Honors Project Description

The honors project is to write an assembler for a simple dialect of MIPS assembly language. When your assembler assembles a program, it should generate a simple "executable" file, meaning that the file contains information about (some of) the global variables of the program and contains the machine code for each instruction in the program. The generated executable file can then be executed using the simulated MIPS CPU that was built for the class project.

Note that in the real world, the assembler generates .o file which contain machine code that is relocatable and linkable with other .o files and libraries (including dynamic libraries that are linked at run time). Your assembler will simply generate a single executable file with a .m extension that is neither linkable nor relocatable, thus simplifying the construction of the assembler.

The Executable Format

The format of the executable file (with a .m extension) that your assembler should generate is described in the file loader.h as well as here. The executable file is a binary file that should be created using a call to fopen() and which should be written to using the fwrite() procedure declared in the usual stdio.h library header file. Do not use printf() or any other character-oriented output function to write to the file.

The format of the file is given below. Note that a 32-bit delimiter value, 0xf0f0f0f0, must appear at the start of the file, after the data section of the file, and after the code section of the file (i.e. at the end of the file). This use of a delimiter is primarily as a simple check that the the file is of the correct format (i.e. not some random file) and has not been corrupted.

The file must contain the following, in exactly the given sequence.

Header

- 4 bytes: Delimiter Value (0xf0f0f0f0)
- 4 bytes: Total size in bytes of the data part (see below) in the file. This is not how many bytes are occupied by global variables in the program, just the number of bytes in the data part of the file itself.
- 4 bytes: Number of instructions (not bytes) in code section of the file. This is the same as the number of MIPS instructions in the assembly language program being assembled.

Data Part

This part contains a series of entries, each entry corresponding to a global variable for which an initial value was specified. Note that a global variable for which no initial value was specified will not have a corresponding entry in this part.

Each entry for a global variable with an initial value contains the following:

- 4 bytes: the address of the global variable in memory.
- 4 bytes: byte_count, the number of bytes of the initial value of the global variable (i.e. the size of the next field of the entry, see below).
- byte_count bytes: The initial value of the global variable.

Delimiter

- 4 bytes: Delimiter Value (0xf0f0f0f0)

Code Part

A sequence of 4-byte (32-bit) instructions. The number of instructions is specified in the header, above)

Delimiter

- 4 bytes: Delimiter Value (0xf0f0f0f0)

That is all that the executable file should contain.

To make clear what the data part of the executable file should contain, suppose that the assembly language program contains the following global variable declarations.

```
        .data

x:      .long 100     //4-bytes with an initial value of 100

y:      .long        //4-bytes, but with no initial value

z:      .ascii   "Hello World"      //12-bytes, including the terminating 0 value,
                                    // with the initial value "Hello World"
```

The data part of the executable file should contain entries for just x and z, since y does not have an initial value. The data part should look like (in binary):

<address_of_x>
4
100
<address_of_z>
12
Hello World0

where <address_of_x> and <address_of_z> are the addresses of x and z, respectively, the 4 and the 12 correspond to the number of bytes in the initial values for x and z, respectively, and the 100 and Hello World0 correspond to the initial values of x and z, respectively. The Hello World0 is just a sequence of bytes representing the characters 'H', 'e', ..., 'd' and the value 0.

Also, since the total number of bytes in the above data part of the file is 32 (just add up the number of bytes of each element, above), the second element of the header section of the executable file will be 32.

Assembly Language

Each MIPS instruction in the assembly language program being assembled will be translated into a 32-bit machine code instruction. You already know the format of each machine code instruction, having implemented the simulated MIPS CPU. Here I define the assembly language format for each instruction that your assembler will have to be able to translate into machine code.

Operands in an assembly language instruction.

All instructions have either registers and/or immediate values as their operands (except for syscall which has no operands). The format of these elements are as follows:

Registers: A register is specified by a $ followed by either a number between 0 and 31 or an alternative name as given in the table below.

