the resulting assembly code. You may modify the original C code as you see fit as long as it retains its original functionality. But don't use any -O options while compiling! Treat this exercise as a contest between you and your friendly neighborhood automatic optimizer.

Turn in your annotated hand-optimized assembly language version of this routine.

Part B

Now, go ahead and compile an optimized (-O2) version of `fixPacketsOriginal.c`, which contains a copy of the original `fixPackets` function renamed to `fixPacketsOriginal`.

Write a C program to test the relative speeds of your hand-optimized code and the compiler-optimized code. The program should be similar to the program you wrote for Problem 4 of the first recitation assignment: for a value of `n` specified on the command line, it should run both your hand-optimized function (`fixPackets`) and the compiler-optimized function (`fixedPacketsOriginal`) on arrays of size `n`; it should do so 10 times, and return the minimum, mean, and maximum times each of the functions took. Plot these times for both functions as a function of `n` for four values of `n` of your choosing and turn in the plot.