CMSC 216 Project #4 Spring 2022

Date due: Thursday, March 10, 11:59:59 p.m.

1 Introduction and purpose

In this project you will write a simulation of the Nelovo CPU, which will be able to execute Nelovo machine–language programs. Although that is the major part of the project, due to the new features introduced in this project, and now that pointers have been covered in full, you will first modify two of your functions from Project #3. The purpose of the project is to use some concepts covered recently, and also to get a better understanding of low–level hardware and machine–language concepts.

Since this project extends upon Project #3, you may not discuss your implementation of that project with anyone else or look at anyone else's Project #3 code, or let anyone else look at yours, even after the Project #3 graded results are made available, until after the late submission period for this project has passed. (And even then, as explained in class, no one could look at code, let anyone look at their code, or discuss how to write a project if they still haven't passed half of its public tests.)

Due to the size of the course it is not feasible for us to be able to provide project information or help via email/ELMS messages, so we will be unable to answer such questions. However you are welcome to ask any questions verbally during the TAs' office hours (or during, before, or after discussion section or lecture if time permits). However, you **cannot wait to write projects in a course with more than 475 students**.

1.1 Extra credit and number of submissions

You can again get extra credit for this project. If you make **only one submission** that passes **all the public tests** you will get **3 extra–credit bonus points**. And if your single submission is made **at least 48 hours before the on–time deadline** you will get **3 additional extra credit bonus points**, for 6 extra credit points total. (Obviously you can't get the second 3 extra credit points if you don't get the first 3, but you can get the first 3 without getting these second 3.)

However, as before, if you make too many submissions you may again lose credit. To avoid making submissions that don't compile or don't pass all of the public tests you should compile the tests, run them, check your output, and fix any problems, all **before** submitting.

(If for some reason your code passes all the public tests on Grace, but doesn't work when you submit it, so you have to submit more than once– **and this is not due to an error on your part or a bug in your code**– you can talk with me verbally in office hours about receiving the extra credit despite having more than one submission.)

2 Modifications to functions from Project #3

In order to write the `run_program()` function described in Section 3 below that simulates the execution of Nelovo programs, you will have to define a type named `Nelovo` that is used by most of the functions, but which is not already defined. A variable of type `Nelovo` has to be able to store 14 registers, each of which is a `Mach_word`, and also has to store the contents of 4096 memory locations, which are also `Mach_words`. The registers should be accessible by register numbers, and the memory locations should be accessible by their Nelovo memory addresses. Instead of us providing the type `Nelovo`, we want you to come up with your own definition for it.

The `project04.tgz` tarfile has a new version of `machine.h` for this project. Besides having the prototypes for the new functions to be written in this project, it also has modified prototypes for two of the Project #3 functions, which now have parameters of type `Nelovo` instead of being arrays of `Mach_words`. As mentioned, the type `Nelovo` is not defined in the new `machine.h`. Instead, the new `machine.h` includes a header file with name `nelovo-datastructure.h` that is not provided. You must create `nelovo-datastructure.h` (spelling its name **exactly** as shown) and in it write a definition of the type `Nelovo`, along with **any other definitions that it requires**.

The changes to the two functions from Project #3 are explained in the following two subsections, but you will not be able to test these modified functions until you write the `load_program()` function described in Section 3 below. So you may want to first write `nelovo-datastructure.h` with your definition of `Nelovo`, then write `load_program()`, then modify these two functions from Project #3 to use the new type `Nelovo`, and lastly write the remaining new functions in Section 3.2, Section 3.3, and Section 3.4. Note that the changes to the two functions from Project #3 described in this section take half a page to explain but they are straightforward to implement and not difficult.

