CDA 3101: Summer 2018
Project 1 - MIPS Assembler

Total Points: 100
Due: Friday 06/08/2018

1 Objective

The objective for this assignment is to make sure

- You are familiar and comfortable with the MIPS instruction set.
- You are familiar with the process of assembly - converting MIPS programs into machine instructions.
- You gain some experience in working with C.

2 Specifications

For this project, you are required to build an assembler for a subset of MIPS. Please write a C program that will read a small, simple MIPS program from standard input and prints the corresponding machine code instructions with their addresses to standard output.

2.1 The Input

You should be able to read and parse the contents of a simple MIPS assembly program, which will be redirected through standard input (please do not try to open a file in your program). You can assume that every line of the program will contain either a directive or an instruction. No lines will be blank and a label will not appear on a line by itself. You may also assume that there are no more than 100 lines in an assembly file. Lines containing directives have the following format (where square brackets indicate an optional element):

`[optional_label:]<tab>directive[<tab>operand]`

Lines containing instructions have the following format:

`[optional_label:]<tab>instruction<tab>operands`

Operands are always comma-separated with no whitespace appearing between operands.

The supported directives are listed in the following table. You may assume that the entire `.text` segment always precedes the `.data` segment. You may also assume that the memory allocation directives only ever have a single operand.

Directive	**Meaning**
.data	Indicates the start of the data section.
.text	Indicates the start of the text section.
.space n	Allocate n bytes of memory.
.word w	Allocate a word in memory and initialize with w.

The supported instructions include the following. You may wish to consult an additional MIPS reference page for more specific information about the instructions and their machine code formats.

Instruction	Type	Opcode/Funct (decimal)	Syntax
ADD	R	32	add $rd,$rs,$rt
ADDI	I	8	addi $rt,$rs,immed
SUB	R	34	sub $rd,$rs,$rt
ORI	I	13	ori $rt, $rs, immed
SLL	R	0	sll $rd, $rt, shamt
LUI	I	15	lui $rt, immed
SW	I	43	sw $rt,immed($rs)
LW	I	35	lw $rt,immed($rs)
BEQ	I	4	beq $rs,$rt,label
J	J	2	j label
LA	-	-	la $rx,label

The registers to be recognized include the following. Registers will always be expressed as $[letter][number] or $0.

Registers	Decimal Representation
$t0-$t7	8-15
$s0-$s7	16-23
$0	0

Please note the use of labels in the branch and jump instructions. You will need to calculate the appropriate immediate and targaddr fields for the machine code based on the layout of your assembly file. **Typically pseudo-direct and PC-relative addressing are relative to PC+4, not PC. That is, for architectural reasons, we define these addressing modes with the address of the next instruction, not the address of the current instruction.**

The immediate field of a branch instruction will be defined using PC-relative addressing where the address of the destination is defined as

```
Addr(dest) = Addr(inst_after_branch) + immed*4
```

That is, the immediate field represents the distance, in instructions rather than bytes, between the branching instruction and the destination instruction. Please note that here, it is assumed that you have adjusted the immediate field to be relative to PC + 1

The targaddr field of the jump instruction will be defined using pseudo-direct addressing where the address of the destination is defined as

```
Addr(dest) = Addr(jump)[31-28] || targaddr || 00
```

Where Addr(jump)[31-28] is the 4 most significant bits of the jump instruction, and || denotes concatenation.

Also note the use of the load address instruction. The load address instruction is a *MIPS pseudoinstruction* which is supported by many MIPS assemblers, but does not directly correspond to a MIPS instruction. Our assembler must replace any use of the load address instruction with a lui instruction, followed by an ori instruction. For example,

```
        .text
        la      $t0,arr
        .data
arr:    .space  50
```

should be translated to

```
        .text
        lui     $1,arr[31-16]
        ori     $t0,$1,arr[15-0]
        .data
arr:    .space  50
```

Here, arr[31-16] is the upper 16 bits of the address corresponding to label arr, and arr[15-0] is the lower 16 bits of the address corresponding to label arr, and $1 is the assembler temporary register $at (00001).

Although this is not realistic, for consistency, we will zero-out all fields which are not well defined for an instruction. These include the rs field for lui and sll as well as shamt for most R-type instructions.

Lastly, no error checking is required. You may assume that the assembly input will be correctly formed.

