CSCI 2321 (Computer Design), Spring 2021

Homework 4

Credit: 25 points.

1 Reading

Be sure you have read, or at least skimmed, all assigned readings from Chapter 2 and Appendix A.

2 Problems

Answer the following questions. You may write out your answers by hand and scan them, or you may use a word processor or other program, but please submit a PDF or plain text via e-mail to my TMail address. (No links to shared files on Google Drive please.) Please use a subject line that mentions the course and the assignment (e.g., "csci 2321 hw 4" or "computer design hw 4").

1. (5 points) Consider this fragment of MIPS code, intended to be a typical MIPS version of if/else:

```
        slt     $t0, $s1, $s2
        beq     $t0, $zero, Else
        addi    $s3, $s3, 1
        addi    $s4, $s4, 1
        j       After
    Else:
        addi    $s3, $s3, -1
        addi    $s4, $s4, -1
    After:
```

   Translate the `beq` and `j` instructions into machine language, assuming that the first instruction is at memory location `0x00400040`. (You don't have to translate the other instructions, just those two. As in Homework 2, first list all the fields (e.g., opcode) in binary and then give the 32-bit instruction in hexadecimal.)

2. (5 points) The MIPS assembler supports a number of *pseudoinstructions*, which look like regular instructions but which assemble into one or more other machine instructions. We've seen how SPIM assembles the `la` pseudoinstruction into a combination of `lui` and `ori`. As another example, pseudoinstruction `ble` generates two instructions, a `slt` and then a `bne`, using "assembly temporary" register `$at`, with

```
ble $t0, $t1, There
```

   being translated to

```
slt $at, $t1, $t0
beq $at, zero, There
```

If you wanted the assembler to support the following pseudoinstructions, say what code (using real instructions) the assembler should generate for the given examples. As with `ble`, you should use `$at` if you need an additional temporary register.

- `bnz` with the two operands (register number and target label/address), that branches to the target if the register contents are nonzero. Example:

  ```
  bnz $s0, There
  ```

- `swap` with two register-number operands, which exchanges the values in the two registers. Example:

  ```
  swap $s0, $s1
  ```

3. (15 points) (*Note:* This problem may initially look intimidating, but if you take it step by step I think you will find it manageable.)

   For this problem your mission is to reproduce by hand a little of what an assembler and linker would do with two fairly meaningless[1] pieces of MIPS assembly code. The textbook has an example starting on p. 132 illustrating more or less what I have in mind here, and we reviewed the example in class, but on reflection it doesn't seem that clear to me, so for this assignment I want you to approach the problem a little differently.

   First, the two files:

   - `procMain.s`.
   - `procSub.s`.

   I worked through a much longer example; results are in this directory. (There is also a `zip` file of the whole directory here.)

   (a) For the "assembly" phase, I don't want you to actually translate the instructions into machine language, but I do want you to construct for each file a human-readable version of the symbol table and relocation information that would be in the object file, with information as listed below.

   Note that you will need to expand the two `la` pseudoinstructions. The example in the textbook doesn't really show how to do this; they instead show how to deal with `lw` and `sw` referencing a symbol and assembled into something using the `$gp` register. How that works is a bit unclear, and SPIM is no help, but for this problem I'm going to bypass that whole can of worms and just ask you to work with `la`, which expands to `lui` followed by `ori`. (You can see examples of this by loading any of the sample programs that use `la` into SPIM and looking at what it shows for code.)

   (*Hint:* Before going further, you'll probably find it useful to produce annotated versions of each `.s` file, showing offsets of each instruction in the text segment and each variable in the data segment. Note that this means actual machine instructions, so you'll need to expand any pseudoinstructions.)

   (In the example, the two source code files are `file1.s` and `file2.s`, and the annotated versions are `file1-annotated.txt` and `file2-annotated.txt`.)

   Then produce, for each of the two source files, the following:

   (Use hexadecimal to represent addresses and offsets. *Don't panic* if you find hexadecimal hard to work with; the example directory contains a little Scala script and a snippet of

[1] They don't do anything very interesting, but together they do represent a complete program.

Scala you could use interactively, either of which should make it easier if still a bit tedious.)

- Text (code) and data sizes, in hexadecimal.
- "Relocation information": For each instruction that involves an absolute address (jumps involving an address, and the instructions corresponding to a `la` pseudoinstruction):
  - Its offset in the text segment.
  - The instruction type (as in the textbook example).
  - The symbol referenced ("dependency" in the textbook example).
- A symbol table listing all symbols, defined or not. For defined labels, give its location (which segment, offset) and whether it's global or local. For undefined labels, just say it's undefined.

(In the example, the results are in `file1-relocation.txt` and `file2-relocation.txt` and `file1-symbols.txt` and `file2-symbols.txt`.)

(A real assembler would probably try to resolve references to local symbols at this point, but for simplicity I want you to just resolve them all in the next step.)

(b) Next, "link" the two object files, for an executable with the text and data segments starting where SPIM puts them, at `0x00400024` and `0x10010000` respectively. (So absolute addresses into the two segments can be based on these values.) Specifically, show:

- Sizes of combined segments (text and data).
- "Patched" versions of instructions that couldn't be correctly and completely assembled at assembly time.

To generate the patched instructions, I recommend that you:

- Create a combined list of symbols (file `combined-sizes-and-symbols.txt` in the example).
- Create a combined list of instructions to patch, from the relocation information (file `instructions-to-patch.txt` in the example).
- Use those two lists to produce a list of patched instructions (file `patched.txt` in the example).

3 Pledge

For programming assignments, this section should go in the body of the e-mail or in a plain-text file `pledge.txt` (no word-processor files please). For written assignments, please put it in the text or PDF file with your answers.

Include the Honor Code pledge or just the word "pledged", *plus* at least one of the following about collaboration and help (as many as apply). Text *in italics* is explanatory or something for you to fill in.

- I did not get outside help *aside from course materials, including starter code, readings, sample programs, the instructor.*
- I worked with *names of other students* on this assignment.
- I got help with this assignment from *source of help — ACM tutoring, another student in the course, etc. (Here, "help" means significant help, beyond a little assistance with tools or compiler errors.)*

- I got help from *outside source — a book other than the textbook (give title and author), a Web site (give its URL), etc.. (Here too, you only need to mention significant help — you don't need to tell me that you looked up an error message on the Web, but if you found an algorithm or a code sketch, tell me about that.)*

- I provided help to *names of students* on this assignment. *(And here too, you only need to tell me about significant help.)*

4 Essay

For programming assignments, this section should go in the body of the e-mail or in a plain-text file `pledge.txt` (no word-processor files please). For written assignments, please put it in the text or PDF file with your answers.

Include a brief essay (a sentence or two is fine, though you can write as much as you like) telling me what if anything you think you learned from the assignment, and what if anything you found interesting, difficult, or otherwise noteworthy.