CS 241 — Fall 2021 — Assignment 3

Assignments for CS 241

--><- Assignment 2	Assignment 3	Assignment 4 ->
Friday, Jan 28th at 5:00 pm	**Friday, Feb 4th at 5:00 pm**	Friday, Feb 11th at 5:00 pm

P1 • P2 • P3 • P4 • P5 • P6 • P7 • P8 • P9 • P10 • Comprehensive Test

In Assignment 3, you will incrementally write an assembler for MIPS assembly language (CS241 dialect). We have provided starter code implementing a scanner to help you read the input, but it is up to you to do the translation from assembly language to machine code.

Reminder: For this and future assignments, be sure to run the command `source /u/cs241/setup` to gain access to the CS 241 tools.

What Is An Assembler?

So far in the course, you have used various tools like `cs241.wordasm`, `cs241.binasm`, `mips.twoints` and `mips.array`.

The tools `cs241.wordasm` and `cs241.binasm` are **MIPS assemblers**. They convert ASCII text commands into MIPS machine language that can be executed by the **MIPS emulators**, which are `mips.twoints` and `mips.array`.

This is important to understand: an assembler does not execute code. If the user writes something like `div $0, $0`, this will obviously cause an error if it is executed. However, the assembler's job is just to **translate** this instruction into machine language. When the emulator executes it, the error will happen, but you are not writing an emulator, just an assembler.

The `cs241.wordasm` assembler you used on Assignment 1 was a primitive assembler that only supports the `.word` directive and nothing else. The `cs241.binasm` assembler you used on Assignment 2 supports all features of MIPS assembly language. Your task on Assignment 3 is to write your own version of `cs241.binasm`.

This means that, for valid assembly programs, you can check that your output is correct by comparing it to `cs241.binasm` using the `diff` command:

```
% ./asm < input.asm > my-output.mips
% cs241.binasm < input.asm > expected-output.mips
% diff my-output.mips expected-output.mips
```

If the files are the same, diff will produce no output, and this means your output is correct. Otherwise it will probably say something like `Binary files my-output.mips and expected-output.mips differ`.

For invalid assembly programs, the only requirement is that your assembler produce an error message containing the string ERROR to standard error. The error message does not need to match the one produced by `cs241.binasm`.

Why do assemblers sometimes seem to produce strange output, or no output at all?

If you send the output of `cs241.binasm` to the terminal, it will probably either display something unreadable, or nothing at all. Terminals interpret program output as ASCII text, but an assembler produces MIPS machine code, which doesn't look like human-readable ASCII text. In fact, if you have implemented your assembler correctly, you should see the exact same bizarre-looking output from your own assembler as you do from `cs241.binasm`.

Use the command `cs241.binview` to view assembler output in a human-readable form:

```
% cs241.binasm <<< 'add $1, $2, $3' > output.mips
% cs241.binview output.mips
00000000 01000011 00001000 00100000
```

Tips for using `cs241.binview`

You can pipe the output of `cs241.binasm` into `cs241.binview`, rather than creating an intermediate file as above. To do this, you need to pass the argument `-` to `cs241.binview`, which tells it to read from standard input.

```
% cs241.binasm < input.asm | cs241.binview -
```

By combining `cs241.binview` with `diff`, you can figure out exactly which binary words are different between your output and the expected output.

```
% ./asm < input.asm | cs241.binview - > my-output.txt
% cs241.binasm < input.asm | cs241.binview - > expected-output.txt
% diff my-output.txt expected-output.txt
1c1
< 00000000 01000011 00001000 00100000
---
> 00000000 01000001 00011000 00100000
3c3
< 00000000 00000000 00111000 00010100
---
> 00000000 00000000 10001000 00010100
```

This example `diff` output indicates differences on lines 1 and 3. The binary words with < beside them are from "my-output.txt", and the binary words with > beside them are from "expected-output.txt". You might also find the command `diff -y` useful, which displays the entire files side by side and marks differing lines.

Marmoset Notes

When debugging your C++ or Racket code, you may find yourself adding print statements to help trace what is going on. However, you should **remove your debug prints before submitting to Marmoset!**

Why? Well, first of all, having your code littered with debug print statements is not good style, and the code you submit to Marmoset should be high quality code that you have thoroughly tested and have confidence in. Secondly, Marmoset has a file size limit. While the correct output for all tests is within the file size limit, too much debug printing might cause you to exceed the limit, in which case you will automatically fail the test.

