



project_2_spec

Project 2 EECS 370 (Winter 2024)

Worth:	100 Points	Point Allocation
Assigned:	Friday, February 9th	
Part 2A Due:	11:55 PM ET, Thursday, February 22nd	40 Points
Part 2L Due:	11:55 PM ET, Thursday, March 21st	60 Points

0. Starter Code

For Project 2A, the assembler, you have 2 choices: build off your project 1a assembler OR start with the [starter code](#), which will be updated after all project 1a submissions have been collected. For project 2L, the LC2K linker [starter code](#) is meant to help you read in and parse object files. It is probably a good idea to break it up into different functions, but is a good place to get started.

1. Purpose

The purpose of this project is to help you understand the assembling and linking process, which we can utilize to create multi-file LC2K programs. In order to do this, we will first create a new assembler (P2A) which will take an assembly file as input and output an intermediate object file. Our linker (P2L) will take object file(s) as input and create the final machine code.

In Project 1a, you wrote an assembler which took an LC2K assembly file as input and produced an executable file as output. This approach is fine if all the code needed is contained in one file, but what happens if we want to use other pieces of code? Libraries contain functions that make coding easier, and are often written in assembly and stored as object files. Splitting code into multiple files encourages modularity and organization. Multiple files are also important for large projects: if you modify one source file, you only need to recompile and reassemble that one and then link everything together, *greatly* reducing the total time to create an executable. Now that we have a better understanding of translation software, we can create a separate assembler and linker.

Here is an example that will help explain the purpose of the linker:

```

main.as

1      lw      0      1      five      ; $1 = 5
2      lw      0      4      SubAdr     ; Store address of SubAdr
3  start  jalr      4      7              ; Store return address and jump to Su
4      beq      0      1      done      ; Finish if $1 == 0
5      beq      0      0      start     ; Otherwise continue (keep calling Su
6  done   halt
7  five   .fill     5

```

```

subOne.as

1  subOne  lw      0      2      neg1     ; $2 = -1
2          add      1      2      1      ; $1 = $1 - 1
3          jalr      7      6              ; Jump back to where we were called f
4  neg1    .fill     -1
5  SubAdr  .fill     subOne              ; Define where our function definitio

```

Here, we have two basic programs. `main.as` loads `$1` with `5`. We then call the `subOne` function. We then decrement the value of `5` and return to `main.as` until our result is `0`. Even though `main.as` tries to call `subOne.as` without knowing at assemble time where it is defined (since it is in a separate file), our program will still work.

The linked result would look like this:

Note: The linker will take in object files and produce a machine code file with the linking process. Assembly files are neither an input nor output. This is just an example of how linking would look visually.

```

count5.as

1      lw      0      1      five      ; $1 = 5
2      lw      0      4      SubAdr     ; Store address of SubAdr
3  start  jalr      4      7              ; Store return address and jump to Su
4      beq      0      1      done      ; Finish if $1 == 0
5      beq      0      0      start     ; Otherwise continue (keep calling Su
6  done   halt
7  subOne  lw      0      2      neg1     ; $2 = -1
8          add      1      2      1      ; $1 = $1 - 1
9          jalr      7      6              ; Jump back to where we were called f
10 five   .fill     5
11 neg1    .fill     -1
12 SubAdr  .fill     subOne              ; Define where our function definitio

```

2. Problem

```
[[assembly files]] -assembler--> [[object files]] -linker--> [executable]
```

This project has two parts. In the [first part](#), you will create a program that assembles an assembly file into an object file. The P2A assembler is an extension of the P1A assembler. The key distinction for P2A is that instead of outputting a machine code (`mc`) file, you will output an object (`obj`) file which contains additional information to assist in the linking process: **a header, a symbol table, and a relocation table**. In the [second part](#), you will write a program to link object files into a **single executable consisting of machine code**, which your project 1 simulator will be able to run.

3. Assembler (40 points)

Your new assembler will take in a single assembly file (see [section 3.1](#)) as input and output a single object file (see [section 3.2](#)).

