Project 1 EECS 370 (Winter 2026)

Worth:	100 points
Assigned:	Thursday, January 8th, 2026
Part 1a Due:	11:55 PM ET, Thursday, January 29th, 2026
Part 1s & 1m Due:	11:55 PM ET, Thursday, February 5th, 2026

0. Starter Code

starter_1a.tar.gz files	Description
Makefile	Makefile to compile the project
spec.as	Spec test case assembly file
spec.mc.correct	Correct machine code output for spec test case
starter_assembler.c	Starter code for the LC-2K assembler

starter_1s.tar.gz files	Description
Makefile	Makefile to compile the project
spec.mc	Spec test case machine code file, this is the same as spec.mc.correct from P1A
spec.out.correct	Correct output for spec test case - note that your simulator should write to standard out
starter_simulator.c	Starter code for the LC-2K simulator

There is no starter code for project 1M, the assembly multiplication program.

Feel free to use `wget` and `tar` as follows:

```
Try clicking the line numbers to copy terminal commands!
1 $ wget https://eecs370.github.io/project_1_spec/starter_1a.tar.gz
2 Saving to: ‘starter_1a.tar.gz’
```

```
starter_1a.tar.gz 100% [==============>]
$ tar -xvzf starter_1a.tar.gz
starter_1a/
starter_1a/spec.as
starter_1a/spec.mc.correct
starter_1a/Makefile
starter_1a/starter_assembler.c
```

terminal

```
$ wget https://eecs370.github.io/project_1_spec/starter_1s.tar.gz
Saving to: ‘starter_1s.tar.gz’
starter_1s.tar.gz 100% [==============>]
$ tar -xvzf starter_1s.tar.gz
starter_1s/
starter_1s/spec.mc
starter_1s/spec.out.correct
starter_1s/Makefile
starter_1s/starter_simulator.c
```

1. Purpose

This is a 3 part project where you will be coding the following:

Project	Description	Required File(s) for Submission
1A - The LC2K Assembler	For project 1A, you will write a c program which takes as input an LC2K assembly file (denoted with `*.as`) and outputs its correct machine code representation into a machine code file (denoted with `*.mc`)	assembler.c, and a suite of test assembly files ending in `*.as` to be ran against your assembler, and buggy instructor assemblers
1S - The LC2K Simulator	For project 1S, you will write a c program which simulates the LC2K ISA, with a given machine code file as input. It will output the simulation to `stdout`	simulator.c, and a suite of test assembly files ending in `*.as`. These test files will first be assembled by the instructor assembler, and then ran against your simulator, and buggy instructor simulators.

Project	Description	Required File(s) for Submission
1M - LC2K Assembly Multiplication	For project 1M you will write an LC2K assembly program which multiplies two positive 15 bit numbers.	mult.as

⚠ ***Pro tip:*** *LC2K assembly files (`*.as`) and LC2K machine code files (`*.mc`) are plain-text files, meaning you should be able to edit and view them in a text editor.*

LC2K assembly files can also use the (`*.s`) and (`*.lc2k`) file extensions. This is helpful for students who use XCode and cannot open (`*.as`) files

2. LC-2K Instruction Set Architecture

Before we dive into project specifics, it is important to understand the LC2K (Little Computer 2000) Instruction Set Architecture. As for this and several of the future projects, you will be gradually "building" out the LC-2K toolchain and LC-2K simulators. The LC-2K instruction set is very simple, but it is general enough to solve complex problems. To complete project 1's three parts, you will need to only know the LC-2K Instruction Set Architecture.

Important facts about the LC-2K ISA:

- There are 8 registers (registers 0 through 7)
- Each address is 32-bits
- Each address stores a word (a word is 4 bytes which is also 32 bits)
- LC-2K has 65536 words of memory
- By assembly-language convention register 0 always has a value of 0
  - This is technically not enforced, but no assembly language program should change register 0 from its initial value of 0.

In general, an **instruction set architecture** defines how a programmer can use the processor, and what operations the processor supports.

