CS 252: Computer Organization

Sim #4

Single-Cycle CPU

milestone 1 - see class website for due date

milestone 2 - see class website for due date

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 Two Milestones

This project has two deadlines, and thus two assignment boxes in GradeScope.

Milestone 1 only requires that you implement the `extract_instructionFields()` and `fill_CPUControl()` functions. And, in your CPU Control function, you are only required to implement the "ordinary" instructions (see below).

For milestone 2, implement the complete spec.

NOTE: Most of the testcases are for milestone 2, but I've provided a couple for milestone 1. The milestone 1 testcases include stub functions for the functions you are not required to implement (to make it easier to write milestone 1). But they won't compile anymore once you define your own, real versions.

1.2 "Extra" Instructions and Their Testcases

There are a certain set of instructions which all students must implement - basically, these are the set which can be easily implemented by the basic processor design that we've provided for you.

However, you will also be expected to **expand** on this design a bit. We have provided a set of "extra" instructions, which can also be implemented

with (relatively) small changes to the processor. You must choose three of these instructions, and update your design to support them (details below).

When you download the testcases for your program, you should download the "ordinary" testcases (they are all numbered), and then download the "extra" testcases for the instructions that you have chosen. Test them all using the grading script.

1.3 Required Filenames to Turn in

Both milestones require that you turn in a C file; name it `sim4.c`. (If you're an overachiever and complete the entire project in the first week, then turn in the file into **both** folders.)

In addition, in **milestone 2 only,** you must turn in testcases for each of the three "extra" instructions that you have chosen to implement. You can find the appropriate testcases in the `extra_testcases/` folder in the zipfile. Turn in exactly the testcases I have provided; if you modify them at all, or if you don't turn them in, then the autograder will treat them as failed testcases.

1.4 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 `sim4_test_commonCode.h` and the matching source file `sim4_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, if you find that interesting.

1.5 Random Numbers

Please note that testcase 08 uses C's `rand()` function to generate inputs. The `.out` file that I have provided works when running on GradeScope, but it may or may not work on your machine. If you are failing testcase 08, try running it on GradeScope instead.

(spec continues on the next page)

2 "Ordinary" 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, xor`[1]

- `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.

Your implementation for these instructions must match the book - and the slides we've been going over in lecture. You must use the same control wires, with the same meanings.[2] Your "extra" control wires (if any) must all be **zero** for these instructions.

(spec continues on the next page)

[1] The book's version of the CPU cannot do XOR - but in Sim 3, we added ALU op 4. We'll keep that up in this project - you must implement XOR as ALU operation 4. We'll pretend that it is "standard."

[2] The only exception - as I've said in class, is that we don't use a separate "ALU Control" in our processor. Instead, you will set the ALU operation directly, in your control logic.

2.1 "Extra" Instructions

Choose three instructions from the following list (no more than one from each group below!), and implement them as well in your CPU:

- `andi, ori, xori, nor`
- `lui`
- `sra, srl, sll`
- `srav, srlv, sllv`
- `bne`
- `lb, sb`
- `mul`
- `mult, div`[3]
- `mfhi, mflo`

Your implementation for these instructions must match the MIPS architecture; you can look each of these up, in Appendix A, to find out exactly how they work. **You must use the opcode that MIPS requires, and do exactly what the MIPS architecture says these instructions do.**

However, none of these can be implemented with the standard CPU design that we've discussed. Each one needs some sort of small change. Some can be implemented by adding a new ALU operation, or by adding a new input to an existing MUX; others will require that you add a new MUX, or new logic somewhere in the processor.

2.2 New Control Bits

To support your extra features, you must set new control bits in the CPU design. One option is to simply add new values to a field; for instance, you could add an ALU operation 5 (the standard processor only supports 0-4[4]), which did something new. Likewise, you might add a new input to the `ALUsrc` MUX (which feeds into ALU input 2).

Alternatively, you might add entirely new control lines. For instance, you might add a new control wire which means "invert the Zero output from the ALU." To support these, the `CPUControl` struct (see below) includes 3 different "extra" fields. **You get to decide what these mean - in theory you could have 96 new control wires!** Or, you may ignore them, if you don't need them.

[3] If you implement this, you **must** also implement `mfhi` or `mflo` so that we can test your results.

[4] See the footnote above, about XOR and ALU op 4.

I have just one limitation: you must **not** simply copy the instruction, opcode, or funct fields into the `extra` fields. Instead, you must **decode** the opcode/funct, and **set control wires.** It's just that you get to decide what the new control wires mean.[5]

2.3 "Don't Cares"

In some instructions, there are control bits which don't matter. For instance, in any instruction which sets `regWrite=0`, the control bits `regDst` and `memToReg` don't matter, since they can't possibly affect what happens this clock cycle.

In real hardware, these are called "don't cares," and thus we can set these to **whatever** value - basically, we choose arbitrary values for them - whatever makes the hardware implementation cheaper.