So far we have created an assembler which can translate assembly language into machine code. However, let's consider a basic program that prints "Hello World":

helloWorld.c

```
1  #include <stdio.h>
2
3  int main(){
4      printf("Hello World");
5      return 0;
6  }
7
```

If we were to compile this into assembly, we would need to branch to the `printf()` function and execute the code at that memory location. This is great because we don't need to rewrite `printf` every time we create a new project, we can just `#include <stdio.h>`. However, our current assembler can't handle *undefined references*. To fix this, we are going to create an assembler that allows for external references (i.e. references to labels that are NOT defined in the file) and a program called the `linker` to resolve those undefined external references.

3.1. Assembly Files

3.1.1 Assembly File Format

Assembly language programs will be of the same format as those from Project 1, with a few extra restrictions.

The first part of the assembly file must contain only assembly instructions. The second part should contain only `.fill` assembler directives. For example, suppose an assembly file is composed of M instructions and N `.fill` s. Lines 0 to $(M-1)$ contain actual instructions, and lines M to $(M+N-1)$ contain `.fill` s, with no mixing between them. We refer to all of our instructions as belonging to the `Text` section of our program. Moreover, everything that contains a `.fill` statement is considered to be in the `Data` section of our program. It is important that all of your test cases separate these two sections such that no `.fill` directives are in the `Text` section and no instructions are in the `Data` section. Below the data section is the `Stack`, which is initially empty; for an instruction to access the stack, e.g load a word from the stack, we will use the `label Stack` to denote the start of the stack section.

3.1.2 Local and Global Labels

LC2K files may now use global symbolic addresses, which means we must now distinguish between local and global labels. The scope of a local label is the file the label is defined in. (This is analagous to a variable or function with the `static` keyword in C. The scope of a local variable in C is at most a function.) The scope of a global label is all object files linked together (more on this in [part 2I](#)). Because of this, different object files can use local labels with the same name and still be linked together. Local labels will start with a lowercase letter `[a, b, ..., z]` while global labels start with a capital letter `[A, B, ..., Z]`. This is unique to LC2K as a way to distinguish between local and global labels. For example, `staddr` is a local label whereas `Staddr` is a global label.

Local symbolic addresses must be defined at assembly time. However, a global symbolic address can be undefined at assembly time. It is assumed that undefined global labels are defined in another file to be resolved at link time, so they should be temporarily resolved as address 0 in the text and data segments. Defined symbolic addresses should be resolved exactly as they were in Project 1. That is, it is entirely possible that a global label is defined and referenced in the same file; if this is the case, the label should be resolved just like a local label. The `Stack` label should be treated as an undefined global label for the purposes of the assembler.

Just like P1A, you can assume assembly files max out at 65536 total instructions and data, although we'll test you on much, much less than that. As suggested in the starter code, you may assume that no input LC2K file is more than 1000 lines.

3.1.3 LC2K Peculiarities Part 1

Firstly, if a `beq` instruction contains a symbolic address, the label it refers to must be a locally defined label. This label can be either a local or global label. A `beq` should not branch to another file, and a programmer should use `jalr` in this case.

Secondly, in LC2K, loading or storing to an absolute address no longer makes much sense. The locations of data and text within the final executable file will likely be different than in the original object file, leading to unintended execution. While this isn't something we will enforce with error checking, it is recommended that labels are used when dealing with loads and stores. In reality, there are reasons to use absolute addressing: memory mapped IO for example (if you're curious about this, take EECS 373 **shameless plug**) or cache analysis (see Project 4). If you come across a label with a constant offset, assemble as in Project 1.

Thirdly, local labels should not be included in the symbol table. However, a local symbolic address does need a relocation table entry as the address of the local label might change. These addresses can be fixed by calculating the new local label location during linking.