The LC-2K ISA is a RISC architecture (*Reduced Instruction Set Computer*): This means that it supports simpler operations. Note that the ISA defines both the assembly language and the machine code. An **assembly language** is a low level programming language that closely relates to the underlying **machine code**. Each line of assembly code can be assembled into 1 line of **machine**

code, which looks like a bunch of numbers. The **machine code** is a representation of assembly code, which is usable by the computer.

The machine code file contains the actual values stored in memory (that is, the assembled assembly code). Specifically we assume that the first line of the machine code file represents the 0th address. Our assembly language also supports the use of **symbolic links**, and **assembler directives**. These higher-level operations specify how the *assembler* should handle the input assembly language and are not visible in the machine code translation after assembly.

2.1. Description of LC-2K Instructions

Assembly language name for instruction	Instruction Opcode in binary	Action
`add` (R-type instruction)	0b000	Add contents of `regA` with contents of `regB`, store results in `destReg`.
`nor` (R-type instruction)	0b001	Nor contents of `regA` with contents of `regB`, store results in `destReg`. This is a bitwise nor; each bit is treated independently.
`lw` (I-type instruction)	0b010	“Load Word”; Load `regB` from memory. Memory address is formed by adding `offsetField` with the contents of `regA`. Behavior is defined only for memory addresses in the range [0, 65535].
`sw` (I-type instruction)	0b011	“Store Word”; Store `regB` into memory. Memory address is formed by adding `offsetField` with the contents of `regA`. Behavior is defined only for memory addresses in the range [0, 65535].
`beq` (I-type instruction)	0b100	“Branch if equal” If the contents of `regA` and `regB` are the same, then branch to the address `PC+1+offsetField`, where `PC` is the address of this beq instruction.
`jalr` (J-type instruction)	0b101	“Jump and Link Register”; **First** store the value `PC+1` into `regB`, where `PC` is the address where this `jalr` instruction is defined. **Then** branch (set PC) to the address contained in `regA`. **Note** that this implies if

Assembly language name for instruction	Instruction Opcode in binary	Action
		`regA` and `regB` refer to the same register, the net effect will be jumping to `PC+1`.
`halt` (O-type instruction)	0b110	Increment the `PC` (as with all instructions), then halt the machine (let the simulator notice that the machine halted).
`noop` (O-type instruction)	0b111	“No Operation (pronounced no op)” Do nothing besides update the PC.

2.2. Description of LC-2K assembly language

An LC-2K assembly file (`*.as`) is made up of multiple lines of assembly. Each line represents the assembly intended to be stored at that address. For example, the first line of the assembly file represents what is going to go in address 0. the second line of the assembly file is what goes in address 1 and so on.

An assembly file needs to be assembled into a machine code file before it is executed by an LC-2K simulator.

An LC-2K machine code file (`*.mc`) is made up of multiple lines of integers. Each integer in the machine code file represents the value stored at that address in memory; the first line of the machine code file represents the value of address 0 when the program begins

2.2.1. LC-2K assembly language syntax

Each line of LC-2K assembly is formatted in the following way:

```
label whitespace opcode whitespace field0 whitespace field1 whitespace field2 whitespace comment
```

Each line of assembly may have the following fields:

Field	Description	Required (Y/N)
label	The leftmost field on a line is the label field. Valid labels contain a maximum of 6 characters and can consist of letters and numbers (but must start with a letter). The label is optional (but the a line without a label must have whitespace before the opcode). Labels make it much easier to write assembly-language programs. Without labels you would need to modify all numeric address fields each time you added a line to your assembly-language program! Labels that appear in the `label` field are considered 'defined'	N
opcode	The opcode field has one of eight LC-2K opcodes (Ex: `add` or `nor`), it can also have directives for the assembler (Ex: `.fill`), see section on LC-2K Directive	Y
field0	Depending on the instruction type, field0 is ignored, or is a register.	Depends on instruction type
field1	Depending on the instruction type, field1 is ignored, or is a register.	Depends on instruction type
field2	Depending on the instruction type, field2 is ignored, is a register, a numeric address, or a symbolic address (represented by a label).	Depends on instruction type
comment	The comment field is ignored	N