But in our code, we will be dumping these values in the testcase, and comparing them against a standard output - so we need a standard value for each one. So our rule is: **if you don't care about a control wire, always set it to zero.**

2.4 Invalid Instructions

In real hardware, if your code includes an instruction with a bad opcode, the program will crash. That is, the CPU has special circuits which force an "exception" if an invalid opcode is found; this forces the software into the Operating System - and normally results in the OS killing your program.

To simplify things, you won't implement any of this, with one exception: if the opcode is invalid (or if the opcode indicates "R-format" but the funct field is invalid), then you will return 0 from `fill_CPUControl()`. (If it is recognized, then you will return 1, after filling in all of the control bits.)

2.5 Not-Required Instructions

Some students like to implement some additional instructions that I don't require. This is fine, but they **must be standard MIPS instructions**, using the normal opcodes. For instance, you could implement `jal` - but if so, it **must** use opcode 0x3.

Our testcases will check to see if your Control component returns 0 from `fill_CPUControl()` as required. But since we don't know which students may have supported additional (normal) MIPS instructions, we will **only test opcodes that are not valid in standard MIPS.**

Remember, only instructions from the **approved list above** count towards your grade. (Sorry, but those are the only ones for which we have testcases.)

[5] For the same reason, global variables are forbidden!

3 Typedefs, Structures, and Utility Functions

I have provided `sim4.h`, which defines several types for you and the prototypes for each of the functions which you must implement.

You must not change this header. If you do, your code won't compile when I test it.

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 (when they are 32 bits as well). Bit fields (from single bits, all the way up to huge ones) are represented by simple `int` variables.

So yes, there are some `int`s in the data structures that represent a single bit (such as `bNegate`). There are others (such as `funct`) that represent 5 or 6 bits, and some (such as `imm16`) that even represent 16 bits!

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 returns the value.

3.3 `struct InstructionFields`

This struct is initialized by the `extract_instructionFields()` function, which you will write.

This struct represents all of the fields in the instruction. When you fill this struct, set **all** of the fields, with **no intelligence at all.** So, for instance, you should always set the `address` field - even if this is not a `J` instruction; the `address` field just represents the 26 wires, which can be then connected to other places.

3.4 `struct CPUControl`

This struct is initialized by the `fill_CPUControl()` function, which you will write. (This function will make up most of your code for this project.)

This struct represents all of the control bits. Your code will read the function fields (out of an `InstructionFields` struct), decode the opcode an funct fields, and set all of the control bits that are required.

We will describe `fill_CPUControl()` in detail later in this spec. For now, remember two rules:

- You must fill in **all** of the fields in this struct, in `fill_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, stored in a `WORD` variable) 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()`.

Always set the `zero` output (to either 0 or 1), no matter what ALU operation you perform.

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()`.[6]

4 The Fields of `CPUControl`

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

4.1 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`[7] .)

See above for the discussion of "ordinary" and "extra" instructions (those that can be implemented by the processor design shown in the book, and those that need additional control bits). For the "ordinary" instructions, you must implement the control bits **exactly as described in the book and slides.** For the "extra" instructions, you must either define new control bits (which are zero for the "ordinary" instructions, but which you turn on for certain "extra" instructions), or define new values (such as adding new ALU operations, or adding more inputs to a MUX).

As noted above, there are some situations where a bit may be a "don't care" - meaning it can have no effect on the operation of the CPU for this clock cycle (because other bits make it pointless). Always set "don't cares" to zero.

[6] Maybe I'm being silly, but I like symmetry!

[7] In C, the operator `->` is used to access the fields inside a struct, when you have a **pointer** to the struct. The `.` ('dot') operator is the same thing, but when you have the struct itself, not a pointer to it.

In this project, we pass a **pointer** to the CPUControl struct to your function, `fill_CPUControl()`. So to access most fields, you will use `->`, like this:

```
pointer->field = value;
```

However, the are a couple of fields which are grouped together into an `ALU` struct, inside `CPUControl`. You must use `->` to get from the pointer to the `ALU` struct, and then `.` to get the field **inside** the ALU struct, like this:

```
pointer->ALU.op = 2;
```

4.2 Extra Words

Finally, `CPUControl` has three extra `WORD`s provided for you. These fields must all be zero for all "ordinary" instructions. However, if you want, you can use these to store up to 96 additional control bits, to make the "extra" instructions work.

It is not required that you use these fields. Some designs may support the "extra" fields simply by adding new legal values to existing fields (such as the ALU operation). But I've provided these for you **just in case you find them handy.**

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 first several 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.

As you are writing your solution, start by "stubbing out" all of the required functions. That is, cut-n-paste the declarations from `sim4.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(WORD curPC, WORD *instructionMemory)`

You must read the proper word out of instruction memory, given the current Program Counter. Return the value. Remember that the Program Counter gives the address of the current instruction in **bytes** but that this is an **array of words.**

5.2 `void extract_instructionFields(WORD, InstructionFields *fieldsOut)`

This function is passed an instruction as input; it must read all of the fields out of the instruction, and store them into the fields in the `InstructionFields` struct.

