CSCI 2321 (Computer Design), Spring 2018

Homework X

Credit: Up to 50 extra-credit points.

1 Overview

You can do as many as you like, but you can only receive a total of 50 extra points.

NOTE that the usual rules for collaboration do not apply to this assignment. More in the following section.

2 Honor Code Statement

Please include with each part of the assignment the Honor Code pledge or just the word "pledged", plus one of the following statements, whichever applies:

- "This assignment is entirely my own work".
- "This assignment is entirely my own work, except that I also consulted *outside course — a book other than the textbook (give title and author), a Web site (give its URL), etc.*."

(As before, "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.)

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. (Optional: Up to 10 extra-credit points.) For this problem your mission is to reproduce by hand a little of what an assembler and linker would do, as you did in the last problem of Homework 3. So you are to do two phases:
   - Assembly, in which you produce for each input file the following:
     - Sizes of text (code) and data segments, in hexadecimal. (*Correction/clarification:* Remember that `la` is a pseudoinstruction that expands to a combination of `lui` and `ori`. `li` is also a pseudoinstruction that potentially expands to the same pair, but it looks like SPIM's built-in assembler produces both instructions only if the immediate value being loaded is more than 16 bits; for small values it expands it just to an `ori`. You should do likewise. (So, as long as the value being loaded is "small enough", the assembler expands `li` to just `ori`.)
     - "Relocation information", as in Homework 3 — a list/table with one entry for each instruction that needs to be patched based on where in memory the program needs to be loaded.

  - A symbol table with entries for all symbols, showing for each its name, which segment it's in (text or data), and its offset into that segment.

- Linking, in which you produce:
  - Sizes of combined text (code) and data segments, in hexadecimal.
  - A symbol table showing locations of all symbols and their addresses.
  - Patched versions of all the instructions from all the "relocation information" segments from the assembly phase, in the form described in Homework 3.

The input files are these:

- `main.s`:

```
        .text
        .globl  main
main:
# opening linkage
        addi    $sp, $sp, -4
        sw      $ra, 0($sp)
# prompt and get two integers from "console"
        la      $a0, prompt
        li      $v0, 4          # "print string" syscall
        syscall
        li      $v0, 5          # "read int" syscall
        syscall
        la      $t0, dataX
        sw      $v0, 0($t0)     # save result in dataX
        li      $v0, 5          # "read int" syscall
        syscall
        la      $t0, dataY
        sw      $v0, 0($t0)     # save result in dataY
# call procedure to add and print
        la      $a0, dataX
        la      $a1, dataY
        jal     foobar
# closing linkage
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        jr      $ra
        .end    main
# variables and constants
        .data
prompt: .asciiz "Enter two integers, one per line:\n"
# note that .word forces alignment -- i.e., causes assembler to insert
#   space if not on a word boundary (address a multiple of 4)
dataX:  .word 0
dataY:  .word 0
```

- `foobar.s`:

```
        .text
```

```
        .globl  foobar
foobar:
# add two integers and print result
# $a0, $a1 have addresses of two integers
# opening linkage
        addi    $sp, $sp, -4
        sw      $ra, 0($sp)
# compute result into $s0
        lw      $t0, 0($a0)
        lw      $t1, 0($a1)
        add     $s0, $t0, $t1
# print
        addi    $a0, $s0, 0
        li      $v0, 1          # "print int" syscall
        syscall
        la      $a0, foobar_nl
        li      $v0, 4          # "print string" syscall
        syscall
# closing linkage
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4
        jr      $ra
# variables and constants
        .data
foobar_nl: .asciiz "\n"
```

and for the link step you should assume:

- Object code for `main.s` is loaded first, then object code for `foobar.s`.
- The combined text segment starts at `0x04000000`.
- The combined data segment starts at `0x10000000`.

1. (Optional: Up to 5 extra-credit points.) One of the questions on Exam 2 asks you about additions to the table of ALU control signals in Figure B.5.13 of the textbook: Each line in the table represents a combination of control inputs (`Ainvert`, `Bnegate`, and a 2-bit `Operation`) to the design shown in Figures B.5.10 and B.5.12. The table doesn't include all 16 possibilities for these inputs, perhaps because some of them don't correspond to actual MIPS instructions. What operation would a line for values `1011` represent? (*Hint:* It may be helpful to review how values `0111` cause the circuit in B.5.12 to compute `slt` on the two inputs.)