2.2.2. LC-2K assembly language instruction types

Here are the instruction types, with a description of the associated fields for each instruction type. Fields that are not required are ignored by the assembler:

Instruction Type	Instructions in category	Description of required fields
R-Type Instructions	`add`, `nor`	`opcode`, `field0`, `field1`, and `field2` are required fields: `field0` is a register (regA) `field1` is a register (regB) `field2` is a register (destReg)

Instruction Type	Instructions in category	Description of required fields
I-Type instructions	`lw`, `sw`, `beq`	`opcode`, `field0`, `field1` and `field2` are required fields: `field0` is a register (regA) `field1` is a register (regB) `field2` is either a numeric address, or a symbolic address (represented by a label)
J-Type instructions	`jalr`	`opcode`, `field0`, and `field1` are required fields: `field0` is a register (regA) `field1` is a register (regB)
O-Type instructions	`noop`, `halt`	Only the `opcode` field is required

2.2.3. LC-2K assembler directives

In addition to LC-2K instructions, an assembly-language program may contain directions for the assembler:

- The only assembler directive we will use is `.fill` (note the leading period).
- The `.fill` assembler directive tells the assembler to put an integer into the place where the instruction would normally be stored.
- `.fill` instructions use one field, which can be either a numeric value or a symbolic address.

For example, `.fill 32` puts the value 32 where the instruction would normally be stored. `.fill` with a symbolic address will store the address of the label.

spec.as - .fill using symbolic address is highlighted

```
1          lw      0       1       five    load reg1 with 5 (symbolic address) -
   note that this instruction is at address 0
2          lw      1       2       3       load reg2 with -1 (numeric address)
3  start   add     1       2       1       decrement reg1
4          beq     0       1       2       goto end of program when reg1==0
5          beq     0       0       start   go back to the beginning of the loop
6          noop
7  done    halt                            end of program
8  five    .fill   5
9  neg1    .fill   -1
10 stAddr  .fill   start                   will contain the address of start (2)
```

```
11
```

In the spec example, `.fill start` will store the value 2, because the label `start` is at address 2. The bounds of the numeric value for `.fill` instructions are to (-2147483648 to 2147483647).

2.2.4. LC-2K symbolic addresses and labels

I-Type instructions and the `.fill` directive can use **defined** labels as arguments. Remember that labels are used to “book-mark” lines of assembly. They provide a way of symbolically indicating a line of assembly (which in turn represents an address) without using its numeric value. They are incredibly useful when doing assembly programming. Remember, in our assembly files we assume that the first instruction is at address 0.

- When a label is used inplace of a numeric address in `field2` for I-Type instructions, we say that the instruction is using a symbolic address. (The address it refers to is not static, but is instead wherever that label is defined).
  - When used with `lw` or `sw` instruction, a label indicates you want to load or store from that label’s address
  - When used with a `beq` instruction, a label indicates you want to branch to that label’s address.
- When a label is used inplace of a number in `field0` for a `.fill` assembler directive, you are to resolve the label’s value, and use that value for the fill.

Take a look at the spec example for project 1A:

spec.as - lines with label definitions are highlighted

```
1           lw      0       1       five    load reg1 with 5 (symbolic address)
2           lw      1       2       3       load reg2 with -1 (numeric address)
3   start   add     1       2       1       decrement reg1
4           beq     0       1       2       goto end of program when reg1==0
5           beq     0       0       start   go back to the beginning of the loop
6           noop
7   done    halt                            end of program
8   five    .fill   5
9   neg1    .fill   -1
10  stAddr  .fill   start                   will contain the address of start (2)
11
```

Notice how we define the labels `start`, `done`, `five`, `neg1`, and `stAddr`. Remember from section 2.2 that each line of assembly represents an address. Thus we say the following:

- The label `start` resolves to a value of 2, since it is defined on the 3rd line, which relates to address 2 (We count starting by 0 for addresses, but by 1 for line numbers).
- The label `done` resolves to a value of 6
- The label `five` resolves to a value of 7
- The label `neg1` resolves to a value of 8
- The label `stAddr` resolves to a value of 9

