CS 61C Project 1: Disassembler Fall 1998

Administrative details

Submit a solution to this project by Noon, September 16. Homework assignment 4 is also due on this date.

The project should be done in partnerships of two or three students. (Partners should be in the same lab section.) Put your names and lab section times on everything you turn in. Your solution will be partly graded by computer; put a file of program source code named proj1.c or proj1.scm in a directory named proj1 and submit it as you did for earlier assignments. We are indeed allowing you to write a version in Scheme, although the rest of your programming for this course will be in C or MAL.

Project description

Complete a disassembler for MIPS machine language by filling in code in one of the framework files proj1framework.c, or for Scheme, proj1framework.scm. A *disassembler* reads a file containing *binary* MIPS instructions and prints a listing of the corresponding TAL instructions. Just as an assembler translates from assembly language to machine language, your disassembler will translate in the opposite direction. This is rather like what spim gives if you ask it to print the instructions in your program, except without symbolic labels.

Your program should handle all instructions listed in Appendix C of Goodman and Miller except for instructions with opcode 16 or 17 (base 10). This includes a number of instructions that you have not yet used in programs. Don't be too concerned; all you have to do is print the instruction, not understand how it works.

For memory accesses you should print something like

```
lw $8, 248($29)
```

with a signed offset and a base register as in the instruction. Immediate operands for instructions that sign-extend the operand should be printed in decimal; immediate operands for instructions that do not sign-extend should be printed in hexadecimal, preceded by "0x". Incorporate the program counter in the address you produce for a branch instruction; for example, given a binary file corresponding to the program

```
main:    bltz   $4,label1
         add    $8,$0,$0
         j      label2
label1:  ori    $8,1
label2:  ...
```

your output would look something like

```
00400000  04800002     bltz    $4, 0x0040000c
00400004  00004020     add     $8, $0, $0
00400008  08100004     j       0x00400010
0040000c  35080001     ori     $8, $8, 0x1
00400010                                ...
```

(Note: spim loads instructions starting at 0x00400000, not at zero! This matters because of branch offsets.) For an instruction not in the Goodman and Miller table, you should just print the message "unrecognized instruction".

The name of the file containing the binary machine language program should be provided as a command line argument. The program frameworks handle the processing of the command line arguments (including a debugging switch that you may wish to use) as well as input from the file.

The file ~cs61c/lib/spim.dump may be used to test your program. It contains binary machine instructions corresponding to the spim source file ~cs61c/lib/proj1test.s (but just the text part, not the data part)*. If you want to make your own test files, give spim the command dump; this will create a file called spim.dump in your working directory that corresponds to whatever source file has just been loaded. (The dump command is not available on the Mac or PC versions of spim.) Make sure, however, that the architecture on which you generate a spim.dump file is the same as the

* proj1test.s is a MAL program that contains assembler directives as well as pseudoinstructions. The assembler directives won't appear in the spim.dump file at all; the pseudoinstructions will appear as the instruction sequences to which they were translated by the assember.

one your disassembler is running on, since the byte order on the little-endian Intel computers is the reverse of that on the big-endian DEC and HP workstations.

You shouldn’t need to change the existing code in the framework files proj1framework.c or proj1framework.scm. These programs use the technique of data-directed programming that you learned in CS 61A (see chapter 2 of Abelson and Sussman). In C, this is done with function pointers.

Note: The decode table in figure A.19 is wrong in the 2nd printing of 2nd edition. The 3rd printing is OK. (If your book says `lb` is 32 decimal, it is correct)!.