3.1.4 Summary

In summary, assembly file formatting rules are:

1. Do not mix instructions with directives (`.fill s`)
2. Instructions come first
3. Directives (`.fill s`) come second
4. Defined symbolic addresses (defined local and global labels) are resolved exactly as they were in the Project 1 assembler
5. Undefined global symbolic addresses are temporarily resolved as address 0
6. Local labels start with `a...z` and must be defined at assembly
7. Global labels start with `A...Z` and can be undefined at assembly
8. Branches cannot use undefined global symbolic addresses

3.2 Object File Format

Object files will contain the following sections in the following order:

- Header
- Text
- Data
- Symbol table
- Relocation table

*** Refer to lecture, lab, and walkthrough for a detailed explanation of each section. ***

Table 1: Object file sections

Section Name	Number of lines	Description
--------------	-----------------	-------------

Section Name	Number of lines	Description
Header	Fixed: 1	The Header contains the size, in lines, of the sections to follow. Sizes are listed in the following order, each separated by a space: Text , Data , Symbol table, Relocation table.
Text	Variable: t t = # of instr.	Each line in the Text segment consists of a single machine code instruction, assembled in the same way as instructions in Project 1.
Data	Variable: d d = # of .fills	The Data segment contains data stored by assembler directives, one word of data per line.
Symbol table	Variable: s s = # of locally-defined global labels + # of Unresolved global symbolic addresses	Each line in the Symbol table consists of a global label, one letter (T/D/U) corresponding to Text , Data , and Undefined respectively, and a line offset from the start of the T/D section (0 if the letter was 'U'). Each value separated by a space, in that order. Each symbol should only appear once in the symbol table, even if it is used multiple Times. Entries can appear in any order.
Relocation	Variable: r r = # of uses of symbolic addresses excluding beq	Each line in the Relocation table consists of a line offset from the start of the T/D section (whichever section the symbol was used in), an opcode, and a label. Each separated by a space, in that order. Entries can appear in any order.

Consider the example:

main.as					
1	lw	0	1	five	; \$1 = 5
2	lw	0	4	SubAdr	; Store address of SubAdr
3	start jalr	4	7		; Store return address and jump to Su
4	beq	0	1	done	; Finish if \$1 == 0
5	beq	0	0	start	; Otherwise continue (keep calling Su
6	done halt				
7	five .fill	5			
8					

The following symbol table is produced:

SubAdr is added to the symbol table as it is a global label used as an argument

The following relocation table is produced:

```
1  0 lw five
2  1 lw SubAdr
```

`0 lw five` is added to the relocation table as the memory address of `five` will be changed during linking. `1 lw SubAdr` is added to the relocation table as the memory address of `SubAdr` will be determined during linking.

Note: We don't add `beq` instructions to the relocation table as those can only branch to text inside that same file and since `beq` is PC relative, no update is needed.

IMPORTANT FORMATTING NOTES:

1. Assembly code in text should be assembled EXACTLY as it was for project 1. This means symbolic addresses are resolved the same, with the exception of undefined global symbolic addresses which are temporarily assembled as 0.
2. Offsets in the Symbol and Relocation Tables indicate the line offset of the label from the start of either the Text or the Data section (whichever section the label is defined in for the symbol table, and whichever section the instruction or `.fill` appears in for the relocation table).

For example, the symbol table entry `Foo D 0` indicates the label `Foo` is defined on the zeroth line in the Data section. The relocation table entry `4 lw Foo` indicates the symbolic address `Foo` is used on the fourth line (zero indexed) of the Text section by a `lw` instruction.

3.3 Error Checking

Your assembler should catch the following errors in assembly files:

- Use of undefined local symbolic addresses
- `beq` using an undefined symbolic address
- Duplicate definition of labels
- `offsetFields` that don't fit in 16 bits
- Unrecognized opcodes
- Illegal register operands (i.e. not an integer or not within `[0,7]`)

Your assembler should `exit(1)` if it detects an error and `exit(0)` if it finishes without detecting any errors. Your assembler should NOT catch simulation time or link time errors, i.e. errors that would occur at the time the assembly-language program is linked or executed (e.g. branching to

address -1, infinite loops, etc.).

3.4 Assembly example

Please see [section 9 example 2a](#).