2. (Optional: Up to 10 extra-credit points each.) One of the homeworks asked you to describe what changes would be needed to the single-cycle implementation sketched in Figure 4.24 of the textbook to allow it to execute additional instructions. For each of the instructions below, describe what would be needed in order to support it. Specifically:

   - Would you need additional control signals? If so, give their names and their values for all of the instructions executed by the design in Figure 4.24 and the instruction you're adding support for.
   - What values would be needed for all of the existing control signals for the added instruction?

- Would you need to make changes or additions to the design (additional combinational logic blocks and/or "wires" connecting things)? If so, describe them in enough detail that another student in this class could turn them into additions to the diagram. It may be simplest and clearest to just print or photocopy the diagram and mark it up, though if what's needed can be clearly described in words or with a smaller sketch, you can do that instead.

The instructions:

- `jr`
- `jal`
- A hypothetical new instruction `throw` inspired(?) by the discussion in Section 4.9 of the textbook of changes to the pipelined implementation needed to support exceptions. The instruction makes use of two new state elements, `Cause` and `EPC`, and works as follows: If *register* is a register designation (e.g., `$s0`),

  `throw` *register*

  places the value in *register* in `Cause`, places the value of the incremented program counter in `EPC`, and makes the new value of the program counter `0x80000180`.

(You can do any or all these.)

3. (Optional: No maximum, though as a rough guideline a page or so of prose will likely get you about 5 points.)

   In this course we focused on the MIPS architecture and its assembly language because it's simple and regular, and in theory once you have this background you should be well-prepared to learn about other architectures and their assembly languages. Choose some other architecture (x86 comes to mind, but there are others) and write a one-page-or-so executive-level summary of how it compares to the MIPS architecture (e.g., does it also have a notion of general-purpose registers, what if any special-purposes registers does it have, how do (some of) the instructions compare to those used in MIPS, etc.). Include a list of the sources you consulted (parts of the textbook, Web sites, etc.) You can even do this more than once for several different architectures.

4. (Optional: No maximum, though as a rough guideline a page or so of prose will likely get you about 5 points.)

   For testing MIPS assembler programs we used a simple emulator (SPIM). Based on a very quick Google search it appears that there are other tools that provide similar or greater functionality (cross-compilers that generate MIPS assembler or object code from C code. full-fledged virtual machines that implement the MIPS architecture.) Find one or more that seem to you likely to be useful for this course and explain why you think it would be useful and what would be involved in installing it.

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 X" or "computer design hw X"). 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. (Optional: Up to 10 extra-credit points.) For this problem, you are to write a MIPS procedure that, given a (null-terminated) string, tries to convert it to a signed integer and reports success/failure. More explicitly, this procedure should get the address of the string as the first argument (in `$a0`) and produce two results:

   - An "error code" in `$v1`), where 0 means success, -1 means the string doesn't represent a signed integer, and -2 means the conversion wasn't possible because it would cause overflow.
   - If there was no error, `$v0` should contain the result of the conversion.

   So "10", "-20", and "2147483647" ($2^{31} - 1$) are all valid, but "10-", "abcd", "10ab", and "2147483648" (2^{31}) are not. To get maximum points you need to detect both kinds of errors, but you can get up to 8 points if you do everything except the check for overflow. Other "corner cases" include the empty string and "-", both of which should produce an error result (-1), but here too if you don't make that work you won't lose many points.

   Starter program http://www.cs.trinity.edu/~bmassing/Classes/CS2321_2018spring/Homeworks/HW0X contains code to prompt the user for a text string, read it, call the convert procedure, and print the results. Your mission is to fill in the body of the convert procedure so it works as described.

   Sample executions:

```
% spim  -f test-convert-int.s
Loaded: /usr/share/spim/exceptions.s
Enter a line of text:
10
Input 10
Result 10

% spim  -f test-convert-int.s
Loaded: /usr/share/spim/exceptions.s
Enter a line of text:
-20
Input -20
Result -20

% spim  -f test-convert-int.s
Loaded: /usr/share/spim/exceptions.s
Enter a line of text:
abcd
Input abcd
Error -1

% spim  -f test-convert-int.s
Loaded: /usr/share/spim/exceptions.s
Enter a line of text:
```

```
1000000000000
Input 1000000000000
Error -2
```

HINTS:

- Note that SPIM seems happy to accept character literals in the same format as C, so for example you can put the ASCII characters for 0 in a register by writing

  ```
  li $t0, '0'
  ```

  and the same thing works for other characters, such as the null character:

  ```
  li $t0, '\0'
  ```

  I strongly advise that you do this rather than looking up ASCII values and putting them in your code: MIPS assembly code is hard enough to read already, and using the ASCII values directly just makes it worse.

