ECE M116C- CS M151B Computer Architecture Systems - UCLA

Computer Assignment 2

Winter 2021

The ultimate goal of these computer assignments is to create a (simple) C/C++ simulator for a (simple) RISC-V CPU. At each step of this project, we will gradually add more units to our processor.

Here are the important rules that we will use in all of these computer assignments:

- We will use the 32-bit version of RISC-V ISA and will focus on implementing only 10 instructions that were described in Lecture 6.
- Later on, we *might add more instructions and/or capabilities to our processor* (either in the subsequent computer assignments or in the Quiz-Projects).
- You should do all your work in the C/C++ programming language. Your code should be written using only the standard libraries. You may or may not decide to use classes and/or structs to write your code (C++ is preferred). Regardless of the design, your code should be modular with well-defined functions and clear comments.
- You are free to use as many helper functions/class/definitions as needed in your project.
- There is *no* restriction on what data-type (e.g., int, string, array, vector, etc.) you want to use for each parameter.
- Unfortunately, experience has shown that there is a very high chance that there are errors in this project description. The online version will be updated as errors are discovered, or if something needs to be described better. It is your responsibility to check the website often and read new versions of this project description as they become available.
- Sharing of code between students is viewed as *cheating* and will receive appropriate action in accordance with University policy. You are allowed to compare your results (***only for the debug part***) with others or discuss how to design your system. You are also allowed to ask questions and have discussions on Campuswire as long as no code is shared.

Project Description

In computer assignment 2, we will complete our single-cycle RISC-V processor. You need to add the required functionalities to the code you developed in CA1. Along with this file, we have uploaded a “trace” folder. Same as before, it has two sub-folders. Instructions are saved in “instMem” (in unsigned decimal format, where each line is one byte). In addition to three “InstMem” files in “debug” folder, we have uploaded three files: “br”, “lw/sw”, and “r-type”. These files show the actual assembly program for each of “instMem” files. You see three columns in each file, these columns show the memory address (in hex), the instruction (in hex), and the actual assembly instruction. For example:

```
“14:        00f06693        ori x13 x0 15”,
```

shows that this instruction's address is 14 (in hex), its binary value (NOTE: shown in hex) is 00f06693, and the assembly representation is *ori x13 x0 15* (NOTE: imm values are all in decimal).

These three files are designed such that the first file ("r-type") only contains r-type and i-type (except LW) instructions. **It is recommended to start with this file so you can only focus on your DE, EX, and WB stages**. For this part, you can safely assume that the next PC will be PC+4. The second file ("lw/sw") adds memory instructions to the mix (still no branch). Once you make sure your design works correctly for r-type instructions, you can start focusing on "MEM" stage. Finally, "br" file adds "beq" instruction, so you need to add support for *PC+offset* and "PCSrc".

Also note that, there is no NOP and/or END instruction in our traces. The assumption is that your instruction memory will fetch an all-zero instruction after it fetches the last instruction in each trace (you have to only count non-END instructions in your stats).

You need to use "instMem" in "test" folder to report the stats and outputs that are described at the end of this project description.

You can keep the same classes you used in CA1 (and add more classes if needed). Similar to the previous assignment, the program should run like this: "./cpusim <inputfile.txt> <-d>" and print the relevant stats/outputs in the **terminal**. In the following, we will describe the new functionalities that you need to add.

Controller. The first thing that you need to add is "Controller" function. This function should take the instruction as an input, and creates all the required control signals (see lecture 7, slide 12) for details. ***You can use a 4 bit ALUOp <u>instead</u> of using a 2-bit ALUOp with an ALU Control. You also need to compute the correct immediate once the instruction is decoded.*** You can either call this function within "Decode" or call it in some other place. Also, you may decide to add this to your "CPU" class (or not).

Register File. This is a <u>function</u> that takes "read reg 1" and "2" as its input. It also gets "write data" and "write reg" as well as "RegWrite" as inputs. Inside it, you need to handle reads and writes to the register file (Register File will be an array with 32 entries. You can define that array in your "CPU" class). Make sure to assign *x0* to zero (always)**.** Also, registers can not be written unless RegWrite is equal to one. This function has two outputs. You need to put this function inside your "CPU" class, and call it when needed.

