CSCI 2321 (Computer Design), Spring 2018

Homework 3

Credit: 50 points.

1 Reading

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

2 Honor Code Statement

Please include with each part of the assignment the Honor Code pledge or just the word "pledged", plus one or more of the following about collaboration and help (as many as apply).[1] Text *in italics* is explanatory or something for you to fill in. For written assignments, it should go right after your name and the assignment number; for programming assignments, it should go in comments at the start of your program(s).

- This assignment is entirely my own work. *(Here, "entirely my own work" means that it's your own work except for anything you got from the assignment itself — some programming assignments include "starter code", for example — or from the course Web site. In particular, for programming assignments you can copy freely from anything on the "sample programs page".)*
- 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, the instructor, 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.)*

3 Problems

Answer the following questions. You may write out your answers by hand or using a word processor or other program, but please submit hard copy, either in class or in one of my mailboxes (outside my office or in the ASO).

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

[1] Credit where credit is due: I based the wording of this list on a posting to a SIGCSE mailing list. SIGCSE is the ACM's Special Interest Group on CS Education.

```
        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, $1, There
   ```

   being translated to

   ```
   slt $at, $t1, $t0
   bne $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) For this problem your mission is to reproduce by hand a little of what an assembler and linker would do with two fairly meaningless[2] pieces of MIPS assembly code. The textbook has an example starting on p. 127 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, one containing a main procedure:

   ```
   .text
   .globl main
   ```

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

```
main:
        addi    $sp, $sp, -4
        sw      $ra, 0($sp)
        jal     subpgm
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        jr      $ra
        .end    main

        .data
        .globl dataX
local:  .word   0
dataX:  .word   1, 2
```

and another a procedure it calls:

```
        .text
        .globl subpgm
subpgm:
        addi    $sp, $sp, -4
        sw      $ra, 0($sp)
# copy data (two "words") from dataX to dataY
        la      $s0, dataX
        la      $s1, dataY
        lw      $t0, 0($s0)
        sw      $t0, 0($s1)
        lw      $t0, 4($s0)
        sw      $t0, 4($s1)
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        jr      $ra
        .end    subpgm

        .data
        .globl dataY
dataY:  .space  8
```

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 table 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.[3] Instead I want you to expand these instructions in the way SPIM does: each as a `lui` followed by a `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 write down, for each of the two files, what's in its text segment (a list of instructions and their offsets, remembering to expand

[3] I'm not quite sure how they get this from MIPS assembly source; SPIM will accept load/store instructions referencing a label, but it turns them into `lui`/`ori` pairs in the same way it does for `la`.

any pseudoinstructions), and what's in its data segment (a list of variables/labels and their offsets and sizes).)

Then produce, for each of the two source files, a table with the following. (Use hexadecimal to represent addresses and offsets.)

- Text (code) and data sizes, in hexadecimal.
- "Relocation information": For each instruction that involves an absolute address (jumps 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, showing for each:
  - Its name.
  - Which segment it's in (text or data) and its offset into that segment.

  For example. the first symbol in the first file is `main`, at offset 0 into the text segment.

(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.)

Next, "link" these two files to produce information for an executable *for the SPIM simulator*. Since programs in this simulator always have their text segments at `0x00400024` and their data segments at `0x10010000`, absolute addresses into either segment can be based on these values. (Normally an executable file might include "relocation information" for any instructions containing absolute addresses that would need to be changed when the program is loaded into memory, but we'll skip that.)

(*Hint:* Notice that the text segment of the executable is just the text segment for the first file followed by the one for the second file, and similarly for the data segment. So you'll probably find it useful to come up with a list of what's in each segment, similar to what you did in the first step, but with addresses rather than offsets.)

The information I want is this:

- Text (code) and data size, in hexadecimal.
- A symbol table showing locations of *all* symbols and their addresses (e.g., `main` is at `0x00400024`). (Really I think this should just be the global symbols, but to patch the unresolved references you'll need some non-global labels, and this is the simplest way to achieve that.)
- Patched versions of the instructions from the object files' "relocation information" sections, in the form of another table, one entry per instruction, with:
  - The instruction's address (or you can specify offset into combined text segment — that's what I asked for originally, but address is both better and easier).
  - The patched instruction, in a form that looks like source code but doesn't reference labels — so for example a `j main` would become `j 0x00400024`. (Use hexadecimal for the constant/immediate values here.)

4 Programming Problems

Do the following programming problems. You will end up with at least one code file per problem. Submit your program source (and any other needed files) by sending mail to bmassing@cs.trinity.edu with each file as an attachment. Please use a subject line that mentions the course and the assignment (e.g., "csci 2321 hw 3" or "computer design hw 3"). You can develop your programs on any system that provides the needed functionality, but I will test them on one of the department's Linux machines, so you should probably make sure they work in that environment before turning them in.

1. (10 points) Add code to your solution[4] to textbook problem 2.27 from Homework 2 to make it a complete program that, run from SPIM, prompts for values for `a` and `b` and prints the ending values of the elements of `D`. I.e., for output the program should do the equivalent of the C code

   ```
   for (k=0; k<4*b; k++)
       printf("%d\n", D[k]);
   ```

   Programs `echo.s` and `echoint.s` on the sample programs page show how to input and output text and integer values.

   (I think this is also a good opportunity to tweak your solution so it uses registers `$s3` and `$s4` (rather than `$t0` and `$t1`) for variables `i` and `j`, but if you can make the program work without doing that, okay.)

2. (15 points) Problem 2.31 from the textbook asks you to write a MIPS implementation of a recursive function `fib` to compute elements of the Fibonacci sequence. For this problem, your mission is to write this MIPS function and incorporate it into a complete program that, run from SPIM, prompts for an integer value N, calls `fib` to compute the N-th element of the sequence, and prints the result. Program `factorial.s` on the sample programs page may be helpful, since it shows how to do the needed input/output and also contains an example of a recursive procedure written in MIPS. *To get full credit, your program must use recursion, and any functions/procedures you define must follow the conventions described in the textbook and in class for passing arguments and saving/restoring registers.*

[4] Or you can start from my sample solution, shared with you via Google Drive.