CS61C Summer 2015 Project 2-1: MIPS Assembler

TA: Derek Ahmed

Due Sunday, July 12, 2015 @ 11:59pm

Updates (Check here if there are updates in the future)

Commit and push your work to GitHub before updating, or you risk losing your work. Only one partner needs to apply any future updates, so communicate with your partner beforehand. To apply the update, fetch/merge from the `proj2_starter` repository by typing:

```
git fetch proj2-starter
git merge proj2-starter/master -m "merge proj2-1 update"
```

Then, verify that your files are OK before pushing the changes to GitHub.

Changelog (a log for future changes):

- **Update #1 (7/09/15 8:30 PM)** - Two sets of changes can be viewed here:
    1. Bugfix to test_assembler.c regarding table name for test_table_1().
    2. Updated reference files for testing

So What Is This About?

In this part of the project, we will be writing an assembler that translates a subset of the MIPS instruction set to machine code. Our assembler is a two-pass assembler similar to the one described in lecture. However, we will only assemble the `.text` segment. At a high level, the functionality of our assembler can be divided as follows:

Pass 1: Reads the input (`.s`) file. Comments are stripped, pseudoinstructions are expanded, and the address of each label is recorded into the symbol table. Input validation of the labels and pseudoinstructions is performed here. The output is written to an intermediate (`.int`) file .

Pass 2: Reads the intermediate file and translates each instruction to machine code. Instruction syntax and arguments are validated at this step. The relocation table is generated, and the instructions, symbol table, and relocation table are written to an object (`.out`) file.

The Instruction Set

Please consult the MIPS Green Sheet for register numbers, instruction opcodes, and bitwise formats. Our asembler will support the following registers: **$zero**, **$at**, **$v0**, **$a0 - $a3**, **$t0 - $t3**, **$s0 - $s3**, **$sp**, and **$ra**. The name **$0** can be used in lieu of $zero. Other register numbers (eg. $1, $2, etc.) are not supported.

We will have 18 instructions and 5 pseudoinstructions to assemble, one that is not a real pseudoinstruction. The instructions are:

INSTRUCTION	FORMAT
Add Unsigned	`addu $rd, $rs, $rt`
Or	`or $rd, $rs, $rt`
Set Less Than	`slt $rd, $rs, $rt`
Set Less Than Unsigned	`sltu $rd, $rs, $rt`
Jump Register	`jr $rs`
Shift Left Logical	`sll $rd, $rt, shamt`
Add Immediate Unsigned	`addiu $rt, $rs, immediate`
Or Immediate	`ori $rt, $rs, immediate`
Load Upper Immediate	`lui $rt, immediate`
Load Byte	`lb $rt, offset($rs)`

Load Byte Unsigned	lbu $rt, offset($rs)
Load Word	lw $rt, offset($rs)
Store Byte	sb $rt, offset($rs)
Store Word	sw $rt, offset($rs)
Branch on Equal	beq $rs, $rt, label
Branch on Not Equal	bne $rs, $rt, label
Jump	j label
Jump and Link	jal label

The pseudoinstructions are:

PSEUDOINSTRUCTION	FORMAT
Load Immediate	li $rt, immediate
Move	move $rd, $rs
Branch on Less Than	blt $rs, $rt, label
Branch on Greater Than	bgt $rs, $rt, label
New Add Unsigned	traddu $rd, $rs, $rt

IMPORTANT: The way traddu works is it stores the sum of three registers $rs, $rt, $rd in register $rd. R[rd] = R[rs] + R[rt] + R[rd].

Implementation Steps

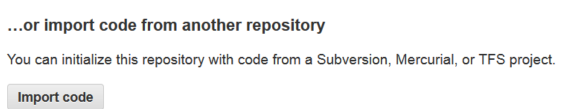
Please note that your project will be graded on the HIVE machines. While you are free to develop on other machines, you need to make sure that your code compiles and runs without errors on the hive machines before submitting. If you do not, you run the risk of turning in non-compiling code and **getting a ZERO on the entire project**.