2.1 unsigned short disassemble(Nelovo machine, unsigned short max_addr)

This function should produce the same output as in Project #3, but its parameters now have different types. Instead of being passed an array of Mach_words, it will be passed something of your type Nelovo, which must contain 4096 Mach_words representing the Nelovo's memory (as well as 14 Mach_words representing its registers), and it should print instructions in the memory, from the one with address 0 up through the one with address max_addr. The only other change compared to Project #3 is that disassemble() should now return 1 if max_addr is valid and all the instructions in the Nelovo parameter up through max_addr are valid (the return value in this case was unspecified in Project #3). It should return 0 without producing any output if max_addr is larger than the address of the last word in the Nelovo.

This function, and two others below (get_register() and get_memory()) that have parameters of type Nelovo, must assume they are being passed a valid variable of that type with definite (non–garbage) values. This is something that in C is the responsibility of the caller of a function to ensure, but not something that the function itself can check.

2.2 short check_branches(Nelovo *const machine, unsigned short max_addr, unsigned short *const invalid_instr)

Besides changing check_branches() to use your new type Nelovo, the way that it returned different values in Project #3 was somewhat awkward and contrived, because pointers and pointer parameters had not yet been covered. Now that they have been, it can have parameters and a return value that are more natural according to C conventions.

As in the changes to disassemble() described above, instead of being passed an array of Mach_words, this function will now be passed something of your type Nelovo, which will contain Mach_words representing memory, and it should check whether any test instructions in the Nelovo variable's memory from the one with address 0 up through the one with address max_addr have invalid addresses (as defined in Project #3). However, the function should now just return either -1, 0 or 1. It should return -1 if its parameters are invalid, meaning if either pointer parameter is NULL or if max_addr is larger than the address of the last word in the Nelovo. It should return 1 if the parameters are valid and none of the test instructions being examined contain invalid memory addresses. It should return 0 if the parameters are valid and one or more of the test instructions of the memory locations being examined have invalid addresses, **and** in this case it should store the Nelovo memory address of the **first** instruction with an invalid address into the variable that invalid_instr points to. **Only** if there are any test instructions with invalid memory addresses should the variable that invalid_instr points to be modified– it should not be changed if the function returns -1 or 1.

This function, and the four new ones below that have one or more pointer or array parameters, may assume that the values passed into their pointer parameters are valid pointers that point to variables of the correct type, in other words, they may assume that pointers passed in as arguments are not uninitialized or invalid pointers. This is something that functions in C cannot check, and is the caller's responsibility to ensure instead.

3 New functions to be written

The first three of these four functions are short and relatively simple to write.

3.1 unsigned short load_program(Nelovo *const machine, const Mach_word program[], unsigned short num_instrs)

When a program in a real computer system is run, the first thing that must occur is that the operating system must load the program into the computer's memory. This function simulates that for the Nelovo by copying the elements of the array program, which represent a Nelovo program, into the memory of the Nelovo parameter machine. If either of its first two parameters are NULL or if num_instrs is more than the maximum **number** of instructions that could fit into the Nelovo's memory the function should just return 0 without changing anything. Otherwise it should copy the first num_instrs elements of the array program into the memory of the Nelovo variable that the first parameter points to, starting at the first memory location (the one with address 0), and return 1.

3.2 unsigned short get_register(Nelovo machine, unsigned short which_register, Mach_word *const result)

This function is somewhat like a get method in Java– it simply gets the value of one component of the Nelovo variable that its first parameter points to (in particular it gets one of its register values). Since you are creating the definition of

Nelovo we don't know how it is written or how to refer to its components, so in order for our tests to check that the components of a Nelovo are valid after calling other functions, you have to write this function (and the next one as well) that just get the values of components of your Nelovo type.

This function should return 0 without changing anything if its parameters are invalid, meaning if its pointer parameter result is NULL or if which_register is not a valid register number for the Nelovo. If its parameters are valid it should return 1, and store the value of the Nelovo parameter's register with number which_register into the variable that result points to. **Only** if the parameters are valid should the variable that result points to be modified; it should not be changed if the function returns 0. (Of course if the function is returning 0 because result itself is NULL, it would be impossible to store anything in the variable that it points to anyway.)

