Assignment #1: Introduction to SUIF

Due: Before class, September 22

Introduction

In this assignment, you will compile a simple C program using SUIF and write a simple peephole optimization pass.

The main purpose of the assignment is to introduce you to the SUIF compiler, rather than to test your knowledge of compilation techniques. Your success in the later homeworks and possibly your project depends upon learning to use SUIF. As you will see, we request that you submit only a small amount of material.

Part 1

First, write a little C program that sorts its command line arguments (which should be integers) and prints the sorted list to standard out. It doesn't matter what kind of sort you use; keep it simple. Test this code using the GNU C compiler (`gcc`). Next look at
`http://www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15745-f03/www/suif.html`
and find the directions (near the bottom) for setting up your shell environment. Run `setup-suif-localtree` and add the `setup-suif-env` script to your shell initialization file, as described, to define the environment variables needed for use of SUIF.

Read the Machine SUIF Overview document to find out how to compile and run your program using SUIF. That document assumes that the reader knows how to run the base SUIF front end to prepare an input file written in C for processing by the Machine SUIF back end. But all you need to know is that the `c2s` command invokes the necessary passes to transform a `.c` file into a `.suif` file, which is a SUIF intermediate file. E.g.,

```
c2s hello.c
```

produces `hello.suif`. The next step in compilation is to use `do_lower`, which is discussed in the overview document. E.g.

```
do_lower hello.suif hello.lsf
```

produces another intermediate file that is the direct input to the first MachSUIF pass. Then you apply other MachSUIF passes until you have an assembly language (`.s`) file. (Part of your job in this assignment is to figure out which passes to run and in what order.) This you can assemble into an executable, e.g.

```
gcc -g -o hello hello.s
```

You will hand in your `hello.c` and the `hello.s` file produced by Machine SUIF. Also, please hand in a text file named `handin.txt` containing the ordered list of passes used to obtain the `hello.s` from `hello.c` and a short explanation of what each pass does and why you ordered them in such way.

Part 2

We have partially implemented a peephole optimization pass that replaces integer multiplies by left shift (for example, 4*x becomes x<<2). Since an integer multiply on a Pentium 4 is 14 cycles versus 1 cycle for a shift this is clearly a good thing to do. In fact, production compilers will perform more aggressive transformations using multiple shifts, adds, and subtracts.
The code for this pass is in the directory:

```
/afs/cs.cmu.edu/academic/class/15745-f03/public/assignments/1/
```

Copy the whole directory to your own `$LOCAL_BASE` directory. E.g., run the commands:

```
cp -a $coursedir/public/assignments/1 ~/localnci/assignments/
```

The following files are provided:

peep	*directory containing machsuif peep pass code you'll be modifying*
input*.in	*inputs for your peep pass*
output*.ref	*corresponding reference outputs*
bench.c	*highly synthetic micro-benchmark*

Now you can `cd` into the `peep` directory and use `gmake` to compile the pass. (Remember to customize your shell environment as described above).

We've provided several input files which are already in the required format (cfg form after the gen pass). You can use the `do_print` program to generate a text version of this SUIF binary file that looks almost like a valid x86 assembly file.

```
$ do_print input0.in
        # [target_lib: "x86"]
        .comm   arr, 400
        .file   "foo.c" # 2
        .text
        .align  2
        .align  2
        .globl  foo
        # .loc  2 0
foo:
        # .loc  2, 5
        movl    foo.x,$vr0
        imull   $0,$vr0
        movl    $vr0,%eax
        # [regs_defd: "sparse", 8, 12]
        ret
        # [instr_ret: s32]
        # [regs_used: "sparse", 0]
        # [incomplete_proc_exit]
```

Each input file has a corresponding output reference file which demonstrate what code the peep pass should generate when run on the input file. Once you've built it, you invoke the peep optimization pass as follows:

```
do_peep input0.in output0.out
```

The file `output0.out` is the optimized output file; you can view this file with `do_print`. Note that if you haven't built a local copy of `do_peep` the global version will not perform the optimizations.

What to do: Extend the `process_two_op` function in `peep.cpp` to handle all the provided input files. Obviously, your solution should work for all powers of two (not just those in the input files).

Compile and run the bench.c file with and without your peephole changes. Record the running times using the unix `time` utility and include them in the `handin.txt` file. Also include a brief explanation of why you get these results (bench.c doesn't contain any multiplies).

Extra Credit

Further extend `process_two_op` to handle divisions by powers of two. Note that this is not is simple as the multiplication case. It might be useful to look at `gcc`'s output.

Hand In

To hand in your solution create a tar ball (e.g.
`tar cvf handin.tar ~/localnci/assignments/1/tohandin/*`)
with all the required files and copy it to:

`/afs/cs.cmu.edu/academic/class/15745-f03/public/handin/<user_name>/<assignment #>`

you can copy as many tar balls as you wish. Only the last version will be graded.

The tar ball should contain the following files:

- hello.c
- hello.s
- handin.txt
- peep.cpp

Good luck!