CIS 600 Computer Architecture, Fall 2022
Project 1 – MIPS Assembler

1. Objectives

This project is designed to help students to understand the RISC architecture (MIPS) and its instruction set and assembly. Students will gain programming experience in C.

2. Goals

Your team (2 persons) will build an MIPS assembler for a subset of MIPS instructions in C. This assembler will read a simple MIPS program and generate an MIPS machine code output file. Due dates for Project 1 is **November 3**.

3. Specifications

3.1 Input

Your assembler will read and parse the contents of a simple MIPS program (program.asm). Each line of the program contains an MIPS instruction or a directive. A label will appear on a line by itself. Operands are comma-separated. **No line is blank. No comment is allowed. There will be no white spaces between operands. All lines will begin with a tab except labels. Only decimal numbers are allowed.** Lines containing instructions have the following format:

    <tab>instruction<tab>comma- separated-operands

Lines containing directives have the following format:

    <tab>directive[<tab>comma-separated-operand]

Lines containing labels have the following format:

    label:

The supported instructions are as follows. You may wish to consult additional MIPS references or the textbook to know the details of the instructions.

Instruction	Name	Syntax	Semantic
ADD	Addition	add $1,$2,$3	$1 = $2 + $3
SUB	Subtract	sub $1,$2,$3	$1 = $2 - $3
SLL	Shift left logical	sll $1,$2,5	$1 = $2 << 5
SRL	Shift right logical	srl $1,$2,5	$1 = $2 >> 5
SLT	Set less than	slt $1,$2,$3	If $2<$3, $1=1; otherwise, $1=0
ADDI	Addition immediate	addi $1,$2,45	$1 = $2 + 45
LUI	Load upper immediate	lui $1,45	Upper 16-bit of $1 = 45 (Lower 16-bit of $1 is set to 0)
ORI	Or immediate	ori $1,$2,45	$1 = $2 | 45 (bitwise OR)
LW	Load word	lw $1,100($2)	$1 = Memory[$2+100]
SW	Store word	sw $1,100($2)	Memory[$2+100] = $1
BEQ	Branch on equal	beq $1,$2,Label	If $1=$2, jump to Label
BNE	Branch on not equal	bne $1,$2,Label	If $1≠$2, jump to Label

J	Jump	j Label	Jump to Label
LA *	Load address	la $1,Label	lui $1, upper 16-bit of Label ori $1, $1, lower 16-bit of Label

* LA is a pseudo-instruction. It is used to load the memory location (Label, 32 bits) into the destination register. The assembler replaces it with two instruction sequence, lui followed by ori.

The supported directives are as follows.

Directive	Name	Syntax	Semantic
.data	Data segment	.data (no operand)	Data section in memory; it begins at 0 (0x0000 0000) by default
.text	Text segment	.text (no operand)	Program section in memory; it begins at 512 (0x0000 0200) by default
.space *n*	Allocation of *n* words of memory	.space 10	Allocation of 10 words (40 bytes) of memory
.word *w*	Allocation of a word and initialized to *w*	. word 16	Allocation of a word (4 bytes) and initialized to 16 (0x0001 0000).

* The program may contain data section only or text section only. If both exist, the data section proceeds the text section.

The registers are denoted as $0, $1, $2, etc. instead of "$s0" or "r1". $0 always takes the value of 0 and is not changeable.

3.2 Output

The assembler generates an output file (program.out) of size 1KB consisting of 512B of data segment (begins at 0x0000) and 512B of text segment (begins at 0x0200). Since each data and instruction is 4B long, there will be at maximum 128 word data and at maximum 128 instruction words. **This file is a binary file. Do not try to open this file as it causes an error.** The assembler does not detect syntax errors and assumes the assembly input is correctly formed. We assume big-endian.

For a sample assembly input file (test.asm):

```
        .data
print_data:
        .word 0
add_result:
        .word   0
load_data:
        .word   1
        .word   2
        .text
start:
        la      $1,load_data
        lw      $4,0($1)
        lw      $5,4($1)
        add     $4,$4,$5
```

```
la      $1,add_result
sw      $4,0 ($1)
```

Then, the contents of the output file (test.out) will be as follows. **<u>Again, this is a binary file and can only be seen with a special tool.</u>**

- In Linux, you will have to use the command line command "od -Ax -t x4" (octal dump). To see the text segment (offset 0x0200), please use the command "od -Ax -t x4 -j512".
- In Windows, you can use "format-hex." Click the Start menu button and type "powershell" (without the quotation marks). In the "Windows PowerShell" window, please use the command line command "format-hex".
- You can also view it online: https://hexed.it/

```
00000000 00000000 00000001 00000002
00000000 00000000 00000000 00000000
..............
3c010000 34210008 8c240000 8c250004
00852020 3c010000 34210004 ac240000
..............
```

Note that the file begins with the data segment (512 bytes) followed by the text segment. "`3c010000 34210008`" denotes the assembled output for the LA instruction (LUI and ORI). "`8c240000`" denotes the LW instruction. And so on.