EX. This is where all the activities during "EX" should be handled. The main part is "ALU". ***It is recommended that you use a separate function for "ALU".*** Also, make sure to use proper muxes (i.e., assignments with if/else) when you are connecting inputs/outputs. Also, don't forget to create an output for "Zero".

WB. In this function, you need to handle the writes to the register file. You can, for example, call "Register File" function again, this time with "RegWrite" equal to 1 (if the control signal is actually equal to one, otherwise, no writing is needed). Also, make sure to handle "MemtoReg" mux. Both "EX" and "WB" should be a part of your "CPU" class.

At this point, your program should be able to correctly execute the first program ("rtype"). You can use the assembly code to track the execution and the value of each register after each instruction. **In your "CPUStat" class, you need to add a function that can print the value of all registers.** This print function should be activated if the user use **"-d1"** flag when he/she executes your program (i.e., "./cpusim <input> -d1"). The print should be in this format (values in signed decimal):

```
" // Register file:
x0:         (tab)   x1:
x2:         (tab)   x3:
…
// END of Register file"
```

Mem. This function should handle reads and writes to/from the data memory. You can use the same method that you used for your instruction memory to create your data memory (don't use vectors because we assume that we can access anywhere in the memory, whenever we want!). Similar to the register file, writes can only happen if "MemWrite" is 1. **Your "CPUStat" should be able to print the values of this data memory (only addresses 0x0 to 0x1f)**. Use the format described above. To print, the user would use **"-d2"** flag. If the user uses **"-d3"**, your program should print ***BOTH*** register file and memory values.

At this point, you can test your code with "lw/sw" file.

PCAddr. This function should handle all the necessary work required to compute the next PC. This includes ***both*** computing the next address (i.e., either PC+4 or PC+offset), and whether the branch is Taken or Not (i.e., by ANDing "Zero" with "Branch"). Both "PCAddr" and "Mem" functions should be also a part of "CPU" class.

"Instruction" class. This class stays the same.

"CPUStat" class. This class will be used to keep track of different statistics during the execution of instructions in the CPU (e.g., number of executed instructions, number of cycles, etc.). In each project, we might add new metrics that need to be collected and/or calculated by this class. For this project "CPUStat" class has to use the values in "Instruction" and "CPU" classes and counts the following:

1. Total number of fetched instructions
2. Number of SW, LW, ADD, and BEQ instructions.
3. Number of Taken Branches

The output should be printed in the terminal. Each group of stats should be printed in one line, i.e., your output should look like this (with no other characters):

```
#fetch_instr
#SW, #LW, #ADD, #BEQ
#Taken
```

Further, your "CPUStat" should be able to print the values in the registers and/or data memory (only up to address 0x1f). As described before, the input command is ./cpusim <inputFile> <-dVal>, where "Val" would be either 0 (i.e., no print at all), 1 (for registers), 2 (for memory), and 3 (for both). Note that the default format should be just printing the numbers in the format shown above.

What to submit.

You need to submit the following files on Gradescope.

1. Your well-commented code (all the files) with an optional makefile/script which can be used to compile the code. Note that we will use a different trace to test your code's correctness. Your code should be compiled with the following command: "g++ *.cpp -o cpusim" (in case you are using C, it would be "gcc *.c -o cpusim"). If the code fails to compile, you will lose points. Your code should produce the results in the format described in the previous page. Failing to create the above format will result in losing points.

2. ***You need to put all your .cpp and .hpp files (and your optional makefile) in one Zip file and upload that as well*** (i.e., upload both individual files and all in one Zip file. You will lose points if you forget to upload the Zip file). ***Important:*** Name your ZIP file this: "CPUSim"

3. A short report (a PDF file) that shows the output of your code for all the stats (see "CPUStat") in a table. You need to use the trace in /test/instMem for reporting your numbers. ***Your report also needs to show screenshots for "-d3" printing option when you are running the code in /test/instMem.***