Step 0: Obtaining the Files

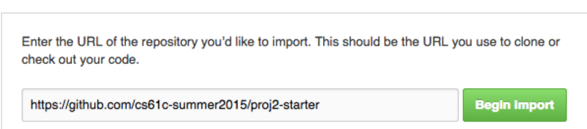
To make this process go as smoothly as possible, make sure you:

1. **Use the shared proj2-XX-YY repositories for this project, NOT your individual repositories.**
2. **Create your shared git repository outside of any existing repositories (unless you *really* know what you're doing).**

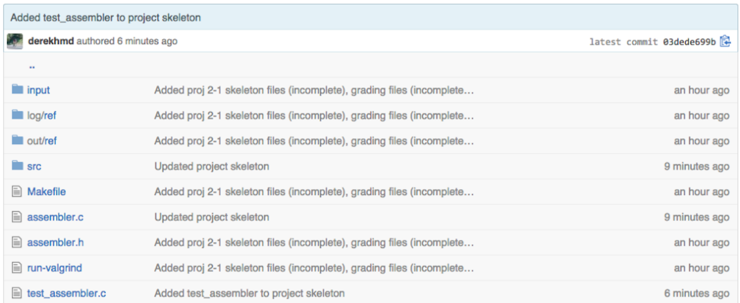
If your shared repository is empty, you can use the GitHub importer by going to the repository's page (`https://github.com/cs61c-summer2015/cs61c-XX-YY`), scrolling to the bottom, and clicking the import button:



Click on "Import Code", and on the next screen enter the URL of the starter repository (`https://github.com/cs61c-summer2015/proj2-starter`):



It will take GitHub a few moments to import the code. After it is done, your repository should have the same files as the starter repository:



Added test_assembler to project skeleton

derekhmd authored 6 minutes ago latest commit 03dede699b

..		
input	Added proj 2-1 skeleton files (incomplete), grading files (incomplete...	an hour ago
log/ref	Added proj 2-1 skeleton files (incomplete), grading files (incomplete...	an hour ago
out/ref	Added proj 2-1 skeleton files (incomplete), grading files (incomplete...	an hour ago
src	Updated project skeleton	9 minutes ago
Makefile	Added proj 2-1 skeleton files (incomplete), grading files (incomplete...	an hour ago
assembler.c	Updated project skeleton	9 minutes ago
assembler.h	Added proj 2-1 skeleton files (incomplete), grading files (incomplete...	an hour ago
run-valgrind	Added proj 2-1 skeleton files (incomplete), grading files (incomplete...	an hour ago
test_assembler.c	Added test_assembler to project skeleton	6 minutes ago

Now you can clone the repository to your cs61c account/your computer, and then add the starter repository as a remote:

```
cd ~                            # Make sure you are outside of any existing repositories (eg. ~/work)
git clone git@github.com:cs61c-summer2015/proj2-XX-YY.git
cd proj2-XX-YY                  # Go inside the directory that was created
git remote add proj2-starter git@github.com:cs61c-summer2015/proj2-starter
```

If you have already made commits to your shared repository, then simply add the

```
# Inside of your proj2-XX-YY repository
git remote add proj2-starter git@github.com:cs61c-summer2015/proj2-starter
git fetch proj2-starter
git merge proj2-starter/master -m "merge proj2-1 skeleton code"
```

You can compile you code by typing `make`. At first, you will get a bunch of `-Wunused-variable` and `-Wunused-function` warnings. The warnings tell you that variables/functions were declared, but were not used in your code. Don't worry, as you complete the assigment the warnings will go away.

Step 1: Building Blocks

Finish the implementation of `translate_reg()` and `translate_num()` in `src/translation_utils.c`. `translate_reg()` is incomplete, so you need to fill in the rest of the register translations. You can find register numbers on the MIPS Green Sheet. Unfortunately, there are no built-in `switch` statements for strings in C, so an if-else ladder is the way to compare multiple strings.

For `translate_num()`, you should use the library function `strtol()` (see documentation here). `translate_num()` should translate a numerical string (either decimal or hexadecimal) into a signed number, and then check to make sure that the result is within the bounds specified. If the string is invalid or outside of the bounds, return -1.

Step 2: SymbolTable

Use the given SymbolTable data structure to store symbol name-to-address mappings in `src/tables.c`. Multiple `SymbolTables` may be created at the same time, and each must resize to fit an arbitrary number of entries (so you should use dynamic memory allocation). A SymbolTable struct has been defined in `src/tables.h`, and you may use the existing implementation or create your own if that feels more intuitive. Feel free to declare additional helper methods. See `src/tables.c` for details.