No other function in your code may modify any field in this struct!

5.3 `int fill_CPUControl(InstructionFields*, CPUControl*)`

The first parameter is an **in** parameter: it is the `InstructionFields` struct that you filled in, in the previous function. The second is a `CPUControl*` struct.

Read the opcode and funct from the Fields struct, and then set all of the correct controls in the Control struct.

This function returns 1 if the instruction is recognized, and 0 if the opcode or funct is invalid.

No other function in your code may modify any field in this struct!

5.4 `WORD getALUinput*(...)`

There are two of these functions, one for each of the ALU inputs. Each function returns a `WORD`, which is the value that should be delivered to that input of the ALU.

These functions have **many** input parameters. You'll only use a very few of them to implement the standard instructions, although you might use a few more when you implement your "extra" instructions for milestone 2. Don't worry if you are ignoring most of them - I just include them for **maximum flexibility** when you implement the "extra" instructions.

The parameters include:

- The `CPUControl` for this instruction
- The `InstructionFields` for this instruction
- The value of the two registers read from the register file (based on the `rs,rt` fields you set in `InstructionFields`)
- The value of registers 33 and 34.

  (The basic set of registers only has 32 registers - but our simulator has 34, so that you can implement the `lo,hi` registers for multiply, if you want.)
- The old PC value - that is, the PC of the currently executing instruction. (This parameter only exists to support certain extra instructions, which some students might implement.)

5.5 `void execute_ALU(CPUControl*, WORD,WORD, ALUResult*)`

This function implements the ALU. Remember, we've **removed the limitations** - so I fully expect you to use the C addition and subtraction operators (or anything else that might be handy). However, you must choose **what** you do only by reading the various control fields in the `CPUControl` struct. (You will notice that you don't have access to the instruction itself in this function!)

The second and third parameters are the ALU inputs 1 and 2 (see the functions above).

The fourth parameter is an **out** parameter: set **all** of its fields, every time that this function is called!

5.6 `void execute_MEM(CPUControl*,ALUResult*, WORD,WORD, WORD*, MEMResult*)`

This function implements the data memory unit. The first parameter is the CPU control; check for the memory control bits inside it to see what you need to do (if anything). Most of the time, you will do nothing - but even in that case, you must set the output bits (to zero).

The second parameter is the `ALUResult` struct, which you set in a previous call to `execute_ALU()`. You must not change anything inside this struct, but you can read the fields.

The third and fourth parameters are the two registers that were read for this instruction - based on the `rs,rt` values you set in the `InstructionFields` struct.

The fifth parameter is an array of `WORD`s, representing the data memory. (Remember: all memory addresses are given in bytes, but the array is an array of words.)

The final parameter is an **out** parameter: it is the "Read data" field, coming out of the memory unit. If you read a value from memory, then this field must have that value. If not (if you write, or if you do nothing), then you must **set this to zero.**

5.7 `WORD getNextPC(...)`

This function implements the logic which decides what the next PC will be. The first parameter is the `InstructionFields` for this instruction, and the second is the `CPUControl`. The third is the `aluZero` output from the ALU. The next two are the two registers read for this instruction; the fifth is the previous PC.

Return the new PC.

5.8 `void execute_updateRegs(...)`

This represents the final stage of the processor: writing to a register (if required). The first parameter is the fields of the instruction, followed by the control bits; the third and fourth are the results from the ALU and Memory, respectively.

The last parameter is a pointer to the current set of registers, which you may have to write to.

6 Data and Instruction Memory Sizes; Number of Registers

I will ensure (in my testcases) that any address we use (any PC or data address) will be small enough to fit into the memory that I provide you. So you don't have to check for any address is that is "too large." (I will also not ever use unaligned memory addresses.)

When I pass you registers (`getALUInput*(), execute_updateRegs()`), I will always pass you an array of 34 registers. This is to represent the 32 ordinary registers, plus the `lo,hi` registers which are used for multiplication and division. You are **not** expected to implement multiplication and division, but I wanted to make it possible for the students who were interested.

6.1 Register Zero

I'm not sure, in the official MIPS specification, what happens when you try to write to register `$zero`. (I'm sure that it doesn't actually change anything - but I don't know if there are any errors or other side-effects that are required.)

So, in our testcases, **we will never try to write to it** - meaning that you don't need to write any special code to handle it. Moreover, you may assume, any time I pass you the array of registers, that element `[0]` of the array contains 0.

7 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!

7.1 Testcases

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.**

7.2 Automatic Testing

We have provided a testing script (in the same directory), named `grade_sim4`. Place this script, all of the testcase files (including their `.out` files if assembly language), and your program files in the same directory (in your Docker container).

7.3 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 Discord. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

8 Turning in Your Solution

You must turn in your code to GradeScope. Turn in only your program; do not turn in any testcases.