This lab investigates problems confronting a compiler writer who wishes to minimize data hazards and use the branch delay slot in MIPS effectively.

1. Get a copy of the program `lab5.c` and put this in a subdirectory named `lab5` under `sde/examples`. Make a `Makefile` for this program. Note there is no assembly language module for this lab. Compile, link, and run the program, which basically tests mixing an ascii signal with some noise but using different chip encodings so that the signal may be extracted later in the main loop of `main()`. This computation is very much like what has to be done in the cell phone protocol CDMA.

**Assignment** Analyze the main loop which extracts the filtered signal in `main()` (just the loop – not the entire program, which is between the third call to `dump_octets` and the fourth call to `printf`). Try compiling the program with two different optimization settings: no optimization and optimization for speed of execution `-O2`, and then analyzing the code in assembly. You have two ways of doing this. You can use “sde-gcc -S” and look at the resulting assembly text file `lab5.s`. For example

```
sde-gcc -S lab5.c
sde-gcc -S -O2 lab5.c
```

This method shows more symbolic information but you will be annoyed that the mips registers are given by number (refer to `mips_regs.txt`).

Alternatively you can set optimization in the `Makefile` via `CFLAGS`, run “sde-make”, and then disassemble the executable, via

```
sde-objdump -d lab5ram > lab5ram.asm
```

This gives the symbolic names of the registers but leaves out other symbolic information.

In general, how effective is the compiler in applying optimization? Things you should check for include:

i. scheduling the branch delay slot.

ii. keeping local variables in registers.

iii. avoiding unnecessary reads and writes to and from the stack.

iv. saving intermediate results and not repeating any unnecessary computations.

v. using shifts instead of multiplication and division if possible.

vi. choice of instruction (e.g. `madd` instead of `mult`) to improve code performance.

Could you do better in assembly knowing what you know now about data and control hazards?