CSCI 2321 (Computer Design), Spring 2017

Homework X2

Credit: Up to 50 extra-credit points. (But be advised that you can't get more than 50 extra-credit points on this assignment and Homework X1 combined.)

1 Overview

This set of extra-credit problems covers mostly material from the second part of the course, though there are some problems related to earlier material as well. You can do as many as you like, but you can only receive a total of 50 extra points on this assignment and Homework X1 combined.

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", and the statement "This assignment is entirely my own work" (where "my own work" means "except for anything I got from the assignment itself, such as starter code, or from the course Web site or sample solutions to other assignments). For this assignment you should *not* work with or seek help from other students or from tutors, but you can consult other sources (other books, Web sites, etc.), as long as you identify them.

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 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 (as was the case for the `bne` in Homework 6), 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 X2" or "computer design hw X2"). 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 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, or vice versa. 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 X1: 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.

   - Simulating operation of the processor design shown in Figure 4.24 of the textbook. Such a program should accept as inputs current values of the PC and all the registers and values of relevant parts of the instruction and data memories and should as closely as possible simulate what the circuit actually does — fetching the instruction from the instruction memory, generating the control signals, and using these signals to control the rest of the circuit. To simplify input processing, you can require that:

- All input values are given as hexadecimal values.
- The program counter (PC) has a value between `0x0` and `0xc` inclusive. (So it's enough to give values for only 4 words of the instruction memory.)
- Addresses to be used in `lw` and `sw` have values between `0x10000000` and `0x1000000c` inclusive. (So it's enough to give values for only 4 words of the data memory.)

Sample input for a representative `lw`:

`lw $t1, 4($t0)`

(text *in italics* is explanatory):

PC:
00000004
Instruction memory (4 words):
00000000
8e080004
00000000
00000000
Register contents (32 values):
.... *8 lines of all 0*
00000005 *$t0*
.... *23 lines of all 0*
Data memory (4 words, starting at address `0x1000000` *)*:
00000000
00000008 *value to load*
00000000
00000000

Output should be all the information requested for the first problem on Homework 6, plus the new contents of the PC, all the registers, and the four words of data memory that are allowed to be used.

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.