CSCI 2321 (Computer Design), Spring 2019
Homework X

Credit: Up to 50 extra-credit points.

1 General Instructions

You can do as many of the following problems as you like, but you can only receive a total of 50 extra points. (How these figure into your grade: My usual grading scheme is based on adding up your points and dividing by perfect-score points. Extra-credit point are added to the "your points" number without changing the "perfect score" number. So the only thing you can lose by attempting extra problems is the time you spend on them.)

NOTE that the usual rules for collaboration do not apply to this assignment. Please work individually, without discussing problems with other students.

2 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 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.) Homework 8 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:

   - What processing is needed to perform the instruction? (Give a numbered list of steps, similar to the ones the textbook gives for the instructions it discusses.)
   - 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.

3 Programming Problems

Do as many of the following programming problems as you like. 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.

MIPS Assembly

1. (Up to 10 extra-credit points.) Write a MIPS procedure that, given a (null-terminated) string, tries to convert it to a signed 32-bit 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 `$v1`: 0 means success, -1 means the string doesn't represent a signed integer, and -2 means the conversion wasn't possible because it would cause overflow.
   - If no errors, the result of the conversion in `$v0`.

So "10", "-20", and "2147483647" ($2^{31} - 1$) are all valid, but "10-", "abcd", "10ab", and "2147483648" (2^{31}) are not. Other "corner cases" include the empty string and "-", both of which should produce an error result (-1). To get maximum points you must recognize all valid integers, including negative ones, and detect both kinds of errors, but you will get some points for anything that solves part of the problem.

Starter program `test-convert-int.s` 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:

- You will probably want to use the `lb` instruction ("load byte") to work with individual bytes. Sample program `palindrome.s` shows using these instructions in a program to check whether a line of text is a palindrome.
- 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. (Up to 10 extra-credit points.) Write a MIPS procedure that, given a memory address `p` and a number of bytes `n`, prints hexadecimal values 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" (ASCII values for 'a' and 'b', in hexadecimal), while if `p` points to an integer (in memory) with value 10 (`0xa`) and `n` is 4, the procedure should print "0a 00 00 00" (why is the a 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 `test-print-hexbytes.s` 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
Input 20
Result 14 00 00 00
```

   (Note that the starter code prints all the bytes of the buffer containing the text input line, hence all those zeros.)

   HINTS:

   - You will probably want to use the `lb` instruction ("load byte") to work with individual bytes. Sample program `palindrome.s` shows using these instructions in a program to check whether a line of text is a palindrome.
   - 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"`).

[1] You could do it in C thus (without error checking), assuming `p` starts out pointing to the beginning of the string:

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

- SPIM has a "print character" system call that you will probably find useful.

3. (Up to 10 extra-credit points.) An example I often use in teaching parallel programming estimates the value of π by approximating the area under the curve

$$f(x) = 4.0/(1 + x^2)$$

between $x = 0$ and $x = 1$, using a technique called numerical integration; Split up the whole range into N "slices" and for each slice compute the area of the rectangle with a width of $1/N$ and a height equal to the value of $f(x)$ at the midpoint of the slice. Program `num-int-pi.c` shows how to do this in C.

Your mission for this problem is to produce a similar program in MIPS for SPIM. Like the C program, it should prompt for the number of slices and print that and the estimated value of π. You don't have to include code to check that what the user entered was sensible (since checking for non-numeric input is a fair amount of trouble). Sample program `newton.s` is an example of doing calculations using the MIPS floating-point instructions.

Other

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?) You will get maximum points if your program does everything described *and* deals reasonably gracefully with bad input. You will get some points for programs that provide any of the functionality described.

1. (Up to 30 extra-credit points.) Write a program to convert 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.

2. (Up to 15 extra-credit points.) Write a program to convert 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` (although you'll get a few more points if you do look up the symbolic name). 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, a program that handles

a representative subset of instructions (say `add`, `sub`, `addi`, `lw`, `beq`, and `j`) would be worth 10 points.

3. (Up to 20 extra-credit points.) Write a program that performs the tasks involved in those "pretend to assemble and link" problems, such as the one in Homework 4: 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.

4. (Up to 20 extra-credit points.) Write a program that traces through operation of the processor design shown in Figure 4.24 of the textbook, as you did by hand in Homework 7. Such a program should accept as inputs the current value of the PC, the instruction found at that address, and any needed values for registers and data memory; it should produce output like what you were asked to produce for the homework. You can decide what input should look like, but explain in comments what you require and give at least one example. A format that occurs to me:

   - Program counter (in hexadecimal).
   - Instruction (in hexadecimal).
   - Zero or more lines giving register values, in the form REG n = *value* (*value* in hexadecimal).
   - Zero or more lines giving values in data memory, in the form DM *address* = *contents* (*address* and *contents* are probably best expressed in hexadecimal, since that's more natural for addresses).

   Output should be all the information requested for the problem(s) on Homework 7.

   Credit here will largely be based on how closely your program really simulates operation of the circuit (e.g., basing what happens on values of state elements and control signals rather than what you think should happen for particular opcodes). However, you don't have to simulate the internals of one of the state elements or logic blocks at the level of AND/OR gates; for example, you could represent the ALU as a function that takes a 4-bit value for "ALU control" and two 32-bit inputs and produces a 32-bit result and a 1-bit "zero" result. You can also represent 1-bit values as Booleans (e.g., a Scala `Boolean`).

4 Honor Code Statement

Include in each part of the assignment (written and programming problems) 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.)

5 Essay

Include a brief essay (a sentence or two is fine, though you can write as much as you like) telling me what about the assignment you found interesting, difficult, or otherwise noteworthy. For programming assignments, it should go in the body of the e-mail or in a plain-text file `essay.txt` (no word-processor files please).