CS 252 (Spring 17): Computer Organization

Project #9

Single-Cycle CPU

due at 5pm, Fri 7 Apr 2017

1 Purpose

In this project, you will implement a single-cycle CPU. One of the keys of this project is understanding how the control bits are used to direct the rest of the processor - so most of your code will be finding (and then using) the various control bits.

Unfortunately, in order to test that you are doing this correctly (without having the TAs spend hours looking at your code), I had to break the CPU down into a fairly large number of relatively small pieces. You will implement a number of functions - most will be fairly small - and my testcases will join all of the little pieces together into a larger system.

You'll be happy to learn that in this project, I have **removed the restrictions!** Now, if you want to add, multiply, or whatever, you are allowed to do it. We can allow this now because now you know what it takes - something as simple as the `+` operator in C actually represents a non-trivial component in hardware.

We'll be writing this project entirely in C.

1.1 Required Filenames to Turn in

Name your C file `proj09.c`.

In addition, turn in a README file (named `README` or `README.txt`). This file must mention:

- Which 2 extra instructions you chose to implement (see below)
- Any extra control bits that you had to define; exactly how you stored them in the `CPUControl` struct, and what they mean
- Any new features that you added to support the new instructions. (For instance, your ALU might now support additional operations.) Document what they are, and how the control bits are used to control them.
- Any other noteworthy information that your TA should know

Since you are turning in multiple files, you may either turn them in to D2L individually, or you may zip them up (as a `.zip` or `.tar.gz`) and turn in the compressed file.

1.2 More Complex Testcases

Because the testcases have a lot of C code that they share, I now have some "common code" which is shared across all of them. This is contained in a header file `proj09_test_commonCode.h` and the matching source file `proj09_test_commonCode.c`. Each testcase will include the header, and link with the shared code.

While you are allowed to include the test-common-code header if you want, I don't think that you will need it. Feel free to poke around the code, however.

2 Instructions

Every student's code will be required to suppor the same set of basic instructions. In addition, each student will choose two additional instructions to implement.

2.1 Required Instructions

Your CPU must support all of the following instructions:

- `add, addu, sub, subu, addi, addiu`

  (Treat the 'u' instructions exactly the same as their more ordinary counterparts. Just use C addition and subtraction, using signed integers. Ignore overflow.)

- `and, or`
- `slt, slti`
- `lw, sw`
- `beq, j`

The testcases will handle the `syscall` instruction on your behalf - you don't have to write any code to make it work.

2.2 Additional Instructions

You must choose **at least** 2 additional instructions to implement. These must be standard MIPS instructions (see your book), and you must use the standard MIPS opcodes for them. Implement them exactly as the MIPS standard requires; this will probably mean that you will have to add new control bits (see below).

Your `README` file must include detailed information about what extra instructions you added, any control bits you added to make them work, and any other changes (such as new ALU operations) that were required.

While you can choose any two instructions to implement, here's a few options that you might consider:

- `bne`
- `jal`
- `jr`
- `lui`
- `andi, ori`

  NOTE: `andi` and `ori` zero-extend the immediate field, instead of sign-extending it.
- `sll, srl, sra`

2.3 Unsupported Instructions

If you are asked to decode an instruction which has an opcode or opcode/funct which you do not suppport, then you will set the `err` control bit (see below), and the testcase will not ask you to execute that instruction.

When grading your code, we will automatically test:

- That you support all of the required instructions
- That you set `err` if we give you an opcode which is not a valid MIPS instruction.

However, our testcases will **not** test opcodes which are valid MIPS (but which are not required). We'll test those by hand.

3 Typedefs, Structures, and Utility Functions

I have provided `proj09.h`, which defines several types for you (you can't change them), and the prototypes for each of the functions which you must implement.

3.1 `WORD`

`WORD` is a typedef that will be 32 bits in size. I use it for parameters and return values which need to be exactly 32 bits, and I encourage you to use it for your variables. Bit fields (from single bits, all the way up to huge ones) are represented by simple `int` variables.

3.2 `WORD signExtend16to32(int)`

This utility function is already provided by me; you don't have to write it. It takes a 16-bit input, sign extends it to a full word, and then return the value.

3.3 `struct CPUMemory`

`CPUMemory` represents everything which the CPU needs to store: its instructions, data, registers, and program counter. You will actually use this very rarely; you will only use this type at the very beginning of the instruction (`getInstruction()` and at the very end (`execute_updateRegs()`). See below for more information about both functions.

3.4 `struct CPUControl`

You will fill this struct in with all of the necessary control bits by decoding the instruction. This has lots of fields, so we will describe it in detail later in this spec. For now, remember two rules:

- You must fill in **all** of the fields in this struct, in `execute_CPUControl()`.
- You must never modify any of these fields later in your program.

3.5 `struct ALUResult`

This tiny struct has two fields: `result` (32 bits) and `zero` (1 bit, although we store it in an `int`). It simply represents the output from the ALU; you will fill in both fields in `execute_ALU()`.

3.6 `struct MemResult`

This tiny struct only has a single field - `readVal`. But I placed it inside a struct so that `execute_MEM()` will work roughly like `execute_ALU()`.

4 The Fields of `CPUControl`

The `CPUControl` struct has many fields. You need to set all of them in `execute_CPUControl()`.

4.1 Instruction Fields

The first ten or so fields are simply the fields from the instruction word. You must set **all** of these fields every time that you decode an instruction, every time that `execute_CPUControl()` is called. (This is to represent that hardware would connect all of these fields to their various destinations, in every clock cycle.)

This means, of course, that you will need to set the 'immediate' field (from I-format) and the 'address' field (from J-format), even if the instruction is an R-format instruction - and vice versa.