In `add_to_table`, you cannot simply store the character pointer that was given, as it could point to a temporary array. You must store a copy of that string instead. You should use the helper functions defined in `src/tables.c` whenever appropriate. The provided SymbolTable also comes with an `INITIAL_SIZE` and `SCALING_FACTOR` to appropriately create a table and also increase the size of the table when trying to add to a table currently at full capacity. It may help to complete `get_addr_forsymbol()` prior to attempting to complete `add_to_table()`.

You must make sure to free all memory that you allocate. See the Valgrind section under testing for more information.

Step 3: Instruction Translation

Implement `translate_inst()` in `src/translate.c`. The MIPS Green Sheet will again be helpful and perhaps even necessary, and so will bitwise operations.

`translate_inst()` should translate instructions to hexadecimal. Note that the function is incomplete. You must first fix the funct fields, and then implement the rest of the function.You will find the `translate_reg()`, `translate_num()`, and `write_inst_hex()` functions, all defined in `translate_utils.h` helpful in this step. Some instructions may also require the symbol and/or relocation table, which are given to you by the `symtbl` and `reltbl` pointers. You are required to implement the `write_*()` functions declared in `translate.h`. This is a requirement for your benefit. To lower the intensity of the project, most of the code for this step has been provided, however it is not complete and some of the code provided may not have the correct values or assignments. The more important issue is input validation -- you must make sure that all arguments given are valid. If an input is invalid, you should NOT write anything to output but return -1 instead.

Use your knowledge about MIPS instruction formats and think carefully about how inputs could be invalid. You are encouraged to use MARS as a resource. Do note that MARS has more pseudoinstruction expansions than our assembler, which means that instructions with invalid arguments for our assembler could be treated as a pseduoinstruction by MARS. Therefore, you should check the text section after assembling to make sure that the instruction has not been expanded by MARS .

If a branch offset cannot fit inside the immediate field, you should treat it as an error.

Step 4: Pseudoinstruction Expansion

Implement `write_pass_one()` in `src/translate.c`, which should perform pseudoinstruction expansion on the **load immediate (li)**, **move (move)**, **branch on less than (blt)**, **branch on greater than (bgt)**, and **new add unsigned (traddu)** instructions.

The `li` instruction normally get expanded into an `lui-ori` pair. However, an optimization can be made when the immediate is small. If the immediate can fit inside the `imm` field of an `addiu` instruction, we will use an `addiu` instruction instead. Other assemblers may implement additional optimizations, but ours will not.

The `move` instruction essentially copies over the contents of one register to another. Think about some operation consisting of a non-pseudo instruction that could accomplish this.

For the `blt` instruction, use the fewest number of instructions possible. After implementing `blt` , `bgt` should not be too difficult to implement.

The `traddu` instruction is not a real pseudoinstruction (but assume it is for this project). This should be quite self-explanatory. How can we take the contents of our three registers and store their sum into one? Hint: you should consider using `addu`.

Also, make sure that your pseudoinstruction expansions do not produce any unintended side effects. You will also be performing some error checking on the pseudoinstructions (see `src/translate.c` for details). If there is an error, do NOT write anything to the intermediate file, and return 0 to indicate that 0 lines have been written.

Step 5: Putting It All Together

Implement `pass_one()` and `pass_two()` in `assembler.c`. You may find the use of `parse_args()` very helpful in this task. You may find the use of `parse_args()` very helpful in this task. **Please keep in mind that if you use parse_args(), it MUST be called after the provided line "char* token = strtok(buf, IGNORE_CHARS);" in pass_one() because parse_args() heavily depends on strtok(). This has to do with the way strtok() works and how passing in NULL as an argument for strtok() is dependent on the last successful function call of strtok()** In the first pass, the assembler will strip comments, add labels to the symbol table, perform pseudoinstruction expansion, and write assembly code into an intermediate file. The second pass will read the intermediate file, translate the instructions into machine code using the symbol table, and write it to an output file. Afterwards, the symbol table and relocation table will be written to the output file as well, but that has been handled for you.

