Asg Mon, Feb. 9, versions 1-6 required (100), 7-8 extra credit (20). Finish and PUSH YOUR WORK: Due Tue, Feb. 17, 11:59PM.

Learning Goals and Assignment

How to achieve the goals is detailed after we state them.

1. Understand the MIPS instruction set and the task of the CPU's control unit by simulating part that task by a C program. Simulate the task of figuring out what the data-path (ALU plus register file) must do **in response to** the current machine instruction (expressed in binary) currently being executed by the CPU.
2. Use C functions and a statically allocated array of structs table in a software design.
3. Practice coding bitwise mask and shift operations in C to decode MIPS instruction codes and fields.
4. Continue to follow practices of testing while developing, and maintaining your revision history (with GIT) for review and credit. Make GIT track a new file by commanding `git add` *name of new file.*
5. Learn and practice how Unix shell output redirection with `>` makes it easy to save a program's output in a file. Also use input redirection `<` to supply a program's input from a file, instead of typing it by hand each time.
6. Systematic testing and incremental development **save time in the end** because they reduce the risk of time-wasting bug finding.
7. Always use `gcc -Wall ...` so `gcc` prints **all warnings** and you can use them to track down mistakes.

Examine NOW and, when you are immersed in the coding, refer to Figure 2.20 (5th ed.), "MIPS instruction formats".

You will begin to demonstrate detailed understanding of what the instructions make the MIPS computer do. That comes from Chapter 2 and your experience with assembly and machine language. The content specific for this project begins in Section 2.5 (5th ed.) "Representing Instructions in the Computer".

A handy reference is the last five pages of Section 2.10 (5th ed.) "MIPS addressing ..." where instruction formats are reviewed, an example of decoding is given, and the **full MIPS instruction coding table** is printed.

Your loyal intern T. T. T. has tediously translated and typed the MIPS instruction coding table into a C array initializer. You will get that work **merged into** your GIT repository when you go into your `COProjects` directory and command:

```
git pull git404@git404.cs.albany.edu:COProjects
```

Next, go into directory `DecodeControl` (Use the `cd` command.)

Your `DecodeControl` programming task is to mimic or **simulate** in C how the CPU hardware **control unit** works. It channels bits through electric circuits to **control** which computation *other* hardware like the **Arithmetic Logic Unit** and the **Register File** will do [1].

[1](This project will prepare you for when we get fully into decoding the instruction codes with digital hardware as in Figure 4.14 (5th ed.), "The three instruction classes (R-type, load and store, and branch) use different instruction formats. ...", and Figure 4.15, "The datapath of Figure 4.11 with all necessary multiplexors ... " You might even now **correlate the 5 notations in Fig. 4.15** like "Instruction [25-21]" for bit-fields defined by bit positions **with their diagrams in Figs. 2.20 and 4.14**.)

MIPS Fields

MIPS fields are given names to make them easier to discuss:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
	Instruction [25:21]	Instruction [20:16]	Instruction [15:11]		Instruction [5:0]

or *R-format*. A second type of instruction format is called *I-type* (for immediate) or *I-format* and is used by the immediate and data transfer instructions. The fields of I-format are

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits
	Instruction [25:21]	Instruction [20:16]	Instruction [15:0]