Furthermore, in the spec example for project 1A, there are a few usages of labels as arguments:

spec.as - lines that define labels are highlighted

```
1           lw      0       1       five    load reg1 with 5 (symbolic address)
2           lw      1       2       3       load reg2 with -1 (numeric address)
3   start   add     1       2       1       decrement reg1
4           beq     0       1       2       goto end of program when reg1==0
5           beq     0       0       start   go back to the beginning of the loop
6           noop
7   done    halt                            end of program
8   five    .fill   5
9   neg1    .fill   -1
10  stAddr  .fill   start                   will contain the address of start (2)
11
```

See the handling labels section to see how your assembler should handle assembling lines of assembly that use symbolic labels into machine code.

spec.as - lines that use labels are highlighted

```
1           lw      0       1       five    load reg1 with 5 (symbolic address)
2           lw      1       2       3       load reg2 with -1 (numeric address)
3   start   add     1       2       1       decrement reg1
4           beq     0       1       2       goto end of program when reg1==0
5           beq     0       0       start   go back to the beginning of the loop
6           noop
7   done    halt                            end of program
8   five    .fill   5
9   neg1    .fill   -1
10  stAddr  .fill   start                   will contain the address of start (2)
11
```

2.3. LC-2K Machine Code Instruction Formats

An LC-2K machine code file (`*.mc`) is made up of multiple lines of hexadecimal numbers. Each line of the machine code file represents the number stored at that address in the memory. For example, the first line of the machine code file represents the value of address 0 when the program begins.

Bits 31-25 are unused for all instructions, and should always be 0. Bit 0 is the least-significant bit.

R-type instructions (`add` , `nor`)	bits 24-22: `opcode` bits 21-19: `reg A` bits 18-16: `reg B` bits 15-3: `unused` (should all be 0) bits 2-0: `destReg`
I-type instructions (`lw` , `sw` , `beq`)	bits 24-22: `opcode` bits 21-19: `reg A` bits 18-16: `reg B` bits 15-0: `offsetField` (a 16-bit, 2’s complement number with a range of -32768 to 32767)
J-type instructions (`jalr`)	bits 24-22: `opcode` bits 21-19: `reg A` bits 18-16: `reg B` bits 15-0: `unused` (should all be 0)
O-type instructions (`halt` , `noop`)	bits 24-22: `opcode` bits 21-0: `unused` (should all be 0)

3. LC-2K Assembly Language and Assembler (40%)

The first part of this project is to write a program to take an assembly-language program and translate it into machine language. You will translate assembly-language names for instructions, such as beq, into their numeric equivalent (e.g. 100), and you will translate symbolic names for addresses into numeric values. The final output will be a series of 32-bit instructions (instruction bits 31-25 are always 0).

The assembler should make **two passes** over the assembly-language program. In the first pass, it will calculate the address for every symbolic label. Assume that the first instruction is at address 0. In the second pass, it will generate a machine-language instruction (in hexadecimal) for each line of assembly language. For example, here is an assembly-language program (that counts down from 5, stopping when it hits 0).

spec.as

```
        lw      0       1       five    load reg1 with 5 (symbolic address)
        lw      1       2       3       load reg2 with -1 (numeric address)
start   add     1       2       1       decrement reg1
        beq     0       1       2       goto end of program when reg1==0
        beq     0       0       start   go back to the beginning of the loop
        noop
done    halt                            end of program
five    .fill   5
neg1    .fill   -1
stAddr  .fill   start                   will contain the address of start (2)

```

And here is the corresponding machine language:

spec.as's corresponding machine language, with addresses and hex | Your output should NOT look like this

```
!! Your output should only include the machine code in hexadecimal !!
!! The addresses and notes are just for your understanding         !!
!! See spec.mc.correct for what your output should look like       !!
(address 0): 0x00810007
(address 1): 0x008A0003
(address 2): 0x000A0001
(address 3): 0x01010002
(address 4): 0x0100FFFD
(address 5): 0x01C00000
(address 6): 0x01800000
(address 7): 0x00000005
(address 8): 0xFFFFFFFF (note: 2's complement representation of -1)
(address 9): 0x00000002
```

Be sure you understand how the above assembly-language program got translated to machine language.