3.3 unsigned short get_memory(Nelovo machine, unsigned short which_memory_addr, Mach_word *const result)

This function is very similar to get_register() except it gets the value of a machine word in the memory of the Nelovo variable that its first parameter points to, instead of the value of a register.

It should return 0 without changing anything if its pointer parameter result is NULL or if which_memory_addr is not a valid memory address for the Nelovo, meaning if it is not divisible by 4 or if it is larger than the address of the last word in the Nelovo. If its parameters are valid it should return 1, and store the value of the Nelovo parameter's memory location with address which_memory_addr into the variable that result points to. **Only** if the parameters are valid should the variable that result points to be modified.

3.4 Status run_program(Nelovo *const machine, unsigned int max_instrs, unsigned short *const num_instrs_executed, unsigned short *const invalid_instr, unsigned short trace_flag)

This function will act as a simple interpreter for Nelovo machine language programs. It is simple because the language that it is interpreting– Nelovo machine code– is very simple. However, your function could still allow Nelovo programs to compute anything that can be computed with integers, given the limited system resources of the simulated Nelovo architecture (although it would be very awkward and clunky to write in Nelovo machine code, because it simulates a very rudimentary instruction set). Your function must model changes to both components of the Nelovo architecture, meaning memory and registers.

3.4.1 Parameters to run_program()

The function's parameters are used for the following purposes:

- The first parameter machine is a pointer to a Nelovo variable that as described above contains simulated copies of the Nelovo's complete memory and registers.

  Where we use wording below like "the function's Nelovo parameter", it means the Nelovo variable that the function's pointer parameter machine is pointing to.

- The second parameter max_instrs gives the maximum number of Nelovo instructions that the function should simulate the execution of before returning.

- The third parameter num_instrs_executed is a pointer to a variable that the function will fill in with the number of instruction that it simulates executing while running the program in the first parameter.

- The fourth parameter invalid_instr is a pointer to a variable into which the function will store the address of the first invalid instruction seen (if any instructions are invalid) while it is simulating the execution of instructions.

- The fifth parameter trace_flag indicates whether or not the function should produce additional debugging output while simulating execution of Nelovo instructions. If 0 is passed into it then no debugging output should be produced. If a nonzero value is passed into it the function can produce any debugging output that you want for your own use in developing and testing your function.

  Our tests will always pass 0 into this parameter, so if it is nonzero you can produce whatever output is useful for you, but we won't test it. **However**, if you have to come to office hours for any help with this function you **must**

have written it so if `trace_flag` is nonzero the address of each instruction, and the instruction itself, are printed prior to the execution of every instruction, and the contents of all of the Nelovo's registers are printed after the effect of each instruction is simulated. If your function has any bugs or produces the wrong results for any tests, run it passing a nonzero value into this parameter, so you can use this output to trace what it's doing.

Your function can produce additional debugging information beyond that described above if you like. You may even want to produce different debugging output for different nonzero values of this parameter, to easily enable or disable different sections or amounts of debugging information.

3.4.2 Operation of `run_program()`

The effects of the Nelovo instructions, which is what `run_program()` is to simulate, are described in Section 2.4 of Project #3. As an example of what your function has to do, if an element of the memory of the `Nelovo` variable that the first parameter points to contains the value `0x0c490000`, it represents the instruction `mul R1 R2 R4`, and if the function simulates executing this instruction it should compute the product of the values in the registers R2 and R4 (the third and fifth registers) in the `Nelovo` parameter, and put the result into register R1, the second register in the Nelovo parameter. And if the program in the Nelovo parameter has an instruction `0x52c00064` that is executed, it represents `sw R11 100`, so your function should copy or store whatever value is in register R11 (the twelfth register in the Nelovo parameter) into the memory element in the Nelovo parameter that has address 100 (meaning the 26th memory word).