The 16-bit address means a load word instruction can load any word within a

How the rs field is used by the hardware

How the rt field is used

How the funct field is used

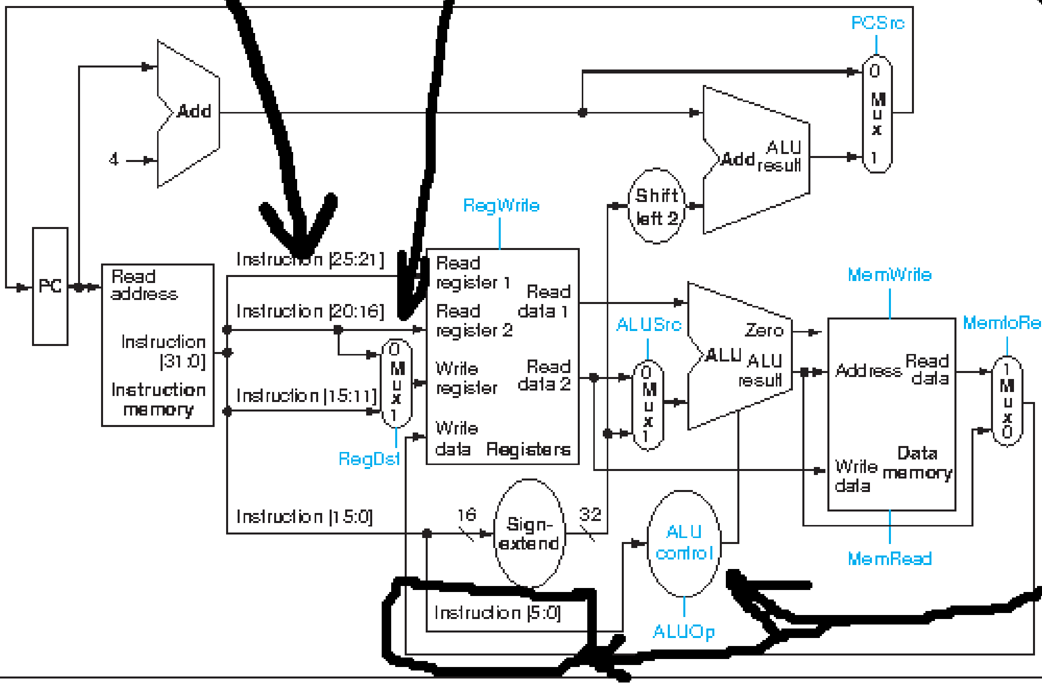

FIGURE 4.15 The datapath of Figure 4.11 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

You should do AT LEAST 8 commits. Here we go:

(This is mostly independent of Programming Project 2, but you must demonstrate using what was covered there about C functions, parameters, the address and dereference operators `&` and `*`, and bitwise and shift operators.)

Functionality required in each version is cumulative! Make perfect the versions IN THE ORDER ASSIGNED to maximize your partial credit

1 Version to fix the on-purpose table mistake

Compile and run Prof. Chaiken's `PrintOpCodes.c` tester program. Prove that you did that by commanding:
`script before.txt`
`gcc -Wall PrintOpCodes.c` *So it will show all Warnings, good!*
`./a.out`
`exit` (which saves the printout in `before.txt`)
`git add before.txt`
New GIT lesson: To make GIT track a file you just created or to make sure its current contents will be part of the next commit, command:
`git add` ***path to the file***

We purposely put a typical mistake in the code you just compiled! Both the message printed when the assertion "fired" will suggest where you need to look for the mistake. It's great to include assertions and examine compile-time warnings to get clues about mistakes!

Fix the mistake, recompile and check the table against the book. For credit, make yourself a MIPS table file and get it added to your next commit:
`gcc -Wall PrintOpCodes.c`
`./a.out > myMIPSTable.txt`
`less myMIPSTable.txt` (remember to use `q` to exit `less`)
`git add myMIPSTable.txt`
`git commit -a -m "`*MESSAGE*`"`
(You MUST create your a message that tells you and your TA what **every** commit is up to.)

From now on, a separate commit is required after completing **and testing** each version, but we will stop reminding you of that.

2 Version where the framework actually takes input

1. Compile and run our framework code with `gcc -Wall DecodeControlProj.c`
2. Note that this version of the code does not accept user input. Fix it, so that it does. The warning printed by `gcc` will give you a clue! (When you fix it, the program run will sometimes temporarilly stop, waiting for you to provide input. Operating system experts say "the process or thread blocks for input.")
3. Test and show us your results. Run your version that takes user input and capture this test in the file named `typescript`. (Recall how from Project 2.)
4. Remember this time to `git add typescript`

3 Version with a file of test instructions

First, you need to maintain a file with test input, which will allow you to repeat the tests multiple times without having to type them every time. The format of this file needs to be, in the first line, how many (a decimal number) MIPS instructions will be processed, and then that many instructions themselves, each in hexadecimal, each in a line by itself.

We gave you the file `testInput.txt` which has the first 5 instructions from Fig. 2.1. The file `testInput_bigger.txt` has the many examples from Prof. Zheleva's lecture. Points will be given for instructions you add yourself into `testInput.txt` to make sure all the cases you programmed or should occur are covered. First, see how `DecodeControlProj.c` works with our tests. Second, for points, add at least two of your own tests and capture into `typescript` the record of doing them. Each time you add or modify a test, remember to (1) update the number of "how many" and (2) `git add testInput.txt`

Two ways to make up test input:

1. By hand: Clumsy and error-prone.
2. Learn to use command line spim or QtSpim (from CSI333) you download or use pre-installed in UA Library computers to write an assembly language file, then make spim read it, make spim print the assembled hexadecimal, and then you copy the hexadecimal form of the instructions into your `testInput.txt`. Consult files like `proj3tests.s` and `use_of_spim.txt` to see how with command line `spim`.

