CSE261 Project 1: Implementing a MIPS Assembler

Due 11:59PM, Nov 12th

1. Overview

This project is to implement a MIPS (subset) ISA assembler. The assembler is the tool which converts assembly codes to a binary file. The goal of this project is to help you understand the MIPS ISA instruction set and be familiar with the principle of assemblers.

The assembler is a simplified assembler which do not support the linking process, and thus you do not need to add the symbol and relocation tables for each file. In this project, only one assemble file will be the whole program.

You should implement the assembler which can convert a subset of the instruction set shown in the following table. In addition, your assembler must handle labels for jump/branch targets, and labels for the static data section.

2. Instruction Set

The detailed information regarding instructions are in the attached MIPS green sheet page.

ADDIU	ADDU	AND	ANDI	BEQ	BNE	J
JAL	JR	LUI	LW	LA*	NOR	OR
ORI	SLTIU	SLTU	SLL	SRL	SW	SUBU

- Only instructions for unsigned operations need to be implemented. (`addu`, `addiu`, `subu`, `sltiu`, `sltu`, `sll`, `srl`)

- However, the immediate fields for certain instructions are sign extended to allow negative numbers (`addui`, `beq`, `bne`, `lw`, `sw`, `sltui`)

- Only loads and stores with 4B word need to be implemented.

- The assembler must support decimal and hexadecimal numbers (0x) for the immediate field, and `.data` section.

- The register name is always "`$n`" n is from 0 to 31.

- `la` (load address) is a pseudo instruction; it should be converted to one or two assembly instructions.

  `la $2, VAR1`: VAR1 is a label in the data section
  ➜ It should be converted to `lui` and `ori` instructions.

  `lui` $register, upper 16bit address
  `ori` $register, lower 16bit address
  If the lower 16bit address is 0x0000, the `ori` instruction is useless.

Case1) load address is 0x1000 0000

```
lui $2, 0x1000
```

Case2) load address is 0x1000 0004

```
lui $2, 0x1000
ori $2, $2, 0x0004
```

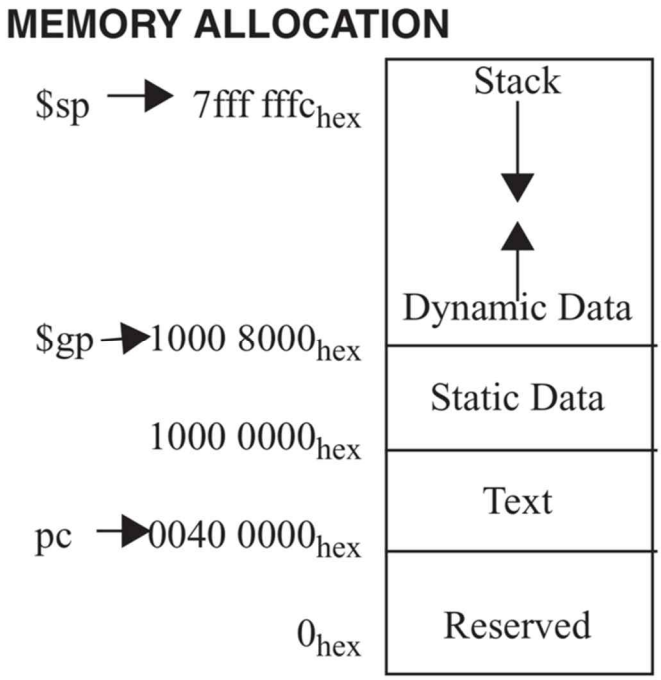
2.1 Directives

`.text`
- indicates that following items are stored in the user text segment, typically instructions
- It always starts from `0x400000`

`.data`
- indicates that following data items are stored in the data segment
- It always starts from `0x10000000`

`.word`
- store n 32-bit quantities in successive memory words

You can assume that the `.data` and `.text` directives appear only once, and the `.data` must appear before `.text` directive. Assume that each word in the data section is initialized (Each word has an initial value). In the following figure, we illustrate the memory map used in our projects.