- Think about the algorithm first, but if nothing occurs to you, see this footnote[1].

2. (Optional: Up to 10 extra-credit points.) For this problem, you are to write a MIPS procedure that, given a memory address `p` and a number of bytes `n`, prints hexadecimal representations of `n` bytes starting at `p`. So for example if the `p` points to a "ab" and `n` is 2, the procedure should print "61 62" (hexadecimal representations of ASCII values for 'a' and 'b'), while if `p` points to an integer (in memory) with value 5 and `n` is 4, the procedure should print "05 00 00 00" (why is the 5 first? SPIM is little-endian, so bytes in integer types are stored in reverse order). More explicitly, this procedure should get `p` as the first argument (in `$a0`) and `n` as the second argument (in `$a1`) and print (to the "console", using SPIM system calls) as described. It doesn't need to return anything in `$v0` and `$v1`.

   Starter program http://www.cs.trinity.edu/~bmassing/Classes/CS2321_2018spring/Homeworks/HW0X contains code to prompt the user for a text string, read it, call the procedure to print the whole buffer, and then prompt for an integer, read it, and call the procedure to print the 4-byte result. Your mission is to fill in the body of the print procedure so it works as described. Sample execution:

   ```
   % spim -f test-print-hexbytes.s
   Loaded: /usr/share/spim/exceptions.s
   Enter a line of text:
   abcd
   Input abcd
   Result 61 62 63 64 0a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   Enter an integer:
   20
   ```

[1] You could do it in C thus, assuming `p` starts out pointing to the beginning of the string (note that this doesn't do any error checking, but you can figure that out?

```
/* put result of conversion in "work", ignoring errors */
int work = 0;
while (*p != '\0') {
    work = work*10 + (*p - '0');
    ++p;
}
```

```
Input 20
Result 14 00 00 00
```

HINTS:

- You will probably want to use the `lb` instruction ("load byte") to work with individual bytes.
- One way to do the conversion is to split the resulting byte into two half-bytes (each representing one hex digit) and then use those as indices into a string containing all the hex digits (`"0123456789abcdef"`).
- SPIM has a "print character" system call that you wil probably find useful.

1. (Optional: Up to 30 extra-credit points each.) Some of the homeworks and exams had you do things that (should?) seem very automatable. For any or all of the following tasks, write a program in a high-level language to perform it. You can use any high-level language I can easily test from the command line on one of our classroom/lab Linux systems. (For many of you Scala is likely to be your first choice, though C++ might appeal to some, or possibly Python.) Your program must include comments explaining what it does and its limitations (e.g., "only works for the following list of instructions"), a brief explanation of how to use it, and an example of suitable input. (Some of these are pretty ambitious but all seem interesting?)

   - Converting a line or lines of MIPS assembler to a text form of its binary representation. Such a program could range from fairly simple (only a subset of instructions, limited or no support for labels) to fairly complex (the equivalent of a full assembler).

     Credit will depend on how much your program does; a simple program that just works for a subset of instructions (including at least one R-format instruction, `lw` and `sw`, `beq` and `bne`, and `j`) would be worth 10 points.
   - Converting MIPS machine-language instructions to (somewhat) human-readable form. Such a program would take one or more machine-language instructions (text strings representing either 32-bit strings of 0s and 1s or 8-digit hexadecimal numbers) and display the operation and the operands as a line of MIPS assembly source code. For register operands, you can just give them as, e.g., `$8`, rather than looking up a symbolic name such as `$t0`. For absolute addresses you can show them as hexadecimal constants, e.g., `0x04004000`. Branch targets are tricky, but you could do more or less what SPIM does, which is to show the byte offset from the updated program counter (i.e., the "immediate" value from the instruction times 4).

     Here too credit will depend on how much your program does; you could make it work only for a subset of the possible opcodes (probably sensible). For this one, since the task is simpler, a program that handles a representative subset of instructions would be worth 10 points, but even one that handles all instructions would be worth at most 15 points.
   - Performing the tasks involved in those "pretend to assemble and link" problems, such as the one in Homework 3: Given one or more MIPS source files, generate for each source file text and data sizes, a symbol table, and relocation information, and then produce combined text and data sizes, a combined symbol table, and patched instructions.

     Here too credit will depend on how much your program does; a program that only deals with a subset of the available instructions and pseudoinstructions (the ones in the homework problem) would be worth 10 points.