Before you begin, make sure you understand the documentation of [fgets()] and [strtok()]. It will be easier to implement `pass_two()` first. The comments in the function will give a more detailed outline of what to do, as well as what assumptions you may make. **Your program should not exit if a line contains an error.** Instead, keep track of whether any errors have occured, and if so, return -1 at the end. `pass_one()` should be structured similarly to `pass_two()`, except that you will also need to parse out comments and labels. You will find the `skip_comment()` and `add_if_label()` functions useful.

As an aside, our parser is much more lenient than an actual MIPS parser. Building a good parser is outside the scope of this course, but we encourage you to learn about finite state automata if you are interested.

Line Numbers and Byte Offsets

When parsing, you will need to keep track of two numbers, the line number of the input file and the byte offset of the current instruction. Line numbers start at 1, and include whitespace. The byte offset refers to how far away the current instruction is from the first instruction, and does NOT include whitespace. You can think of the byte offset as where each instruction will be if the instructions were loaded into memory starting at address 0. See below for an example.

The address of a label is the byte offset of the next instruction. In the example below, `L1` has an address of 4 (since the next instruction is `lw`, whose address is 4) and `L2` has an address of 8 (since the next instruction is `ori`, whose address is 8).

LINE #	INPUT FILE	OUTPUT FILE	BYTE OFFSET
1	`    addiu $t0 $a0 0`	`addiu $t0 $a0 0`	0
2	`L1: lw $t1 0($t0)`	`lw $t1 0($t0)`	4
3	`# This is a comment`	`ori $t1 $t1 0xABCD`	8
4	`L2:`	`addiu $t1 $t1 3`	12
5	`    ori $t1 $t1 0xABCD`	`bne $t1 $a2 label_2`	16
6	`    addiu $t1 $t1 3`		
7			
8	`    bne $t1 $a2 L2`		

Error Handling

If an input file contains an error, we only require that your program print the correct error messages. The contents of your `.int` and `.out` files do not matter.

There are two kinds of errors you can get: errors with instructions and errors with labels. Error checking of labels is done for you by `add_if_label()`. However, you will still need to record that an error has occurred so that `pass_one()` can return -1.

In `pass_one()`, errors with instructions can be raised by 1) `write_pass_one()` or 2) the instruction having too many arguments. In `pass_two()`, errors with instructions will only be raised by `translate_inst()`. Both `write_pass_one()` and `translate_inst()` should return a special value (0 and -1 respectively) in the event of an error. You will need to detect whether an instuction has too many arguments yourself in `pass_one()`.

Whenever an error is encountered in either `pass_one()` or `pass_two()`, record that there is an error and move on. Do not exit the function prematurely. When the function exits, return -1.

For information about testing error message, please see the "Error Message Testing" section under "Running the Assembler".

Step 6: Testing (NO, DO NOT JUST SKIP THIS SECTION.)

You are responsible for testing your code. While we have provided a few test cases, they are by no means comprehensive. In the previous semester's rendition of the project, it was heavily reflected in student grades who and who did not test their code. Fortunately, you have a variety of testing tools at your service.

CUnit

Note: CUnit tests must be compiled on either the hive or the Soda 2nd floor machines, or you will get compilation errors. We have set up some unit tests in `test_assembler.c`. You can run them by typing:

```
make test-assembler
```

Check out [Project 1] for a refresher on CUnit. You are encouraged to write more tests than the ones that we have given.

Valgrind

You should use Valgrind to check whether your code has any memory leaks. We have included a file, `run-valgrind`, which will run Valgrind on any executable of your choosing. If you get a permission denied error, try changing adding the execute permission to the file:

```
chmod u+x run-valgrind
```

Then you can run by typing:

```
./run-valgrind <whatever program you want to run>
```

For example, you wanted to see whether running `./assembler -p1 input/simple.s out/simple.int` would cause any memory leaks, you should run `./run-valgrind ./assembler -p1 input/simple.s out/simple.int`.

MARS

Since you're writing an assembler, why not refer to an existing assembler? MARS is a powerful reference for you to use, and you are encouraged to write your own MIPS files and assemble them using MARS.

Warning: in some cases the output of MARS will differ from the specifications of this project. You should always follow the specs. This is because MARS 1) supports more pseudoinstructions, 2) has slightly different pseudoinstruction expansion rules, and 3) acts as an assembler and linker. For example, MARS may expand an `addiu` with a 32-bit immediate into `li` followed by an `addiu` instruction. Not only will the machine code be different, but the addresses of any labels will also be different. Therefore, you should always examine the assembled instructions carefully when testing with MARS.