Name	Register	Description
at	$1	assembler temporary
v0	$2	values for function returns and expression evaluation
v1	$3	
a0	$4	function arguments
a1	$5	
a2	$6	
a3	$7	
t0	$8	temporaries (caller-saved)
t1	$9	
t2	$10	
t3	$11	
t4	$12	
t5	$13	
t6	$14	
t7	$15	
s0	$16	saved temporaries (callee-saved)
s1	$17	
s2	$18	
s3	$19	
s4	$20	
s5	$21	
s6	$22	
s7	$23	
t8	$24	temporaries (caller-saved)
t9	$25	
k0	$26	reserved for OS kernel
k1	$27	
gp	$28	global pointer
sp	$29	stack pointer
fp	$30	frame pointer
ra	$31	return address

Note that you do not need to worry about how the above registers are used. You only need to be able to translate the alternative name into the correct register number. Furthermore, note that register names are case insensitive, so that $v0 and $V0 refer to the same register.

Immediate Values: An immediate value is specified as a one of the following:

- a numeric literal in decimal (e.g. 100),
- a numeric literal in hexidecimal (e.g. 0x45),
- a label reference (e.g. x or DONE), or

- a shift expression involving a numeric literal or label reference and a constant shift amount (e.g. 1 << 10 or FOO >> 16).

A label reference may refer to a label in the data segment (e.g. referencing a global variable) or in the code segment (e.g. in a jump or branch instruction). Your assembler should translate the label reference into a numeric value corresponding to the label. For example, if your assembler enounters the assembly language instruction

beq $t1,$t2,DONE

then the value of the 16-bit immediate field in the machine code instruction should be a number that causes the PC to be set to the address corresponding to the label DONE if the branch is taken.

A shift expression must be evaluated by the assembler and replaced, in the machine code, by the corresponding value. For example, if FOO is a label corresponding to some 32-bit address, then the FOO >> 16 is the value of that address shifted to the right by 16 – that is, the 16-bit immediate value containing the upper 16 bits of the address. This is generally used when using the LUI instruction to load the upper 16 bits of a 32-bit address into a register.

There are three categories of instructions, R-instructions, I-instructions, and J-instructions. Within each category, every instruction has the same format, as described below.

R-Instructions

As you know, R-instructions have the following machine-code format, from left to right (most significant bit to least significant bit):

- opcode field: 6 bits (NOTE: for all R-instructions, opcode = 0)
- RS field: 5 bits (specifies one of the registers)
- RT field: 5 bits (specifies one of the registers)
- RD field: 5 bits (specifies one of the registers)
- shamt field: 5 bits (specifies a shift amount in a shift instruction)
- funct field: 6 bits (specifies which R-instruction to execute)

The table below describes the assembly language format of each R-instruction that your assembler must translate into the above machine code format. In the assembly language version of each instruction, the operands *rd*, *rs* and *rt* denote the occurrence of a register name (e.g. $23 or $a0). The operand *shamt* denotes a constant numeric literal (e.g. 5).

Remember, the opcode for R-instructions is always 0.

Instruction	Assembly Language Format	Funct
add	add *rd,rs,rt*	0x20
addu	addu *rd,rs,rt*	0x21
sub	sub *rd,rs,rt*	0x22
subu	subu *rd,rs,rt*	0x23
mult	mult *rs,rt*	0x18
multu	multu *rs,rt*	0x19
div	div *rs,rt*	0x1a

Instruction	Assembly Language Format	Funct
divu	divu *rs*,*rt*	0x1b
mfhi	mfhi *rd*	0x10
mflo	mflo *rd*	0x12
and	and *rd*,*rs*,*rt*	0x24
or	or *rd*,*rs*,*rt*	0x25
xor	xor *rd*,*rs*,*rt*	0x26
nor	nor *rd*,*rs*,*rt*	0x27
slt	slt *rd*,*rs*,*rt*	0x2a
sltu	sltu *rd*,*rs*,*rt*	0x2b
sll	sll *rd*,*rt*,*shamt*	0x00
sllv	sllv *rd*,*rs*,*rt*	0x04
srl	srl *rd*,*rt*,*shamt*	0x02
srlv	srlv *rd*,*rs*,*rt*	0x06
sra	sra *rd*,*rt*,*shamt*	0x03
srav	srav *rd*,*rs*,*rt*	0x07
jr	jr *rs*	0x08
jalr	jalr *rs*	0x09
syscall	syscall	0x0c