Before starting to simulate the execution of the Nelovo program, the function will have to initialize all 14 registers in its Nelovo parameter, including the program counter register, to have the value zero. Initializing the program counter register to 0 will cause the first instruction in the program to be the first one that is executed. The memory locations in the Nelovo parameter do not have to be initialized to any specific value before the function starts simulating execution of the program; they will just have garbage values. (The effect is undefined if a Nelovo program uses the contents of a memory location that hasn't first been given a specific value.)

The function will have to simulate the fetch/decode/execute cycle of the Nelovo's CPU by repeatedly:

1. Determining what the address of the next instruction to be executed is, based on the value of the program counter register (the contents of register R13, the fourteenth register in the Nelovo parameter).
2. Accessing the element of the Nelovo's memory that corresponds to the address in the program counter, to fetch the instruction to be executed.
3. Performing that instruction, modifying the contents of its Nelovo parameter's memory or registers as appropriate (or performing I/O) depending on what instruction it is and the values of its operands.
4. Advancing the program counter as appropriate and repeating the above steps. Every instruction causes the program counter to be incremented by 4 to refer to the next four–byte instruction, except:
    - If a `halt` instruction is executed the program counter should not be incremented at all, and
    - If a `test` instruction is executed when its first register operand contains a nonzero value, in which case instead of being incremented by 4, the program counter should be changed to whatever value is in the memory address field of the instruction.

3.4.3 `run_program()`'s return value

The function should continue simulating the Nelovo CPU performing the fetch–decode–execute cycle described above until one of the following occurs:

- If in the process of simulating the execution of the Nelovo program a `halt` instruction is performed. In this case the function should return the value `HALTED`, which is a constant in an enum `Status` defined in `machine.h` (it should **not** increment the program counter register before returning).
- If an instruction of the program is invalid (described below) the function should return the value `ERROR` (another `Status` enum value) **without** incrementing the program counter register. In this case it should store the Nelovo memory address of the invalid instruction into the variable that its fourth parameter `invalid_instr` points to. Unless the function returns `ERROR`, the value of the variable that `invalid_instr` points to should not be modified.

An instruction that the program attempts to execute is invalid in these cases:

  - In the same conditions that `is_valid()` from Project #3 would call it invalid, or
  - If it is a `div` or `rem` instruction and the value in the register that its third register operand refers to is zero when the instruction is reached.

  If `ERROR` is returned the incorrect instruction should not have an effect on memory or registers (including the program counter) before the function leaves.

- If after simulating the execution of `max_instrs` a `halt` instruction has not been performed and `ERROR` has not been returned, the function should return the value `RUNNING` (another *Status* value) without simulating the execution or more instructions. This simulates the Nelovo CPU and operating system stopping the execution of one program, which has not completed, to be able to switch to and execute another program, as real systems do.

Regardless of the reason that the function returns, or what value it returns, the variable that `num_instrs_executed` points to should always have the number of valid instructions that the Nelovo program executed before the function returned; this number will be zero or more.

Note that the function must check whether `max_instrs` instructions have been executed **before** executing each instruction, so it's not possible for both of the non–error conditions or return values to simultaneously apply– if the function has executed `max_instrs` − 1 instructions and the next one is `halt`, the `halt` will be executed and `HALTED` will be returned. But if the function has executed `max_instrs` instructions and the next one is `halt`, the function will return before the `halt` is even seen.

The arithmetic (`add`, `sub`, etc.) and logical instructions should produce the same results as C's operators would when applied to the values of their operands. The situations described above are the only conditions that would make an instruction invalid. If any instruction performing an arithmetic operation would result in a value outside the range that can be stored in an `unsigned int` (`Mach_word`) variable on the Grace machines, which are used to represent registers and memory, the results of the function are undefined, so it doesn't matter what the function does or returns.

Note that it doesn't make complete logical sense to have a `neg` instruction that performs arithmetic negation, when all Nelovo memory and registers just store unsigned values. We are using unsigned values for Nelovo memory and registers because it's much easier to perform bitwise operations and do number conversions when only nonnegative values need to be considered. But you can actually apply the C unary negation operator to an unsigned value, and as seen in lecture recently, you will just get a result that wraps around and becomes a (likely large) positive unsigned value instead of a negative number. This is just what your function should produce when simulating the `neg` instruction.