Since your programs will always start at address 0, your program should only output the memory contents in hexadecimal and not output the addresses.

spec.mc.correct for P1A / spec.mc for P1S

```
0x00810007
0x008A0003
0x000A0001
0x01010002
0x0100FFFD
```

```
0x01C00000
0x01800000
0x00000005
0xFFFFFFFF
0x00000002
```

3.1. Handling labels

For `lw` or `sw` instructions, the assembler should compute `offsetField` to be equal to the address of the label. This could be used with a zero base register to refer to the label, or could be used with a non-zero base register to index into an array starting at the label. For `beq` instructions, the assembler should translate the label into the numeric `offsetField` needed to branch to that label.

3.2. Your Assembler's Input and Outputs

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

```
./assembler program.as program.mc
```

Note that the format for running the command must use command-line arguments for the file names (rather than standard input and standard output). Your program should store only the list of hexadecimal numbers in the machine-code file, one instruction per line. Any deviation from this format (e.g. extra spaces or empty lines) will render your machine-code file ungradeable. Any other output that you want the program to generate (e.g. debugging output) can be printed to standard output.

Note to compile your assembler, see Appendix B Makefile tips

3.3. Error Checking

Your assembler should catch the following errors in the assembly-language program:

- Use of undefined labels
- Duplicate definition of labels
- `offsetFields` that don't fit in 16 bits

- Unrecognized opcodes
- Non-integer register arguments
- Registers outside the range [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 errors, i.e. errors that would occur at the time the assembly-language program executes (e.g. branching to address -1, infinite loops, etc.). You are not required to output any specific output when an error is encountered.

3.4. Test Cases

An integral (and graded) part of writing your assembler will be to write a suite of test cases to validate any LC-2K assembler. Writing thorough and robust test suites is a common practice in in the real–world software companies, Writing a comprehensive suite of test cases will deepen your understanding of the project specification and your program, and it will help you a lot as you debug your program. Moreover, staff will have a much easier time helping you identify issues in your project if you have test cases that are producing incorrect output on your implementation.

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 needed for full credit (the solution test suite is composed of 5 test cases, each < 10 lines long). See Section 6 for how your test suite will be graded.

> **Hints:** The spec assembly-language program is a good case to include in your test suite, though you'll need to write more test cases to get full credit. Remember to create some test cases that test the ability of an assembler to check for the errors in Section 3.3.

Note: All instructions should appear before any `.fill`'s. Instructions and `.fill`'s should not be interleaved. e.g. Your assembly programs should look like this:

Correct usage of .fill

```
1    noop
2    noop
3    .fill 0
4    .fill 1
```

They should **NOT** look like this:

Incorrect usage of .fill

```
    noop
    .fill 0
    noop
    .fill 1
```

This won't be enforced for project 1, but assembly programs with interleaved instructions and `.fill`'s will not work properly in project 2. Note that many students like to reuse their project 1 assembly tests for project 2.

> ⚠ ***IMPORTANT:*** *Test case names must NOT have empty spaces in them. Any test cases with spaces in it will not be graded. For example, "tes t.as" is incorrectly formatted.*

3.5. Assembler Hints

Since `offsetField` is a 2's complement number, it can only store numbers ranging from -32768 to 32767. For symbolic addresses, your assembler will compute `offsetField` so that the instruction refers to the correct label.

Remember that `offsetField` is only a 16-bit 2's complement number. Since Linux integers are 32 bits, you'll have to chop off all but the lowest 16 bits for negative values of `offsetField`. Consider where a value being negative is significant. See the providied `static inline int isNumber(char *string)` method.

To print integers in their hexadecimal represenation, you can use the `printf()` function with the `%x` format specifier. For example, the following code snippet will output `ff`, which is the hexadecimal representation of 255.

```c
int num = 255;
printf("%x", num);
```

See Appendix A for more information on `printf()`.