Encoding of MIPS instructions is as follow. You may wish to consult additional MIPS references or the textbook to know the details of the instructions.

Instruc-tion	Syntax	MIPS instruction encoding (32 bits)					
	(R-type)	Opcode(6)	Rs(5)	Rt(5)	Rd(5)	Shamt(5)	Func(6)
ADD	add $1,$2,$3	000000				N/A	100000
SUB	sub $1,$2,$3	000000				N/A	100010
SLL *	sll $1,$2,5	000000	N/A				000000
SRL **	srl $1,$2,5	000000	N/A				000010
SLT	slt $1,$2,$3	000000					101010
	(I-type)	Opcode(6)	Rs(5)	Rt(5)	Immediate (16)		
ADDI	addi $1,$2,45	001000					
LUI	lui $1,45	001111	N/A				
ORI	ori $1,$2,45	001101					
LW	lw $1,100($2)	100011					
SW	sw $1,100($2)	101011					
BEQ	beq $1,$2,Label	000100					
BNE	bne $1,$2,Label	000101					
	(J-type)	Opcode(6)	Offset(26)				
J	j Label	000010					

Please note the use of labels in the branch (beq and bne) instructions. You will need to compute the appropriate immediate fields for the machine code based on the following relationship.

Addr(Label) = Addr(inst_after_branch) + immediate(16 bits)*4

First, the immediate field represents the distance, in instructions rather than in bytes, between the branching instruction and the destination instruction. This explains *4 at the end of the relationship. Second, typically PC-relative addressing is relative to PC+4, not PC. That is, it is relative to the next instruction, not the current instruction. This explains "Addr(inst_after_branch)" in the relationship.

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

Addr(Label) = Addr(inst_after_jump)[31-28] || immediate(26 bits) || 00

where Addr(inst_after_jump)[31-28] is the 4 most significant bits of PC+4, and || denotes concatenation.

3.3 Useful standard C library functions for this project
(ref: https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/rtref/stalib.htm**)**

You may want to use file-related C functions such as fopen(), fclose(), and fwrite() and string-related C functions such as strncmp(), strtok(), and atoi(). Please make yourself familiar with those functions first.

You may find another string-related C functions fgets() to be particularly useful for this assignment. 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 following example shows the usage of fgets() function:

```
#include <stdio.h>

int main () {
   FILE *fp;
   char str[60];

   /* opening file for reading */
   fp = fopen("file.txt" , "r");
   if(fp == NULL) {
      perror("Error opening file");
      return(-1);
   }
   if( fgets (str, 60, fp)!=NULL ) {
      /* writing content to stdout */
      puts(str);
   }
   fclose(fp);

   return(0);
}
```

The sscanf function would be also useful. It 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. The following example shows the usage of sscanf() function:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main () {
   int day, year;
   char weekday[20], month[20], dtm[100];

   strcpy( dtm, "Saturday March 25 1989" );
   sscanf( dtm, "%s %s %d  %d", weekday, month, &day, &year );

   printf("%s %d, %d = %s\n", month, day, year, weekday );

   return(0);
}
```

The fwrite function writes a raw data to a file. On success, the function returns the count of the number of items successfully written to the file. On error, it returns a number less than the specified number of items to be written. The following example shows the usage of fwrite() function:

```
#include<stdio.h>

int main () {
   FILE *fp;
   int year;

   fp = fopen( "file.txt" , "w" );
   if (fwrite(&year , size(int), 1, fp )!=1) {
         perror("Error writing file");
         return(-1);
   }

   fclose(fp);

   return(0);
}
```

You may also want to make use of some bitwise operators (&, |, <>) or the union construct to manage the instruction fields easily. Also, watch out for signed-ness.

3.4 Testing

A few sample test assembly files will also be provided about a week before the project is due. Make use of the sample executable to help verify your output.

You can test your design using your own programs. The quality of your assembler will be determined by how much 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.

To confirm your assembler generates the correct machine code, you can get some help from Internet, e.g., https://www.csfieldguide.org.nz/en/interactives/mips-assembler/ or https://alanhogan.com/asu/assembler.php. Just type your assembly program such as “add $s1, $s2, $s3”, it will output machine code for you. (Note that this assembler recognizes $s1, $s2, etc. but does not recognize $1, $2, etc.)

4. Submission and Grading

Each group submits via Blackboard the source code: assembler.c.
At the top of the source code, please list CLEARLY all the instructions and directives your program **cannot** handle and **all known issues** with your program (those that are not implemented, those that are implemented but not work correctly, etc.).

You must submit before 11:59 PM on November 3 to get full credit. Late submissions will be accepted for 10% off per day up to three additional days. **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.