3.5 Running Your Assembler

Write your program to take two command-line arguments. The first argument is the filename where the assembly-language program is stored, and the second argument is the filename where the output (the object file) will be written. For example, with a program name of `assembler`, an assembly-language program in `program.as`, the following would generate an object file `program.obj`:

```
./assembler program.as program.obj
```

Note that the format for running the assembler must use command-line arguments for the file names (rather than standard input and standard output). Your program should store only object files in the format specified above. Any deviation from this format (e.g. extra spaces or empty lines) will render your object file ungradable. Any other output that you want the program to generate (e.g. debugging output) can be printed to standard output or standard error.

3.6 Using the Instructor Project 1a Solution

The instructor project 1a solution, released after the final student submission of project 1a, is a compiled object file (`inst_p1a_obj.<target>.o`) of the C code solution from project 1a, with modifications. The instructor solution does not check for undefined labels of any type, and instead resolves them to 0. Otherwise, all machine code is correctly translated and all other project 1a error checks will exit correctly. After downloading the starter code and renaming `starter_assembler.c` to `assembler.c`, you can compile `assembler.c` and link with `inst_p1a_obj.o` using `gcc assembler.c inst_p1a_obj.o`, and run the executable on an LC2K assembly code file.

If you choose to use the instructor object file, beware of the following particulars: First, this object file is only officially supported on CAEN systems since it was compiled on a CAEN system similar to the autograder servers. If you are not already in the habit of testing your projects on CAEN, it is recommended to follow [setup guide](#) from EECS 280. Also, this object file does not contain debug information so that instructors can protect the instructor project 1a solution. This means that debugging with a visual debugger or GDB will only show you what is happening in your `assembler.c`. It also does not print anything to stdout, so print debugging will not be as helpful as if you use your project 1a code. Finally, if you choose to use the starter `Makefile`, make sure

to change the compilation dependencies in there to link with the instructor project 1a solution.

3.7 Test Cases

The test cases for the assembler part of this project will be short assembly-language programs that serve as input to an assembler. You will submit your suite of test cases together with your assembler, and we will grade your test suite according to how thoroughly it exercises an assembler. Each test case may be at most 50 lines long, and your test suite may contain up to 20 test cases. These limits are much larger than those needed for full credit. See [section 7](#) for how your test suite will be graded.

i Hint: The example assembly-language program (Example 2a) in [section 9](#) is a good case to include in your test suite, though you'll need to write more test cases to fully test your code and receive credit on the autograder. Remember to create some test cases that test the ability of an assembler to check for the errors in [section 3.3](#).

4. Linker (60 points)

Now that you've written an assembler to create object files, you need a way to link these files together. In this part of the project, you will write a linker to combine multiple object files into a single executable. This final executable can be run with the simulator from project 1.

4.1 LC2K linker description

Your linker should be able to take an arbitrary number (between 1 and 6) of object files as input. It will concatenate all text and data segments within each object file, creating one unified executable. Segments should be combined in the order they appear as arguments. The combined text section should be placed before the combined data section. Then, for each object file, the linker iterates through their relocation table. For each relocation entry, the linker determines the location of the label in the combined file and fixes the reference. The final executable will be a machine code file.

4.2 What about `main()` ?

You might be asking yourself, what will be executed first? Shouldn't there be a `main()` function or label?

To simplify the process of linking and simulating, LC2K code is executed starting at the first line in a machine code file (memory address 0). In order to specify what object file should execute first,

ordering of the linker's arguments is needed. This means that our `main` will be the first file provided to the linker.

```
./linker file_0.obj file_1.obj ... file_N.obj machine_code.mc
```

The single executable from the above command will be laid out according to the diagram below. This is assuming we are linking N files together, where `file_0` is the first file passed into our linker and `file_N` is the last file passed into our linker.

```
1  _____ machine_code.mc _____
2      <file_0.obj> TEXT
3      <file_1.obj> TEXT
4      .
5      .
6      .
7      <file_N.obj> TEXT
8      <file_0.obj> DATA
9      <file_1.obj> DATA
10     .
11     .
12     .
13     <file_N.obj> DATA
```

For more information on the linker's command line arguments, please see [section 4.7](#). For more information on how linkers actually handle this, see [section 4.4](#).