4. Behavioral Simulator (40%)

The second part of this assignment is to write a program that can simulate any legal LC-2K machine-code program. The input for this part will be the machine-code file that you created with your assembler. With a program name of `simulator` and a machine-code file of `program.mc`, your program should be run as follows:

```
./simulator program.mc > output
```

This directs all printfs to the file `output`.

The simulator should begin by initializing all registers and the program counter to 0. The simulator will then simulate the program until the program executes a halt.

The simulator should call the `printState` function before executing each instruction and once just before exiting the program. This function prints the current state of the machine (program counter, registers, memory). `printState` will print the memory contents for memory locations defined in the machine-code file (addresses 0-9 in the spec example).

4.1 Simulator Behavior

The purpose of the simulator is to keep a record of the current state of our registers and memory. Before each instruction is executed, a call to `printState` will be made, showing the values of your program's memory and registers. The input for the simulator will be a machine code file, meaning you will need to parse the input and determine what actions to take.

Consider the following machine code: `0x000A0003`

The same number, but in binary: `0b 0000 0000 0000 1010 0000 0000 0000 0011`

From here, we can determine the opcode and all other arguments. Recall that all numbers are binary under the hood, so we can implicitly think about the machine code in binary (even though it is given to us in hexadecimal notation)

Looking at positions 24-22, the opcode bits are 000, implying it is an ADD instruction per Section 2, ADD has 3 arguments:

```
RegA: which is bits 21-19,  is 0b001, or 1
RegB: which is bits 18-16,  is 0b010, or 2
DestReg: which is bits 2-0, is 0b011, or 3
```

Therefore, we know this line of machine code is trying to do:

```
Register 3 = Register 1 + Register 2
```

Note: we are adding the register's values, not their names.

4.2 Test Cases

As with the assembler, you will write a suite of test cases to validate any LC-2K simulator.

The test cases for the simulator part of this project will be short, valid assembly-language programs that, after being assembled into machine code, serve as input to a simulator. You will submit your suite of test cases together with your simulator, and we will grade your test suite according to how thoroughly it exercises an LC-2K simulator. Each test case may be at most 50 lines and may execute at most 200 cycles on a correct simulator, and your test suite may contain up to 20 test cases. These limits are much larger than needed for full credit (the solution test suite is composed of a couple test cases, each executing less than 40 instructions). See Section 6 for how your test suite will be graded.

> ⚠ **Warning**: Behavior is defined only for accesses to memory addresses in the range [0, 65535]. In your test cases, do not access memory addresses outside of this range with LW or SW instructions. This is **NOT** one of the errors you are required to check for, but assembly programs with undefined behavior may execute differently on your simulator than on our reference simulator.

4.3 Simulator Hints

Be careful how you handle `offsetField` for `lw`, `sw`, and `beq`. Remember that it's a 2's complement 16-bit number, so you need to convert a negative `offsetField` to a negative 32-bit integer on the Linux workstations (by sign extending it). One way to do this is to use the following function, also given in the starter code:

convertNum function from starter code

```
1  static inline int convertNum(int32_t);
2  // convert a 16-bit number into a 32-bit Linux integer
3  static inline int convertNum(int32_t num) {
4      return num - ( (num & (1<<15)) ? 1<<16 : 0 );
5  }
```

An example run of the simulator (not for the specified task of multiplication) is included in the starter code for Project 1 S in the file `spec.out.correct`

5. Assembly-Language Multiplication (20%)

The third part of this assignment is to write an assembly-language program to multiply two numbers. Input the numbers by reading memory locations called `mcand` and `mplier`. The result should be stored in register 3 when the program halts. You may assume that the two input numbers are at

most 15 bits and are positive; this ensures that the (positive) result fits in an LC-2K word. Remember that shifting left by one bit is the same as adding the number to itself. Given the LC-2K instruction set, it's easiest to modify the algorithm so that you avoid the right shift. Submit a version of the program that computes ( ).