2.2 The Output

Lets assume the test case contains the following:

```
        .text
main:   la      $s0,arr
        lw      $t6,0($s0)
        addi    $t7,$t6,1
        sw      $t7,0($s0)
        .data
arr:    .space 4
```

Then, the output for this case should be:

```
0x000000: 0x3C010000
0x000004: 0x34300014
0x000008: 0x8E0E0000
0x00000C: 0x21CF0001
0x000010: 0xAE0F0000
```

2.3 Suggested Approach

Your assembler, just like a real assembler, will need to make two passes over the assembly file. During the first pass, the assembler should associate each symbolic label with an address. You may assume that the first instruction starts at address 0. You may also assume that memory allocations occur directly after the instructions in the process space. During the second pass, you should translate the symbolic assembly instructions into their corresponding machine code. You should print the address, followed by a space, and then the instruction. Both the address and the instruction should be in hexadecimal format and every instruction should appear on its own line.

You may find the functions fgets and sscanf to be particularly useful for this assignment. The fgets function has the following prototype:

```
char *fgets(char *str, int n, FILE *stream);
```

The fgets function reads n characters from stream and writes them to the str buffer. On success, the function returns the same str parameter. If the end-of-file is encountered, a null pointer is returned. The sscanf function has the following prototype:

```
int sscanf(const char *str, const char *format, ...);
```

The sscanf function scans the str buffer to try to match the format specifiers in the format string. Additional pointer arguments may be supplied to indicate variables that should be filled with elements found in the str buffer. You may need to consult some documentation to use these functions properly. Do not ask me what the format strings should look like this is part of the assignment! You must be able to read the documentation on these functions and figure out how to use them if you choose to do so.

You may also want to make use of some bitwise operators (&, |, <<and >>) or the union construct to manage the instruction fields easily. There is no need to "translate" decimal values of the instruction fields into binary manually they are already represented as binary numbers in memory! Also, watch out for signed-ness.

2.4 Executable and Sample Test Cases

A sample executable (to be executed on linprog) will be provided on the course website about 10 days before the project is due. A few sample test cases will also be provided. Make use of the sample executable to help verify your output. The sample executable is not guaranteed to be bug-free and there is extra credit available for students who find any errors in the sample executable. Also, please be aware that passing all of the test cases does not guarantee a perfect score.
‘

We are delaying the release of the executable to give you some time to think about the algorithm for assembly, instead of just turning it into a "match the output" game.

2.5 Testing

Part of the process of developing a good software artifact is thoroughly testing the artifact. The sample test cases, while extensive, do not cover all the possible combination of instructions you assembler could encounter.

For this project, you are required to develop your own suite of test cases. You can write anywhere between 1 (very long) and 10 (somewhat small) test assembly programs, using the reduced instruction set for this project) to ensure the quality of your assembler. The quality of the test suite will be determined by how many of our reduced set of instructions are tested and how varied the tests are. For example, testing if the assembler works for BNE for both the equal and not equal cases, and forward and backward branching, and all combinations of these, would get a better grade than just testing for one case of BNE.

3 Submission and Grading

You are required to turn in the following:

- You C program, called "proj1_LastName.c"
- A suite of test cases that you have generated, beyond the test cases provided, that demonstrate your rigorous testing of your solution.

- A README file listing all the instructions your program can handle, and all known issues with your program (parts not implemented, parts implemented with issues, etc).

Please tar your C program and your test cases into one tarball. Please make sure your tarball does not contain any other files, especially executables.

Submissions may be made through Canvas in the Assignments section. You must submit before 11:59 PM on June 8 to get full credit. Late submissions will be accepted for 10% off on June 9. The first person to report any errors in the provided materials will be given 5% extra credit. Automatic plagiarism detection software will be used on all submissions any cases detected will result in a grade of 0, reduction of the overall grade by one letter grade, and a report will be sent to Academic Affairs.

Your submitted C program will be compiled and run on `linprog` with the following commands, where `test.asm` is an assembly file, as described below:

```
$ gcc proj1.c
$ ./a.out < test.asm
```

You should not rely on any special compilation flags or other input methods.

Programs that do not compile will automatically receive a grade of 0. Every test case that results in a segmentation fault will result in a loss of 10 points.

The grade breakdown for the project is as follows:

- Reading and parsing - 10 points
- First pass over the code to generate the Symbol Table - 15 points
- Handling the directives -10 points
- Handling R Type instructions - 15 points
- Handling I Type instructions - 15 points
- Handling J Type instructions - 10 points
- Handling the LA instruction - 5 points
- Generating instructions for the data part - 10 points
- Quality of the test suite - 10 points