Now making your program take its input from your test file INSTEAD OF you typing input by hand:

```
./DecodeControlProj  < testInput.txt
```

or, if you simply compiled with `gcc -Wall DecodeControlProj.c`

```
./a.out < testInput.txt
```

Capture this work with `script` and **remember to do what's required from now on!**

4 Version to reprint instructions in binary

Finish our "stub" implementation of `print_bin32` (or replace it with your work from Project 2, if it was correct or you made it correct).

Test, `script`, commit as usual. (We did remind you!)

5 Version to detect R-format instructions

By design, a MIPS R-format instruction is easy to detect because it starts with 6 high-order zero-bits. Use this fact to finish our stub `detect_R` function. Note that it is documented in the code comments. When you are done, as usual, TEST that your code compiles and runs correctly, (and `git commit -a -m`).

6 Version to figure out register usage for R-format instructions

Put your code inside our `analyze_R` function.

Use our functions `print_reg_read` and `print_reg_written` to print which two register (numbered 0 to 31) is read by the CPU and which register is written. Print the register readings FIRST.

The MIPS instruction might specify the same register two or even three times. For example, `add $1,$1,$1` will multiply the value in reg 1 by two, just like C code `X=X+X;` is equivalent to `X=2*X;` Every time the CPU would access a particular register, your program should call `print_reg_read` or `print_reg_write` to print a report for each individual access.

7 Version to print the name of the major opcode

(The major opcode is the number 0-63 coded in binary by bits 31 to 26.)

Finish, test, etc, our "stub" implementation of `print_major_opcode_name`

This is really easy once you understand how my `PrintOpTable.c` works with the table defined in `MIPSTables.h` Understanding how to use the table is **a learning goal.** The C language has arrays of ANY TYPE OF VARIABLE, even `struct` types, not just `int` or `char`.

Notice this should work for ALL three formats of instructions!

8 Version to print what R- format instructions do

Add your code inside the `analyze_R` function.

Make it print what it does with the register values it reads, in other words, what operation (add, etc.) is used to generate the value that is written to the register your program reports with `print_reg_written`

We give no function stubs or detailed programming tips for this so you get practice with more independent problem solving and C language mastery.

9 Finish!

All your `git commit -a` actions save the latest version of already tracked or added files into **your local repository**. TO GET CREDIT, you MUST push to our server:
`git push git404@git404.cs.albany.edu:submissions/`*YourNetID*`/COProjects   master`

More Freq. Asked Questions about GIT

- You said all my previous versions are in the repository. How can I get them back, say if I lost or really messed up a file?

  1. To get a file like `DecodeControlProj.c` from your most recent (actually, HEAD) commit, simply command:`git checkout DecodeControlProj.c` Then **heed step 5. below**.
  2. Use `git log` and your informative messages to decide which commit to get a previous version of a file from.
  3. Copy the **first** 6 or so random looking characters (SHA-1 hash) after `commit`. For example, `e692e99` in `commit e692e992b520042...`
  4. To get back that commit's say `DecodeControlProj.c` file for example, command:
     `git checkout e692e99 DecodeControlProj.c`
  5. You'll have to do another commit to make that old one, or modifications of it, go into your repository.

- I finished a version but I'm not really sure it's right. Advice?
  Commit anyway, with a message that indicates doubt. Then investigate and commit any improvements you make. Extra commits, even with mistakes, do no harm, because later commits should correct them.

- What if I, after my finishing commit, I `git push` to the TAs early, and then realize I missed or could improve something?
  NO Problem! Improve your work, make more commits, and push again. Just make sure your "informative messages" will inform your TA which commit contains your best work. The *latest* commit that says "`Project 3 is done`" will be the one that is graded, if it's not late. (If you're late, discuss any issues with the TA.)

- How can we find out more about GIT?
  GIT is a key software development and collaboration technology today. It is easy to use for simple things when you follow instructions, but it is conceptually subtle and has an enormous number of capabilities, commands and options. Prof. Zheleva will explain it a little at a time throughout the

course, and please do ask her your questions.
Harvard's GIT notes from its Systems Programming and Machine Organization course at `http://cs61.seas.harvard.edu/wiki/2014/Git` are a great introduction to how we will be using GIT.
My favorite site of GIT tutorials leading to concepts and implementation is `http://www.gitguys.com/`
And `http://git-scm.com/documentation` provides Scott Chacon's comprehensive book and the reference manual free.