Your multiplication program must be reasonably efficient — it must be at most 50 lines long and execute at most 1000 cycles for any valid input (this is several times longer and slower than the solution). To achieve this, you are strongly encouraged to consider using a loop and shift algorithm to perform the multiplication; algorithms such as successive addition (e.g. multiplying by adding 5 six times) will take too long.

6. Grading, Auto-Grading, and Formatting

We will grade primarily on functionality, including error handling, correctly assembling and simulating all instructions, input and output format, method of executing your program, correctly multiplying, and comprehensiveness of the test suites.

To help you validate your project, your submission will be graded automatically after submission. 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** on any given day. That is, you will receive feedback with your score only three times on any given day. All subsequent submissions will be silently graded. Your final score will be derived from your overall best submission to the autograder.

⚠ Submissions will only be accepted if the submitted student test cases expose a minimum number of buggy instructor solutions. If not enough buggy instructor solutions are exposed, the submission will not be graded and will not count towards your daily submission limit. In order for the submission to be accepted, it must expose:

- 1a: 6 buggy instructor solutions
- 1s: 3 buggy instructor solutions

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 suite of test cases for the assembler and simulator parts of this project will be graded according to how thoroughly they test an LC-2K assembler or simulator. We will judge thoroughness of the test suite by how well it exposes potential bugs in an assembler or simulator.

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. The test suite is graded based on how many of the buggy assemblers were exposed by at least one test case. This is known as *mutation testing* in the research literature on automated testing. Your test suite is run on **19** buggy assemblers. To receive all Mutation Testing points, your test suite must expose at least **15** of the **19** buggy assemblers.

For the simulator test suite, the auto-grader will correctly assemble each test case, then use it as input to a set of buggy simulators. A test case exposes a buggy simulator by causing it to generate a different answer from a correct simulator. The test suite is graded based on how many of the buggy simulators were exposed by at least one test case. Your test suite is run on **10** buggy assemblers. To receive all Mutation Testing points, your test suite must expose at least **7** of the **10** buggy assemblers. Note that the test cases for the simulator should all be valid, correct assembly language programs.

Because all programs will be auto-graded, you must be careful to follow the exact formatting rules in the project description:

1. **(assembler)** Follow exactly the format for inputting the assembly-language program and outputting the machine-code file.
2. **(assembler)** Call `exit(1)` if you detect errors in the assembly-language program. Call `exit(0)` if you finish without detecting errors.
3. **(assembler)** Do not modify `readAndParse`, `isNumber`, or `printHexToFile` at all. Download this code into your program electronically (don't re-type it) to avoid typos.
4. **(simulator)** Don't modify `printState` or `stateStruct` at all. Download this code into your program electronically (don't re-type it) to avoid typos.
5. **(simulator)** Call `printState` exactly once before each instruction executes and once just before the simulator exits. Do not call printState at any other time.
6. **(simulator)** Don't print the sequence "@@@" anywhere (except where the provided `printState` function prints it).
7. **(simulator)** `state.numMemory` must be equal to the number of lines in the machine-code file.
8. **(simulator)** Initialize all registers to 0.

9. **(multiplication)** Store the result in register 3.

10. **(multiplication)** The two input numbers must be in locations labeled `mcand` and `mplier` (lower-case).

7. Turning in the Project

Use autograder.io to submit your files.

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

```
1) assembler (part 1a)
    a. Your assembler, a C program named "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")

2) simulator (part 1s)
    a. Your simulator, a C program named "simulator.c"
    b. suite of test cases (each test case is an assembly-language program
        in a separate file, ending in: ".as", ".s", or ".lc2k")

3) multiplication (part 1m)
    a. Your assembly program for multiplication, named "mult.as", "mult.s", or
"mult.lc2k"
```

Your code will be compiled with the GCC compiler using the C99 standard. Use the provided makefile to compile your programs. See Appendix B Makefile tips

The official time of submission for your project will be the time the last file is sent. If you send in anything after the due date, your project will be considered late (and will use up your late days). If you have already used up all of your late days, additional late submissions will not be scored for your project grade.