4.3 Stack Label

As discussed in lecture, programs build up stack frames as they execute. The stack is important for storing data that can't fit within a machine's registers, such as a function's local data. As seen in the below example, this is done in LC2K by using a global label `Stack`.

Here is a small LC2K program that uses a subroutine call. It takes an argument in register 1 and calls a subroutine to compute the quantity `4*input`. Register 1 is used to pass input to the subroutine; register 3 is used by the subroutine to pass the result back. The current top-of-stack (first empty location) is given by `Stack` + the contents of register 5.

main.as - Line highlighted jumps to the code in file sub4n.as

1	lw	0	1	input	\$1 = memory[input]
2	lw	0	4	SubAdr	prepare to call sub4n. \$4 = addr(sub
3	jlr	4	7		call sub4n; \$7 = return address; \$3
4	halt				
5	input	.fill	10		
6					

sub4n.as - Lines highlighted are addressing the stack

1	sub4n	lw	0	6	pos1	r6 = 1
2		sw	5	7	Stack	save return address on stack
3		add	5	6	5	increment stack pointer
4		sw	5	1	Stack	save input on stack
5		add	5	6	5	increment stack pointer
6		add	1	1	1	compute 2*input
7		add	1	1	3	compute 4*input into return value
8		lw	0	6	neg1	r6 = -1
9		add	5	6	5	decrement stack pointer
10		lw	5	1	Stack	recover original input
11		add	5	6	5	decrement stack pointer
12		lw	5	7	Stack	recover original return address
13		jalr	7	4		return. r4 is not restored.
14	pos1	.fill	1			
15	neg1	.fill	-1			
16	SubAdr	.fill	sub4n			contains the address of sub4n
17						

In LC2K, the `Stack` label is a special label inserted by the linker that should not be defined by any object file, but it can be used as a symbolic address. The stack array starts at the implicit label `Stack` and extends to larger addresses, which is why the linker automatically resolves the `Stack` label beyond the text and data segments in the final executable. For example, if there are M instructions and N pieces of data in the final executable, the linker should resolve the symbolic address, `Stack`, as $(M + N)$. This allows the stack to grow to higher addresses without affecting the instructions or data.

4.4 LC2K Peculiarities Part 2

Programming languages often specify where to begin executing. In reality, a linker typically inserts an object file into the linking process. This inserted code appears first and jumps to a specified function (`main`) to begin executing the program, among doing other things. The LC2K method of ordering files during the linking process to indicate what to execute first is a simplification.

LC2K also lacks an instruction that jumps to labels while saving the return address. Instead, `jalr` jumps to registers that hold function addresses (so the register is a function pointer). This means that a function can have a local label, yet still be accessible from other files, so long as the function pointer is global. Linking should still succeed in this case.

Additionally, LC2K's use of the `Stack` label doesn't reflect how all assembly languages use the stack. ARMv7, for example, has special instructions such as `push` and `pop` that directly interface

with the stack, providing a layer of abstraction to assembly programmers. The stack is typically allocated by an operating system that passes the stack pointer to an executing program.

4.5 Error Checking

Your linker should catch the following errors:

- Duplicate defined global labels
- Undefined global labels
- `stack` label defined by an object file

Your linker can assume that any object file used as input is properly formatted.

4.6 Tip - Local Labels

Fixing local symbolic addresses during linking can be tricky, since we don't have symbol table entries associated with them. It might help to store certain data for each file read in: text size, data size, text starting location (in final mc), and data starting location (in final mc). By also storing which file each relocation table entry is in, you should have all the data needed to adjust each local symbolic address.

Actually fixing a local symbolic address in the relocation involves several steps. First, identify which section of the file the label is in, either text or data. Second, parse the original symbolic address value from the instruction referenced by the relocation entry. Fix this value by adding an offset to the address, to account for the new location of the local label.