If you are using C++ for this assignment, Marmoset will run your programs with Valgrind, a tool that checks for memory-related errors. If Valgrind detects an error in your program, you will fail the Marmoset tests. A guide for debugging Valgrind errors is available.

Problems 1 through 10 on this assignment do not contain any secret tests, only public and release tests. For the Comprehensive Test problem, *all* marks come from secret tests.

Problems 1 through 10 account for **40%** of the Marmoset test marks, while the Comprehensive Test accounts for **60%** of the Marmoset test marks.

Writing An Assembler

In the following problems, you will implement an assembler for progressively larger subsets of MIPS assembly language.

Be sure to read the MIPS Assembly Language Specification. It should answer most questions you have about what is considered valid MIPS assembly language syntax.

The problems may be done in either Racket or C++14. See language-specific notes for each option at the end of this document.

We have provided a scanner (also called a tokenizer) for MIPS assembly language for each available language option (see language-specific notes). You should use this scanner as a starting point for your assembler.

Each problem in requires you to submit a program that reads from standard input and writes to standard output as well as standard error. **The input and output specifications are identical regardless of which language you choose.** The only difference is that you must submit the appropriate `.rkt` or `.cc` file depending on your choice of language.

For each problem, we ask you to implement support for additional instructions or features. You may submit the same assembler for all the problems. We encourage you to submit to Marmoset early. As soon as you implement support for the instructions specified by a problem, submit the current version of your assembler to Marmoset. That way, if you do not complete all of the problems before the deadline, you will still get credit for those that you did complete.

Hint: Depending on the design decisions you make in your solutions to Problems 1 and 2, you may have to restructure your code to get a working solution to Problem 3. You may have to do further restructuring for Problem 4 and onwards. Therefore, you may want to read and understand all the problems before beginning Problem 1. However, if you find this overwhelming, you may find it easier to just focus on the problems in order. The decision is yours.

Problem 1 — 17 marks of 250 (filename: `asm.rkt` or `asm.cc`)

Begin by writing an assembler that correctly translates input containing no labels and no instructions other than `.word`. You may assume that the input to your assembler contains no labels and no instructions other than `.word`.

Your assembler should never crash or leak memory, even if the input is not a valid assembly language program; it should produce an error message and return gracefully if the input is invalid. If the input is not a valid MIPS assembly language program, your assembler should print a message containing the word ERROR in all capitals to **standard error** and stop. It is good practice, but not a requirement, to embed ERROR within a meaningful error message.

If the input contains a correct MIPS assembly language program, your assembler should output the equivalent MIPS machine language to standard output.

The error checking and output requirements above apply to this and all future problems on the assignment.

Hint: there are relatively few ways in which an assembly language program can be valid (and all the valid forms are spelled out here), but many ways in which it can be invalid. You will find it much easier to write code that looks for valid input and rejects everything unexpected, rather than code that explicitly looks for all the different ways in which the input could be invalid.

Problem 2 — 17 marks of 250 (filename: `asm.rkt` or `asm.cc`)

Add support for label definitions to your assembler.

In addition, if the input is a correct MIPS assembly program, your assembler should output a symbol table: a listing of the names and values of all defined labels to standard error. The list should be printed on several lines, one line for each label in the input. Each line should consist of the label (without the trailing colon), followed by a space, followed by the value of the label (in decimal). The labels may appear in the symbol table in any order. For example, the following input:

```
begin: .word 2
middle: .word 0
```

```
.word 0
end:
```

Should print the following to stderr (but possibly with the lines reordered):

```
begin 0
middle 4
end 12
```

In handling labels, you may use any data structure or data structures you choose, but be sure to take efficiency into account.

Problem 3 — 17 marks of 250 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to allow labels to be defined and also to be used as operands. Henceforth, you no longer need to output a symbol table as in Problem 2 (although you will not be penalized if you do so).

Problem 4 — 7 marks of 250 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle `jr` and `jalr` instructions.

Problem 5 — 7 marks of 250 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle `add`, `sub`, `slt`, and `sltu` instructions.

