Instructions: All assignments are due by **midnight** on the due date specified.

Programming assignment should be completed in C.

Please present your solutions in a clean, understandable manner.

You must complete this program individually.

Assembler for a Subset of the MIPS Assembly Language

Objective:

Practice the basics of the assembly process, which includes encoding instructions and replacing symbolic labels with appropriate offsets.

Your goal is to write an assembler for a subset of the MIPS instruction set. It should read the assembly file from standard input and write the machine code to standard output. Below are the MIPS assembly directives that you are required to recognize.

Directive	Explanation
.text	Switch to the text (code) segment.
.data	Switch to the (static) data segment.
.word $w1, w2, ..., wn$	Store n 32-bit integer values in successive memory words.
.space n	Allocate n words that are initialized to zero in the data segment.

Below are the instructions you have to recognize. The R-format instructions can have a different number of arguments depending upon the type of instruction. You can find the encoding for each instruction in the MIPS sheet provided on Canvas.

Mneumonic	Format	Args	Description
addiu	I		add immediate without overflow
addu	R	3 (rd,rs,rt)	addition without overflow
and	R	3 (rd,rs,rt)	bitwise AND operation
beq	I		branch on equal
bne	I		branch on not equal
div	R	2 (rs,rt)	signed integer divide operation
j	J		unconditional jump
lw	I		load 32-bit word
mfhi	R	1 (rd)	move from hi register
mflo	R	1 (rd)	move from lo register
mult	R	2 (rs,rt)	signed integer multiply operation
or	R	3 (rd,rs,rt)	bitwise OR operation
slt	R	3 (rd,rs,rt)	set less than
subu	R	3 (rd,rs,rt)	subtraction without overflow
sw	I		store 32-bit word
syscall	R	0	system call

You can assume that an assembly file will have instructions preceding data and the general format of a file will be as follows:

```
.text
<instructions>
.data
<data declarations>
```

Each instruction or data word can have a symbolic label. Each assembly line can also have a comment that starts with the # character, which indicates a comment until the end of the line.

You can assume that the maximum number of instructions is 32768 and the maximum number of data words is also 32768. You can also assume the maximum length of an assembly line is 80 characters and the maximum size of a symbolic label is 10 characters. Finally, you can assume there are no whitespace (blank or tab) characters between the arguments in each assembly instruction.

Below is an example assembly file that reads a number n,reads n values, and then prints the sum of those n values.

```
        .text
        addu    $s0,$zero,$zero         # s0 = 0
        addu    $s1,$zero,$zero         # s1 = 0
        addiu   $v0,$zero,5             # v0 = read value cmd
        syscall                         # v0 = read()
        sw      $v0,n($gp)              # M[n] = v0
L1:     lw      $s2,n($gp)              # s2 = M[n]
        slt     $t0,$s1,$s2             # if s1 >= s2 then
        beq     $t0,$zero,L2            #    goto L2
        addiu   $v0,$zero,5             # v0 = read value cmd
        syscall                         # v0 = read()
        addu    $s0,$s0,$v0             # s0 += v0
        addiu   $s1,$s1,1               # s1 += 1
        j       L1                      # goto L1
L2:     addu    $a0,$s0,$zero           # a0 = s0
        addiu   $v0,$zero,1             # v0 = print value cmd
        syscall                         # print(a0)
        addiu   $v0,$zero,10            # v0 = exit cmd
        syscall                         # exit()
        .data
n:      .word   0
```

Symbolic labels can be referenced as the target for transfers of controls (branch or jump instructions) or as an offset from the global pointer ($\$gp$). Your assembler will require two passes, where the second pass resolves forward references to labels, such as L2 or n in the previous example, to determine the proper displacement value. References to data labels are at a displacement from the global pointer register, $\$gp$, which in our assembler is assumed to point to the beginning of the data segment.

The first line of the machine code file contains an object file header that consists of the number of instructions and the number of words of data, written as decimal values. The next set of lines consists of hexadecimal values representing the encoding of the machine instructions in the

text segment of the assembly instructions. The final set of lines consists of hexadecimal values representing the initial values of the words in the data segment.

Below is the machine code file produced from the assembly code on the previous page.

```
18 1
00008021
00008821
24020005
0000000c
af820000
8f920000
0232402a
11000006
24020005
0000000c
02028021
26310001
08000005
02002021
24020001
0000000c
2402000a
0000000c
00000000
```

If you log into one of the on-domain Linux computers, or into one of the remote Tech servers (such as guardian.it.mtu.edu), you can run a sample assembler by typing:

```
/classes/cs3421/assem.exe < name.asm > name.obj
```

at a Linux terminal where `name.asm` should be replaced by the assembly file, and `name.obj` should be replaced by the name of the output file you'd like to use. When you complete your program, it should provide the exact same output as this sample assembler.

Helpful C Review

There are some features in C that will be helpful for implementing this assignment. The $fgets$ function reads a line of characters from *stream* into s until a newline or end-of-file is reached or $n-1$ characters have been read without encountering a newline or end-of-file character. $fgets$ returns a null pointer if end-of-file is encountered before any other characters are read.

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

The $sscanf$ function is similar to $scanf$, but reads from the string s.The value returned from $sscanf$ is the number of successful assignments before a terminating null character is read from the string or there is a conflict between the control string and a character read from s.

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

You should include $< stdio.h >$ to obtain the proper prototypes of $fgets$ and $sscanf$. For instance, below is an example code fragment that reads a line and checks for a 3-address R-type instruction.

```
#include <stdio.h>
...
#define MAXLINE 80
```

```
#define MAXNAME 10
#define MAXREG   5
...
char line[MAXLINE], oper[MAXNAME], rs[MAXREG], rt[MAXREG], rd[MAXREG], *s;
...
while (fgets(line, MAXLINE, stdin)) {
    s=skiplabel(line);
    ...
    /* check if a 3-address R format instruction */
    else if (sscanf(s, "%10s $%5[^,],$%5[^,],$%5s", oper, rd, rs, rt) == 4) {
    ...
    }
}
```

Other features that you may find useful are C bit fields and unions. C allows the width in bits of integer fields to be specified. A union is similar to a C struct, except that the fields overlap in memory. Below is an example union that has 3 fields (f,x, and FPS) that overlap in space, which allow a float value to be read and printed as an unsigned integer or the three components of the IEEE floating-point standard (FPS). Note that the order in which the bit fields should be listed will depend on if the processor is little-endian (as is *guardian*) or big-endian.

```
union {
    float f;
    unsigned int x;
    struct {
        unsigned int F:23;
        unsigned int E:8;
        unsigned int S:1;
    } FPS;
} u;

scanf("%f", &u.f);
printf("Hexadecimal representation: %08x\n", u.x);
printf("FPS: S=%u, E=%u, F=%u\n", u.FPS.S, u.FPS.E, u.FPS.F);
```

Submission Requirements

Your submission must be written in C.

Use Canvas to submit a tar file named `hw2.tgz` that contains:

- A copy of the source **with comments**.
- A makefile with *all*, *assem*, and *clean* rules (will be provided).
- A README file explaining any parts of the program that doesn't work.

When I execute the commands: `tar zxf hw2.tgz; make` against your submission, an executable named `assem` should be created in the current directory.