4.7 Linker Example

Please see [section 9 example 2I](#).

4.8 Running Your Linker

Write your program to take N command-line arguments, where $N \geq 2$. The first argument is the object file to execute first, arguments 2 through $N-1$ are additional object files (these are not required), and the N th argument is the filename where the machine code will be written. For example, with a program name of `linker` and an assembly-language program in `prog_1.obj` and `prog_2.obj`, the machine code file `prog.mc` will be generated with this command:

```
./linker prog_1.obj prog_2.obj prog.mc
```

Do note that you can expect your linker to be tested on linking more than two files together. If you want to test linking with more than two files, supply more than two object files as arguments.

The number of object files your linker must be able to link together is between 1 and 6. If a program is self-contained within one object file, your linker should still be able to translate it into a machine code file. We will not test you on linking more than 6 object files.

Note that the format for running the linker must use command-line arguments for file names (rather than standard input and standard output). Your program should store only machine code in the format specified above. Any deviation from this format (e.g. extra spaces or empty lines) will render your machine code file ungradable. Any other output that you want the program to generate (e.g. debugging output) can be printed to standard output or standard error.

4.9 Test Cases

Test cases for the linker part of this project will be short, valid assembly-language programs that, after being assembled into object files, serve as input to a linker. You will submit a suite of test cases together with your linker, and we will grade your test suite according to how thoroughly it exercises an LC2K linker. Each test assembly file may be at most 50 lines long, and your test suite may contain up to 20 test cases. A test can contain no more than 6 assembly files to be linked together. These limits are much larger than needed for full credit. See [Section 7](#) for how your test suite will be graded.

A naming scheme is needed to specify what test assembly files should be linked together. A single “test” refers to a group of 1 or more assembly files to be linked together. The naming scheme is as follows.

```
<test name>_<{0, ..., N}>.as
```

All tests with the same `<test name>` will be assembled and linked together (do not include angled brackets or spaces in the test name). An underscore character, ‘_’, separates the test name and the assembly file’s number (do not include angled brackets or curly brackets in the number). Assembly files within the same test should be numbered starting at zero, with the zeroth assembly file being the first code to be executed.

The following testcases:

```
test_0.as test_1.as test_2.as anotherTest_0.as anotherTest_1.as
```

Will be assembled and then linked by the autograder as follows:

```
1 ./linker test_0.obj test_1.obj test_2.obj test.mc
2 ./linker anotherTest_0.obj anotherTest_1.obj anotherTest.mc
```

! DO NOT use more than one underscore in your test case names. We will not grade your test case if you do. File names CANNOT have spaces in them, or any character besides letters, numbers, 1 underscore, and 1 period.

i Remember to create some test cases that test the ability of a linker to check for the errors in [Section 4.5](#).

5. Compiling the Project

Your code will be compiled with the GCC compiler using the C99 standard. The following bash command compiles `program.c` and writes the executable into `program`. You are allowed to use any standard C libraries which compile with the specified flags below.

```
gcc -std=c99 <programName>.c -o <programExecutableName>
```

6. Grading, Auto-Grading, and Formatting

We will grade primarily on functionality, including error handling, correct assembly, and comprehensiveness of the test suites.

To help you validate your project, your submission will be graded automatically, and the result will be available on the autograder. You may then continue to work on the project and re-submit. To deter you from using the autograder as a debugger, you will receive feedback from the autograder only for the first **THREE SUBMISSIONS** for each project part on any given day. All subsequent submissions will be silently graded (this means the submission will be graded, but you will not have access to the grade nor the results of your submission). Your final score will be derived from your overall *best submission* to the autograder, including silent submissions.

The feedback from the autograder will not be very illuminating; it won't tell you where your problem is or give you the test programs. The purpose of the autograder is to let you know that you should keep working on your project (rather than thinking it's perfect and ending up with a 0). The best way to debug your program is to generate your own test cases, figure out the correct answers, and compare your program's output to the correct answer. This is also one of the best ways to learn the concepts in the project.