Problem 6 — 7 marks of 250 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle `beq` and `bne` instructions with an integer or hex constant as the branch offset.

Problem 7 — 7 marks of 250 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle `beq` and `bne` instructions with a label as the branch target operand.

Problem 8 — 7 marks of 250 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle the `lis`, `mflo`, and `mfhi` instructions.

Problem 9 — 7 marks of 250 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle the `mult`, `multu`, `div`, and `divu` instructions.

Problem 10 — 7 marks of 250 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle the `sw` and `lw` instructions.

Comprehensive Test — 150 marks of 250, 150 secret marks (filename: `asm.rkt` or `asm.cc`)

For this problem, you are not required to implement any new features. At this point, your assembler should correctly translate all valid MIPS assembly language programs, and write ERROR to standard error for any input that is not a valid MIPS assembly language program.

When you are confident that your assembler is complete and correct, submit your assembler to the project **A3Comprehensive** on Marmoset. Marmoset will test it thoroughly with many large and complex programs, both valid and invalid.

The comprehensive test will cover all instructions and features from A3P1 up to A3P10. However, it is not necessary to output a symbol table to standard error as in A3P2.

The comprehensive test involves large programs, which may have as many as 100,000 lines. Programs with a small number of lines that are very long could be included as well. If your assembler is inefficient (runs in quadratic time or worse in terms of the input size) it will likely exceed Marmoset's time limit and fail some of the test cases. Other forms of inefficiency, like using too much memory, can also cause Marmoset test failures, but timeouts are the most common. Only some of the Comprehensive Test cases are large programs, and you can still get most of the marks with an inefficient assembler.

Remember to disable debugging output from your assembler before submitting to this problem. If you produce too much debugging output and exceed Marmoset's file size limit, you will automatically fail the tests.

Click here to return to the top of Assignment 3.

Language-Specific Details

Racket

The provided starter `asm_rkt.zip` has a function called `scan` that takes as input a string and returns a list of tokens.

When submitting to Marmoset, you will need to add all your files to a `.zip` file and submit that to Marmoset. The top level directory of your `.zip` file must contain the `asm.rkt` file. For example, if your zip file contains a directory called `a3` and the `asm.rkt` file is stored under this directory, Marmoset will not be able to find the file.

You can also simply pass multiple files to `marmoset_submit` and it will zip them up for you:

```
marmoset_submit cs241 A3P1 asm.rkt scanning.rkt
```

The Using Racket in CS 241 document contains hints and techniques for using Racket to write the assembler. *However, it's rather old and hasn't been updated in a while, so there may be better ways to do some of the things mentioned in it...*
See also the comments in the provided scanner.

Run a Racket program using the command: `racket asm.rkt`

Click here to return to the top of the page.

C++

The provided starter `asm_cpp.zip` has a method called `scan` that takes as input a string and returns a vector of tokens.

When submitting to Marmoset, you will need to add all your files to a `.zip` file and submit that to Marmoset. The top level directory of your `.zip` file must contain the `asm.cc` file. For example, if your zip file contains a directory called `a3` and the `asm.cc` file is stored under this directory, Marmoset will not be able to find the file.

You can also simply pass multiple files to `marmoset_submit` and it will zip them up for you:

```
marmoset_submit cs241 A3P1 asm.cc scanner.cc scanner.h
```

You are **strongly** advised to check for memory-related errors by vetting your programs with Valgrind. To do this, run "valgrind *program optionsAndArguments*" instead of just "*program optionsAndArguments*".

Marmoset will run your submissions with Valgrind as well, and will reject any submission that contains memory-related errors. Be aware that running Valgrind increases the execution time of your program by a factor of 5 to 20.

See the following page on Debugging Valgrind Errors for a discussion of common Valgrind errors and how to resolve them.

Compile a program in C++ using the command "`g++ -g -std=c++14 -Wall asm.cc scanner.cc -o asm`".

This command will create a file called `asm` containing the compiled code.

Run the program using the command: `./asm`
Or use `valgrind ./asm` to run with Valgrind.

The STL Quick Reference for CS 241 document outlines the parts of the STL most likely to be of use in CS 241.

Click here to return to the top of the page.