

CDA3101 Project 1: MIPS Assembler

Due 6/6

I. Purpose

The primary purpose of this project is to help cement the concepts of the MIPS instruction set and give you insight into how MIPS assembly programs are translated into machine code. The secondary purpose of this project is to give you some experience with C programming.

II. Basic Description

Your job is to create a program, contained in a single C file called **proj1.c**, which 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 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. 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 the course website, as well as some test cases. 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 – take the time to come up with your own additional test cases to verify that your program works in all scenarios.

III. Assembly Input File

You should be able to 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. Lines containing directives have the following format (where 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 <i>n</i>	Allocate <i>n</i> bytes of memory.
.word <i>w</i>	Allocate a word in memory and initialize with <i>w</i> .

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

The registers to be recognized include the following.

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

Registers will always be expressed as \$[letter][number] or \$0.

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. **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

$$\text{Addr}(\text{dest}) = \text{Addr}(\text{branch}) + \text{immed} * 4$$

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

$$\text{Addr}(\text{dest}) = \text{Addr}(\text{jump})[31-28] \parallel \text{targaddr} \parallel 00$$

Where $\text{Addr}(\text{jump})[31-28]$ is the 4 most significant bits of the jump instruction, and \parallel 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, _a
_a:    .space    100
```

should be translated to

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

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

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

V. Sample Output

A sample executable is provided, but here is quick run-through. Let’s assume the test case `test2.asm` contains the following:

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

Then running the sample executable with test2.asm as input yields the following output.

```
> ./proj1_linprog < tests/test2.asm
0x000000: 0x3C010000
0x000004: 0x34300014
0x000008: 0x8E0E0000
0x00000C: 0x21CF0001
0x000010: 0xAE0F0000
```

VI. Miscellaneous

Submissions may be made through Blackboard in the Assignments section. You must submit before **11:59 PM** on **June 6** to receive full credit. Late submissions will be accepted for 10% off each day they are late, for up to two days. 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.