CDA3101 – Project 1

MIPS Assembler

Submission:

1. Create a tar file that includes your program and any test files you may want the TAs to use.
2. Make an electronic submission of your tarball file to Canvas.

Due Date:

1. September 27, 2018, 11:50pm for all students

Late Penalty:

1. 10% per day (round up)
2. Maximum two days

1. Overview

The objectives of this project include the following:

- Grasp the concepts of the MIPS instruction set;
- Get familiar with how MIPS assembly programs are translated into machine code; and
- Demonstrate your understanding of MIPS instructions and code translation via a C/C++ program.

2. Basic Description

Your job is to create a program that takes as input (from standard input) a small simplified MIPS assembly program and prints to standard output the corresponding machine code instructions and their associated addresses. Your program should be able to compile and run on `linprog` with an inputfile test.asm, which is an assembly file with MIPS instructions.

Your program should be named as `proj01`, with a proper C or C++ suffix. It should compile using the appropriate compiler, `g++` or `gcc`, as listed below.

```
$ g++ proj01.cpp -o masm
$ gcc proj01.c -o masm
```

The executable (`masm`) should read lines of MIPS instructions from `stdin`, convert the instructions to the MIPS machine language, and send the output to `stdout`. See the section on "**Sample Output**" for more information. You can also redirect the stdin to an input file as shown below.

```
$ ./masm < test.asm
```

If your program does not compile and execute as expected with the above commands, points will be deducted from your project grade. A sample executable (to be executed on linprog) is provided on canvas, as well as a sample input file. 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 the sample input file does not guarantee a perfect score – take the time to come up with your own additional test cases to verify that your program works in all scenarios.

3. Assembly Input

You should read and parse the contents of a simple MIPS assembly program, which will be redirected through standard input (i.e. 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.

a. Directives

Lines with directives have the following format (where brackets indicate an optional element):

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

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*.

b. Instructions

Lines containing Instructions have the following format:

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

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
NOR	R	39	nor $rd,$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)
BNE	I	5	bne $rs,$rt,label
J	J	2	j label
LA	-	-	la $rx,label

c. Registers and Operands

Operands are always comma-separated with no whitespace appearing between operands. Registers will always be expressed as $[letter][number] or $0. The registers to be recognized include the following.

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

d. Lables

Labels can be used 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. **Note an important detail here: 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. Here, we will simply be using PC.** 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(branch) + immed*4
```

That is, the immediate field represents the distance, *in instructions rather than bytes*, between the branching instruction and the destination instruction. 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 *pseudo-instruction* 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,

```
     la   $t0,nums
nums: .space     100
```

should be translated to

```
     lui   $1,nums[31-16]
     ori   $t0,$1,nums[15-0]
nums: .space     100
```

Where `nums[31-16]` is the upper 16 bits of the address corresponding to label `nums`, and `nums[15-0]` is the lower 16 bits of the address corresponding to label `nums`, 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.

4. Suggested Development 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 data 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.

If you program in C, 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 `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.

If you use C++, you shall find C++ equivalents or alternatives to `fgets` and `sscanf` on your own. You shall not ask what the format strings for such functions should look like. These are part of the assignment! You must be able to read the documentation on these functions and figure out these functions.

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.

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

5. Sample Output

A sample executable is provided, but here is quick run-through. Let's assume the test case test.asm contains the following:

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

The sample executable using test.asm as input yields an output for the instructions as shown below.

```
$ masm < test.asm

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

Note that the sample output does not contain the data allocation for `nums` (this is intentional, **not an error to report**). Your program does need to consider all directives and include the data allocations in the output. As mentioned earlier, you can assume that data allocation starts after the instructions.

6. Breakdown of points

The grade breakdown for the project is as follows:

Points	Requirements
5	Conformant to the required compilation and execution commands.
10	Reading and parsing all lines from the input
10	First pass to generate the symbol table

15	Handing the directives correctly
15	Handling R-type instructions correctly
10	Handling I-type instructions correctly
10	Handing J-type instructions correctly
15	Handing the LA instruction correctly
10	Generating instructions for data allocation

7. Miscellaneous

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 for those involved.