I-instructions

I-instructions have the following machine code format, from left to right (most significant bit to least significant bit):

- opcode field: 6 bits (specifies which instruction to execute)
- RS field: 5 bits (specifies one of the registers)
- RT field: 5 bits (specifies one of the registers)
- immediate field: 16 bits (specifies a constant value to be used)

The table below describes the assembly language format of each I-instruction that your assembler must translate into the above machine code format. The *imm* operand, as described above, may be a numeric literal, a label, or a shift expression. Also as described above, if the instruction is a branch instruction (beq or bne) and the *imm* operand is a label, then the immediate value should be computed by the assembler as the value needed to branch to the corresponding label in the program.

Instruction	Assembly Language Format	opcode
addi	addi *rt*,*rs*,*imm*	0x08
addiu	addiu *rt*,*rs*,*imm*	0x09
lw	lw *rt*,*rs*,*imm*	0x23
lh	lh *rt*,*rs*,*imm*	0x21
lhu	lhu *rt*,*rs*,*imm*	0x25
lb	lb *rt*,*rs*,*imm*	0x20
lbu	lbu *rt*,*rs*,*imm*	0x24
sw	sw *rt*,*rs*,*imm*	0x2b
sh	sh *rt*,*rs*,*imm*	0x29
sb	sb *rt*,*rs*,*imm*	0x28
lui	lui *rt*,*imm*	0x0f
ori	ori *rt*,*rs*,*imm*	0x0d

Instruction	Assembly Language Format	opcode
andi	ani *rt,rs,imm*	0x0c
xori	xori *rt,rs,imm*	0x0e
slti	slti *rt,rs,imm*	0x0a
sltiu	sltiu *rt,rs,imm*	0x0b
beq	beq *rt,rs,imm*	0x04
bne	bne *rt,rs,imm*	0x05

J-instructions

J-instructions have the following machine-code format.

- opcode field: 6 bits (specifies which instruction to execute)
- address field: 26 bits (used in specifying the target address of the jump)

The table below describes the assembly language format of each I-instruction that your assembler must translate into the above machine code format. The *address* operand is an immediate value which, as described above, may be a numeric literal, a label, or a shift expression. If it is a label, then the immediate value should be computed by the assembler as the value needed to jump to the corresponding label in the program.

Instr	Assembly Language Format	Opcode
j	j *address*	0x02
jal	jal *address*	0x03

Assembler Directives

The simple assembler directives that your assembler will need to handle are primarily those that you are familiar with from the programming assignments earlier in the semester. The directives are as follows:

`.text`

indicates that the following elements (e.g. instructions and labels) relate to the code section.

`.data`

indicates that the following elements relate to the data section. Note that both `.text` and `.data` may appear multiple times in the assembly file.

`.long` *val*

indicates the allocation of a word (32 bits, 4 bytes) in memory. *val* is optional and, if included, specifies the value to be written into the allocated word. val may be of the form of a numeric literal in decimal or hex.

`.byte` *val*

indicates the allocation of a byte in memory. *val* is optional and, if included, specifies the value to be written into the allocated bye. val may be of the form of a numeric literal, in decimal or hex, or an ascii character of the form `'c'` (i.e. a single character in single quotes).

`.short` *val*

indicates the allocation of a half-word (16 bits, 2 bytes) in memory. *val* is optional and, if included, specifies the value to be written into the allocated half-word. val may be of the form of a numeric literal in decimal or hex.

`.space` *size*

indicates the allocation of *size* bytes in memory.

`.ascii` *string*

Indicates the allocation of sufficient space to hold *string*, including an implicit zero in the last byte of the string. string must enclosed in double quotes. For example, the directive

```
.ascii "Hello World"
```

should allocate 12 bytes of memory (including the zero in the last byte) and initialize those bytes to the "Hello World" string (again, including a zero after the 'd').