4.2 `err`

The second part of `CPUControl` is a single bit: `err`. After you have filled out all of the fields above, check to see if you recognize the opcode (and, if necessary, the funct field). If you do not recognize them, then set `err=1` and then stop, before you go any further. But if you recognize the opcode, then proceed to the next step.

4.3 The Real Control Bits

`CPUControl` has fields for every one of the control bits that we've discussed in class and in the book - except for the `ALUop` that goes from the main Conrol using to the ALU Control. (In this struct, we'll simply set the proper ALU operation directly, in the field named `ALU.op` .

You must set these bits exactly as we've described in class and in the book. (We'll check all of them in our testcases.) However, for your two "extra" instructions, you have more flexibility: you can add new features. For instance, you might define new ALU operations - in addiiton to the four basic ones (AND, OR, PLUS, LESS).

Sometimes, the value of a control bit may have no impact on the result of the instruction; for instance, if `regWrite=0`, then `regDst, memToReg` don't matter. However, since you need to match the testcase, your code should always set fields to zero when they are not otherwise required.

4.4 Extra Words

`CPUControl` has three extra fields (`extra1, extra2, extra3`). Each is a `WORD`. You may use these for any purpose you can imagine; our automatic grading script will not look at these fields, or print them out.

Use these fields to hold extra control bits, which you can use to implement your 2 **extra** instructions. For example, I implemented several extra instructions - and ended up using the lowest 4 bits of `extra1` as 4 different 1-bit control falgs. (I also added several new operations to the ALU.)

NOTE: While you are allowed to use the extra words for any purpose, you **must not** change how the standard control bits are used. Remember, the grading script will expect that you are using them properly!

5 The Functions

You must implement all of the following functions. **I strongly recommend that you implement these one at a time, and test them individually.** The testcases are designed to test these one at a time, sometimes in isolation, and sometimes in concert with other pieces. After the intro testcases, we will then test complete instructions, all as one pack - and then small programs.

So as you are writing your solution, start by "stubbing out" all of the required functions. That is, cut-n-paste the declarations from `proj09.h` into your file, and give them (empty) bodies. That way, the code will compile - and you can start testing - long before the rest of your code is written.

5.1 `WORD getInstruction(CPUMemory*)`

You must read the proper word out of instruction memory, given the current Program Counter. Return the value.

5.2 `void execute_CPUControl(WORD, CPUControl*)`

You must read the instruction provided, and fill in the fields of the `CPUControl` struct that was passed to you.

5.3 `WORD getALUinput*(CPUControl*, WORD *regs)`

There are two of these functions, one for each of the ALU inputs. These function take, as input, the `CPUControl` struct which you previously filled out. They must return the proper value to be delivered to each of the two ALU inputs.

Note that the second parameter is a pointer to an array of words; I intentionally give you the registers in this form (instead of passing you the `CPUMemory`. I do this because I want to demonstrate that you are actually using the control bits - instead of simply decoding the instruction again.

5.4 `void execute_ALU(WORD,WORD, int,int, ALUResult*)`

You must take the inputs to the ALU, perform the correct operation, and write the result into the `ALUResult` struct provided.

The first two parameters are the inputs to the ALU; the third is the ALU operation (copied out of the `CPUControl` struct), and the fourth is the `bNegate` input.

5.5 `void execute_MEM(CPUControl*,ALUResult*, WORD, WORD*, MEMREsult*)`

You must take the inputs to the memory, perform the correct operation (if any). You must also set the proper value in the `MemResult` struct (if you don't read anything, then simply set the field to zero.)

The first two inputs are the output structs from `execute_CPUControl()` and `execute_ALU()`. The third is the value of the `rt` register (the testcase reads this for you, based on the `rt` field in the control struct). The fourth input is an array of `WORD`s, representing main memory. The last parameter is the output struct that you must fill in.

5.6 `WORD getNewPC(CPUControl*, int, WORD)`

You must calculate the new Program Counter for the next instruction; it returns the new address from the function.

The first parameter is the `CPUControl` struct, which contains all of the control bits that you have set. The second parameter is the `zero` output from the `ALUResult` struct. The third parameter is the old Program Counter.

5.7 `void execute_updateRegs(CPUControl*, ALUResult*, MemResult*, WORD*)`

You must update the appropriate register (if any) in the `CPUMemory` struct. The first parameter is the `CPUControl` struct; the second and third are the results from the ALU and Memory operations; the last is the array of registers. As with the `getALUInput*()` functions, we use an array of registers - instead of directly giving you a pointer to `CPUControl` - so that you can only do what the control bits tell you you should do.

6 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

6.1 Testcases

You can find a set of testcases for this project at
`http://www.cs.arizona.edu/classes/cs252/spring17/projects/proj09/`
You can also find them on Lectura at
`/cs/www/classes/cs252/spring17/projects/proj09/`
(this is the backing directory for the website).

For assembly language programs, the testcases will be named `test_*.s` . For C programs, the testcases will be named `test_*.c` . For Java programs, the testcases will be named `Test_*.java` . (You will only have testcases for the languages that you have to actually write for each project, of course.)

Each testcase has a matching output file, which ends in `.out`; our grading script needs to have both files available in order to test your code.

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcases of your own, in order to better test your code.**

6.2 Automatic Testing

We have provided a testing script (in the same directory), named `grade_proj09`. Place this script, all of the testcase files (including their `.out` files if assembly language), and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac, but no promises!)

6.3 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.

While you normally cannot share code with friends and classmates, **testcases are the exception.** We encourage you to share you testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

7 Turning in Your Solution

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only your program; do not turn in any testcases or other files.