The student suites of test cases will be graded according to how thoroughly they test both the assembler (for [part 2a](#)) and linker (for [part 2l](#)). We will judge thoroughness of the test suites by how well they expose potential bugs. That is, the test suites are graded based on how many of the buggy assemblers / linkers were exposed by at least one test case. This is known as "mutation testing" in the research literature on automated testing.

For the assembler test suite, the auto-grader will use each test case as input to a set of buggy

assemblers. A test case exposes a buggy assembler by causing it to generate a different answer from a correct assembler. Your test suite is run on **12** buggy assemblers. To receive all Mutation Testing points **A total of 4 points**, your test suite must expose at least **10/12** of the buggy assemblers.

For the linker test suite, the auto-grader will first assemble the test files and use them as input to a set of buggy linkers. A test case exposes a buggy linker by causing it to generate a different answer from a correct linker. Test cases must use the naming scheme specified in [section 4.9](#). Your test suite is run on **10** buggy linkers. To receive all Mutation Testing points **A total of 7 points**, your test suite must expose at least **7/10** of the buggy linkers.

7. Turning in the Project

Use autograder.io to submit your files. You have been added as a student to the class, so you should see EECS 370 listed as a class.

Here are the files you should submit for each project part:

1. assembler (part 2a)
 - a. C program for your assembler called "assembler.c"
 - b. Suite of test cases. (Each test case is an assembly-language program in a separate file, ending in `.as`, `.s`, or `.lc2k`. Test case names should ****on** include letters, numbers, underscores, and periods.)
2. linker (part 2l)
 - a. C program for your linker called "linker.c"
 - b. Suite of test cases (each test case is a set of assembly-language programs using the naming scheme specified in [section 4.9]).

8. Sample Test Cases

Example 2a

Here is a multi-file assembly-language program that counts down from 5, stopping when it hits 0, and then halts:

main.as :

1	lw	0	1	five	; \$1 = 5
2	lw	0	4	SubAdr	; Store address of SubAdr
3	start jalr	4	7		; Store return address and jump to Su
4	beq	0	1	done	; Finish if \$1 == 0

```

5         beq      0      0      start      ; Otherwise continue (keep calling Su
6 done    halt
7 five    .fill    5

```

subone.as :

```

1 subOne  lw      0      2      neg1      ; $2 = -1
2         add      1      2      1      ; $1 = $1 - 1
3         jalr     7      6
4 neg1    .fill    -1
5 SubAdr  .fill    subOne      ; Define where our function definitio

```

And here are the corresponding object files after running the following lines of code:

```

1 ./assembler main.as main.obj
2 ./assembler subone.as subone.obj

```

⚠ WARNING: Text within parentheses SHOULD NOT be included in your assembler's or linker's output. There also should not be any trailing spaces or tabs. This text is added here only to help students identify the different sections of the object files in these examples.

main.obj :

```

1  6 1 1 2      (Header)
2  8454150      (Text)
3  8650752
4  23527424
5  16842753
6  16842749
7  25165824
8  5            (Data)
9  SubAdr U 0    (Symbol Table)
10 0 lw five     (Relocation Table)
11 1 lw SubAdr
12

```

subone.obj :


```

1  3 2 1 2      (Header)
2  8519683      (Text)
3  655361
4  25034752
5  -1          (Data)
6  0

```



```
7 SubAdr D 1 (Symbol Table)
8 0 lw neg1 (Relocation Table)
9 1 .fill subOne
10
```

 **WARNING:** Be careful when copying and editing these examples!

Example 2l


This example uses the object files from example 2a. Here is the machine code produced after the linking process:

```
./linker main.obj subone.obj count5.mc
```

count5.mc :

```
1 8454153 (main.as TEXT)
2 8650763
3 23527424
4 16842753
5 16842749
6 25165824
7 8519690 (subone.as TEXT)
8 655361
9 25034752
10 5 (main.as DATA)
11 -1 (subone.as DATA)
12 6
13
```

This code can be simulated using your project 1 simulator.

 **WARNING:** Be careful when copying and editing these examples!