Appendix A: C Programming Tips

Here are a few programming tips for writing C programs to manipulate bits:

1) To indicate a hexadecimal constant in C, precede the number by `0x`. For example, 27 decimal is `0x1B` in hexadecimal.

2) The value of the expression `(a >> b)` is the number “a” shifted right by “b” bits. Neither a nor b are changed. E.g. `(25 >> 2)` is 6. Note that 25 is `11001` in binary, and 6 is `110` in binary.

3) The value of the expression `(a << b)` is the number “a” shifted left by “b” bits. Neither a nor b are changed. E.g. `(25 << 2)` is 100. Note that 25 is `11001` in binary, and 100 is `1100100` in binary.

4) To find the value of the expression `(a & b)`, perform a logical AND on each bit of a and b (i.e. bit 31 of a ANDed with bit 31 of b, bit 30 of a ANDed with bit 30 of b, etc.). E.g. `(25 & 11)` is 9, since:

```
    11001 (binary)
  & 01011 (binary)
--------------------
 =  01001 (binary), which is 9 decimal.
```

5) To find the value of the expression `(a | b)`, perform a logical OR on each bit of a and b (i.e. bit 31 of a ORed with bit 31 of b, bit 30 of a ORed with bit 30 of b, etc.). E.g. `(25 | 11)` is 27, since:

```
    11001 (binary)
  | 01011 (binary)
--------------------
 =  11011 (binary), which is 27 decimal.
```

6) `~a` is the bit-wise complement of a (a is not changed).

Use these operations to create and manipulate machine-code. For example:

- To look at bit 3 of the variable a, you might do: `(a>>3) & 0x1`
- To look at bits (bits 15-12) of a 16-bit word, you could do: `(a>>12) & 0xF`
- To put a 6 into bits 5-3 and a 3 into bits 2-1, you could do: `(6<<3) | (3<<1)`

If you’re not sure what an operation is doing, print some intermediate results to help you debug.

7) To print in C, use the printf() function (this is included in stdio.h). We can print a message like such: printf(“Hello world\n”); If we want to print values, we need to include their types in the output string:

```
    use "%d" for an int value
    use "%c" for a char value
    use "%s" for a string value
    use "%f" for a non int number
        use "%x" for an int value in hexadecimal format
```

When we want to print a value, we need to include these identifiers, alongside the value of the variable as an argument of printf().

Example:

```
int num = 370;
char name[5] = "EECS";
printf("Welcome to %s %d\n, name, num); //This prints "Welcome to EECS 370".
```

8) Please review discussion 1 for more C information

Appendix C: Makefile Tips

You can use the provided `Makefile` in the starter code by doing

```
$ make <rule>
```

Where `<rule>` is a rule that is defined in the makefile.

Open up the `Makefile` and see what `<rule>`'s we have already written for you, they will have the following format:

```
targets : prerequisites
        recipe
        …
```

(From https://www.gnu.org/software/make/manual/html_node/Rule-Syntax.html#Rule-Syntax)

Example 1 : Compiling the assembler executable

In our provided makefile we see that we define the rule `assembler` below. This rule relies on assembler.c. The recipe compiles the `assembler.c` file and creates the `assembler` executable. Feel free to explore the `Makefile` and see what other rules we provide!

Makefile

```
# Compile Assembler
assembler: assembler.c
    $(CXX) $(CXXFLAGS) $< -o assembler
```

Calling `make assembler` in your terminal to compile the assembler executable

```
$ make assembler
gcc -std=c99 -Wall -Werror -g3 assembler.c -o assembler -lm
```

Example 2 : Compiling the assembler executable

We also define some rules that use *pattern rules*, an example is the rule `%.mc` which we can use to assemble an LC2K assembly file into it's machine code representation.

Makefile

```
# Assemble an LC2K file into Machine Code
%.mc: %.as assembler
	./assembler $< $@
```

Example, calling `make spec.mc` in your terminal to assemble `spec.as` to create `spec.mc`

```
$ make spec.mc
gcc -std=c99 -lm -Wall -Werror -g3 assembler.c -o assembler
./assembler spec.as spec.mc
```