2.2 Input format

```
1         .data
2  array:  .word   3
3          .word   123
4          .word   4346
5  array2: .word   0x11111111
6          .text
7  main:
8          addiu   $2, $0, 1024
9          addu    $3, $2, $2
10         or  $4, $3, $2
11         sll $6, $5, 16
12         addiu   $7, $6, 9999
13         subu    $8, $7, $2
14         nor $9, $4, $3
15         ori $10, $2, 255
16         srl $11, $6, 5
17         la  $4, array2
18         and $13, $11, $5
19         andi    $14, $4, 100
20         lui $17, 100
21         addiu   $2, $0, 0xa
```

Here is one of the input files we will use. As mentioned in Section 3, each input file consists of two sections, `data` and `text`. In this example, `array` and `array2` are data.

2.3 Output format

The output of the assembler is an object file. We use a simplified custom format.
- The first two words (32bits) are the size of text section, and data section.
- The next bytes are the instructions in binary. The length must be equal to the specified text section length.
- After the text section, the rest of bytes are the initial values of the data section.

The following must be the final binary format:

```
<text section size>
<data section size>
<instruction 1>
…
<instruction n>
<initial values of the data section>
```

3. Download Project1 Repository

You can download the skeleton code from the 2021-Fall-Computer-Architecture/uni{student ID}/Project1 repository to server or local machines. Then you are ready to start the project.

1) Go to the each gitlab page. The page will contain project1 directory.
2) Change directory to the location you want to clone your project and clone!
3) You will get the clone repo in your machine.

Be sure to read the `README.md` file for some useful information. It includes the explanation of each file and which files you are allowed to modify for this project.

If you do not want to use the skeleton code, it is allowed to write code from scratch. However, you are supposed to follow the input and output file format because the grading script works on the provided `sample_input` and `sample_output` files described in the following section.

4. Grading Policy

Grades will be given based on the examples provided for this project provided in the `sample_input` directory. Your assembler should print the exact same output as the files in the `sample_output` directory.

We will be automating the grading procedure by seeing if there are any difference between the files in the `sample_output` directory and the result of your simulator executions. Please make sure that your outputs are identical to the files in the `sample_output` directory.

You are encouraged to use the `diff` command to compare your outputs to the provided outputs. If there are any differences (including whitespaces) the diff program will print the different lines. If there are no differences, nothing will be printed. Furthermore, we have provided a simple checking mechanism in the `test.sh`. Executing the following command will automate the checking procedure of all test cases.

```
$ ./test.sh
```

There are 7 test codes to be graded for each correct binary code and **being "Correct" means that every digit and location is the same** to the given output of the example. If a digit is not the same, you will receive **0 score** for the example. Each test codes are [10pts] so the total points are 70pts.

5. Submission

The grading will be on the code at **Master branch** at the deadline. Any other branches are not considered as submission of the assignments. Please make sure that you can see the same result on the submission on the master branch as on your local machine.

6. Updates/Announcements

If there are any updates to the project, including additional tools/inputs/outputs, or changes, we will post a notice on the BB, and will send you an e-mail using the BB system. Please check the notice or your e-mail for any updates.

One concern is that we are using Python as an assignment language, there is plenty of packages you can leverage. Therefore, we restrict you not to import external packages. If you have question about using external package, you must email to Tas to discuss about it. Also, if you send email of using external package, you need to demonstrate "Why you have to use this package?" to TAs. If not sending email and use external packages, **we regard this behavior as a cheating**.

7. Misc

We will accept your late submissions; the deduction is 10% of each day. The minimum score is 50% of total score. Also, you have 4 **sleep days** as professor announced. Please do not give up the project.

Be aware of plagiarism! Although it is encouraged to discuss with others and refer to extra materials, copying other students or opened code is strictly banned. The TAs will compare your source code with other team's code. If you are caught, you will receive a penalty for plagiarism.

Regarding the plagiarism, please **commit frequently** so that we can know that you are really trying to solve the assignment by yourself. The only one commit with whole file changing will be suspicious of cheating.

Last semester, we found a couple of plagiarism cases through an automated tool. Please do not try to cheat TAs. If you have any requests or questions regarding administrative issues (such as late submission due to an unfortunate accident, git push or repository is not working) please send an e-mail to the TAs (tykim8191@unist.ac.kr / heelim@unist.ac.kr).