Labels

A label must appear at the beginning of a line (although whitespace before the label is allowed) and be terminated by a ':' (colon). The label may occupy the line on its own or may be followed by an instruction or memory directive (e.g. .long, .short, etc.). A label simply represents the current address in the data or code section.

Comments

A comment starts with // or # and includes the rest of the line.

Implementing the Assembler

You will need to implement the assembler in a file named assembler.c . Here are some suggestions that may help you:

The main() procedure in assembler.c should take a single command-line parameter, which gives the name of the assembly language file (with a .s extension) to assemble. Your code should open that file to read the assembly language program from the file as well as open a file with the same prefix, but with a .o extension, for writing the executable output to. For example, if someone runs your assembler program by typing

```
assem myfile.s
```

Your program should generate the executable file `myfile.m` .

You should use the constants DATA_STARTING_ADDRESS, and CODE_STARTING_ADDRESS declared in cpu.h and DELIMITER declared in loader.h .

As your assembler is reading through a data section (following a .data directive), it will need to keep track internally of the "current" address in the data segment of memory. Initially, the

current data address should be DATA_STARTING_ADDRESS. Each time the assembler sees a directive to allocate memory, it should increment the current data address to reflect the size specified by the directive. When it encounters a label, it should associate the label with the current data address in some internal data structure (table, linked list, etc.). For example, if the assembler sees

```
        .data
x:      .long 100
y:      .space 20
z:      .ascii   "Hello World"
```

then, assuming this is the first occurrence of `.data` in the program, since the data section starts at DATA_STARTING_ADDRESS, the address associated with the label `x` will be DATA_STARTING_ADDRESS, the address associated with `y` will be DATA_STARTING_ADDRESS+4, and the address associated with `z` will be DATA_STARTING_ADDRESS+24. When the assembler sees a reference to one of these labels in an instruction, it should replace the reference with the corresponding address.

As your assembler is reading through a code section (following a .text directive), it will also need to keep track internally of the "current" address in the code segment of memory. Initially, the current code address is CODE_STARTING_ADDRESS. Each time the assembler sees an instruction, it should increment the current code address by four to reflect the size of the instruction. When it encounters a label, it should associate the label with the current code address in some internal data structure. For example, if the assembler sees

```
        .text
FAC_REC:
        bne     $a0, $0, RECURSIVE
        addi    $v0, $0, 1
        j       DONE
RECURSIVE:
        addiu   sp,sp,-8
```

then, assuming this is the first occurrence of `.text` in the program, since the code section starts at CODE_STARTING_ADDRESS, the address associated with the label `FAC_REC` will be CODE_STARTING_ADDRESS and the address associated with `RECURSIVE` will be CODE_STARTING_ADDRESS+12. When the assembler sees a reference to one of these labels in an instruction, it should replace the reference with a value that, depending on the instruction, causes the address of the label to be used appropriately.

For example, each of the following instructions reference a label, but the immediate field generated by the assembler will be slightly different:

```
beq     $v1,$v4,FOO

j   BAR

ori $v0,$v0,BAZ
```

In the first instruction (`beq` ...), the 16-bit immediate value *imm* generated for the reference to `FOO` should be such that *current_address+4+*(*imm* << 2*)* evaluates to the address denoted by the label `FOO`, where *current_address* is the address of the instruction. In the second instruction (`j` ...), the 26-bit immediate value *imm* generated for the reference to `BAR` should be such that (*imm* << 2) evaluates to the address denoted by the label `BAR` (recall how the `j` instruction uses its operand – and ignore the issue of the leftmost 4 bits). Finally, in the third instruction (`ori` ...), the

immediate value for the reference to `BAZ` should just be the lowest 16 bits of the address denoted by the label `BAZ`.

Important: Notice that a reference to a label, such as the reference to `RECURSIVE` in the above code example, may appear before the label itself (allowing a a jump forward in the code). Thus, your assembler will need to perform two passes: The first to figure out the address associated with each label and the second to generate machine code with appropriate numeric values for the label references appearing in the instructions.