Also note that the fact that a program contains an invalid instruction does **not** necessarily mean that `run_program()` will return `ERROR`; this is because the instruction may never be executed (similar to how a C program could have an infinite loop but if that part of the code is never executed, the infinite loop will not occur). Similarly, `ERROR` would not be returned even if an instruction contains an invalid memory address if the program never tries to access that address; for example, a `test` instruction may contain an invalid address but when that instruction is executed its register operand's value is 0, so the address is not jumped to.

You may wonder why incorrect instructions and memory addresses need to be detected while a Nelovo program is executing: can't they checked for earlier, before even starting to run the program? Actually not, because it's possible for an assembly program to try to execute the contents of a memory location into which data was stored by the program itself, and that data may not represent a valid instruction. It is even possible for a running program to modify its own instructions, or create new instructions and then execute them, so validity of instructions can only be checked as each one is executed. Also, the second operand of a division or modulus operation may not be known when a program is first loaded into memory; it could be performing a division or modulus with a value that was read from the input.

3.4.4 `run_program()`'s input and output

The function should produce no output except for the results of any `prt` instructions executed in the program being interpreted. The values printed by `prt` instructions should be printed in **decimal**, without any extra leading zeros (the only number printed that should begin with 0 is the number 0 itself). The simple Nelovo does not have facilities to manipulate character data, so if multiple numbers were printed they would appear without anything separating them. A program that printed 1 and then 2 could not be differentiated from one that printed 12. To avoid ambiguity, the values printed by any `prt` instructions should be printed **on a line by themselves, immediately preceded by a newline**.

If a `read` instruction is executed the function should read a single **unsigned** integer from the input and store it into the register that the value of the instruction's first register operand refers to.

A Development procedure review

A.1 Obtaining the project files, compiling, checking your results, and submitting

Log into the Grace machines and use commands similar to those from before:

```
cd ~/216
tar -zxvf ~/216public/project04/project04.tgz
```

This will create a directory `project04` that contains the files for the project, including the header file `machine.h` and the public tests. You **must** have your coursework in your course disk space for this class. Copy your `machine.c` file from your `project03` directory to this new `project04` directory and in this new directory create a file named `nelovo-datastructure.h` (spelled **exactly** that way) that contains a definition of a type `Nelovo`, and any types that your definition needs. Then edit the copy of `machine.c` in the new directory to modify the functions from Project #3 described in Section 2 and write the new functions described in Section 3. The new and modified prototypes are in the new `machine.h` for this project.

A command like `gcc public01.c machine.c -o public01.x` will compile your code with the first public test (or you can use Emacs to compile if desired); replace the 1s in the command with 2s for the second public test, etc. You can also use separate compilation, compiling each source file to form object files, which are then linked together. As before, use `diff` to compare your code's output to the public test outputs in the project tarfile, for example `public01.x | diff - public01.output` to test your results on the first public test.

Any tests that call `run_program()` to simulate execution of a Nelovo program containing `read` instructions will have an associated input data file, and those tests should be run with input redirected from that file, for example `public11.x < public11.input | diff -b - public11.output` (the comment at the top of such tests will specify this).

Running `submit` from the project directory will submit your project, but as mentioned in the introduction above, **before** you submit you **must** make sure you have passed all the public tests, by compiling and running them yourself.

A.2 Grading criteria

Your grade for this project will be based on:

public tests	40 points
secret tests	45 points
programming style	15 points

B Project–specific requirements, suggestions, and other notes

- When you are declaring local variables (or any identifiers, such as helper functions or their parameters), note that `register` is actually a keyword in C, although it was not explained in class (it is in the Reek text though). So it will be a syntax error if you try to give anything the name `register`.
- You **cannot** modify anything in the header file `machine.h` or add anything to `machine.h`, because your submission will be compiled on the submit server using our version of this file.