MARS also supports an assemble-only mode via the command-line (see [documentation here](#)). To save assembled output to a text file, run:

```
mars a dump .text HexText <output location> <input MIPS file>
```

Diff (this should be your best friend for this project)

`diff` is a utility for comparing the contents of files. Running the following command will print out the differences between `file1` and `file2`:

```
diff <file1> <file2>
```

To see how to interpret diff results, [click here](#). We have provided some sample input-output pairs (again, these are not comprehensive tests) located in the `input` and `out/ref` directories respectively. For example, to check the output of running `simple.s` on your assembler against the expected output, run:

```
./assembler input/simple.s out/simple.int out/simple.out
diff out/simple.out out/ref/simple_ref.out
```

Running the Assembler

First, make sure your assembler executable is up to date by running `make`.

By default, the assembler runs two passes. The first pass reads an input file and translates it into an intermediate file. The second pass reads the intermediate file and translates it into an output file. To run both passes, type:

```
./assembler <input file> <intermediate file> <output file>
```

Alternatively, you can run only a single pass, which may be helpful while debugging. To run only the first pass, use the `-p1` flag:

```
./assembler <-p1> <input file> <intermediate file>
```

To run only the second pass, use the `-p2` flag. Note that when running pass two only, your symbol and relocation table will be empty since labels were stripped in pass_one(), so it may affect your branch instructions.

```
./assembler <-p2> <intermediate file> <output file>
```

When testing cases that should produce error messages, you may want to use the `-log` flag to log error messages to a text file. The `-log` flag should be followed with the location of the output file (WARNING: old contents will be overwritten!), and it can be used with any of the three modes above.

Error Message Testing

We have provided two tests for error messages, one for errors that should be raised during `pass_one()`, and one for errors that should be raised during `pass_two()`. To test for `pass_one()` errors, assemble `input/p1_errors.s` with the `-p1` flag and verify that your output matches the expected output:

```
./assembler -p1 input/p1_errors.s out/p1_errors.int -log log/p1_errors.txt
diff log/p1_errors.txt log/ref/p1_errors_ref.txt
```

To test for `pass_two()` errors, assemble `input/p2_errors.s` running both passes:

```
./assembler input/p2_errors.s out/p2_errors.int out/p2_errors.out -log log/p2_errors.txt
diff log/p2_errors.txt log/ref/p2_errors_ref.txt
```

Your intermediate and output files (`.int` and `.out` files) do NOT need to match the reference output if the input file contains an error.

Submission

Did you test thoroughly on the hive machines? If you do not, you risk getting a ZERO on the entire project.

There are **two** steps required to submit proj2-1. Failure to perform both steps will result in loss of credit:

1. First, you must submit using the standard unix submit program on the instructional servers. This assumes that you followed the earlier instructions and did all of your work inside of your `git` repository. To submit, follow these instructions after logging into your cs61c-XX class account:

   ```
   cd ~/proj2-XX-YY                              # Or where your shared git repo is
   submit proj2-1
   ```

   Once you type `submit proj2-1`, follow the prompts generated by the submission system. It will tell you when your submission has been successful and you can confirm this by looking at the output of `glookup -t`.

2. Additionally, you must submit proj2-1 to your **shared** GitHub repository:

   ```
   cd ~/proj2-XX-YY                              # Or where your shared git repo is
   git add -u
   git commit -m "project 2-1 submission"
   git tag "proj2-1-sub"                         # The tag MUST be "proj2-1-sub". Failure to do so will result in loss of credit.
   git push origin proj2-1-sub                   # This tells git to push the commit tagged proj2-1-sub
   ```

Resubmitting

If you need to re-submit, you can follow the same set of steps that you would if you were submitting for the first time, but you will need to use the `-f` flag to tag and push to GitHub:

```
# Do everything as above until you get to tagging
git tag -f "proj2-1-sub"
git push -f origin proj2-1-sub
```

Note that in general, force pushes should be used with caution. They will overwrite your remote repository with information from your local copy. As long as you have not damaged your local copy in any way, this will be fine.