  Your code **may not** comprise any source (.c) files other than `machine.c`, so all your code **must** be in that file. You **cannot** write any new header files of your own either.

  Do not write any code in `machine.h`– your code must be in the file `machine.c`. Header files are for **definitions, not executable code. Your code will not compile on the submit server** unless all of your code is in `machine.c`. (Your **definition** of the type `Machine` and any other types it needs will be in `nelovo-datastructure.h`, but there will not be any functions with executable code in `nelovo-datastructure.h`.)

  Do not write a `main()` function in `machine.c`, because your code won't compile in that case (since our tests already have `main()` functions). Write any tests in your own separate source files, and compile them together with `machine.c`. (Don't modify the public tests, otherwise you won't be testing your code on the cases that the submit server is running. Just create your own tests in different source files.)

- The project policies handout says that in writing a project you can only use the features of C that have been covered so far, up through when the project is assigned. (As in the recent projects, you may use C operators covered in Sections 5.1.4 through 5.1.8 of the Reek text, and any C features in Chapter 4 of the textbook, even if they were not explicitly covered, for reasons mentioned in the preceding projects. This applies to all remaining projects also, so it may not be repeated every time.) You may **not** use bit fields of structures (Section 10.5 in the Reek text); they will not be covered at all this semester, and that chapter has not been covered yet anyway.
- For this project you will **lose one point** from your final project score for every submission that you make in excess of four submissions. You will also **lose one point** for every submission that does not compile, in excess of two noncompiling submissions. Therefore be sure to compile, run, and test your project's results **before** submitting it. Hopefully everyone will check their code themselves carefully, and avoid these penalties.
- Unless you have versions of all required functions that will at least compile, your program will fail to compile at all on the submit server. (Suggestion– create skeleton versions of all functions when starting to code, that just have an appropriate `return` statement.)
- In just a few keystrokes Emacs can reindent an entire program, so you would be certain to avoid losing any credit for that aspect of style grading. The UNIX tutorial explains how to do this.
- As in Project #3 you **must** use the bit operators anywhere you have to extract or isolate parts of words. **Do not multiply numbers, or divide them, or use exponentiation, to manipulate the bits of numbers– use the bit operators instead.** And when you need to extract or manipulate a part of a number **do not shift it in both directions to isolate only some of its bits– use bit masking instead**.
- Note that the project style says that global variables **may not be used** in projects unless you are specifically told to use them. **You will lose credit for using any global variables in this project.** (Read this item again.)
- If your code compiles on Grace but not on the submit server either you modified `machine.h` or your account setup may be wrong. Run `check-account-setup` to check the way your account is set up.

  If your code passes tests on Grace but not on the submit server, use `check-vars`, or inspect your code manually, to see if it is using any variables that were not previously given a value.
- Recall that the course project grading policy on ELMS says that all your projects must work on **at least half of the public tests** (by the end of the semester) for you to be eligible to pass the course.
- If you have a problem with your code and have to come to the TAs' office hours, **you must come with tests you have written yourself**– not the public tests– that illustrate the problem, what cases it occurs in, and what cases it doesn't occur in. In particular you will need to show the **smallest** test you were able to write that illustrates the problem, so the cause can be narrowed down as much as possible before the TAs even start helping you.

  You **must also have used the `gdb` debugger**, and be prepared to show the TAs how you attempted to debug your program using it and what results you got.

  Lastly, if your problem has to do with the `run_program()` function, you must have written it to print the output described in Section 3.4 when the parameter `trace_flag` is nonzero.

C Academic integrity

Please **carefully read** the academic honesty section of the syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, publicly providing others access to your project code online, or unauthorized use of computer accounts, **will be submitted** to the Office of Student Conduct, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to projects. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. More information is in the course syllabus– please review it now.

The academic integrity requirements also apply to any test data for projects, which must be **your own original work**. Exchanging test data or working together to write test cases is also prohibited.