Date due: Monday, February 28, 11:59:59 p.m.

1 Introduction and purpose

In this project you will write some functions to manipulate the instructions of a fictional simple processor created by a computer company named Nelovo, in a family of new CPUs they are developing. The Nelovo CPU has a limited set of instructions, is able to access just a small amount of memory, and is primarily intended for use in embedded systems, like a toaster for example. (The computational needs of a toaster are not high.) One purpose of the project is to use the features of C covered recently, namely bit operators and arrays, but another important objective is to introduce hardware and assembly language concepts, which will be revisited later in the course.

Note that Project #4 will extend upon this project, so you have to get the basic functionality of this project to work.

Much of this project description explains the assembly and machine language concepts involved, as well as the hypothetical Nelovo CPU. The descriptions of the functions you have to write is less than a third of the pages of this assignment.

Although the amount of code to be written in this project is not large compared to some CMSC 132 projects you may find it **significantly** harder than Projects #1 and #2, so start working on it **right away**. **Before** starting to code, look at the public tests to see how the functions are called, and look at all of the definitions in the header file `machine.h` described below. Also first study bit masking from lecture, which you need to understand very well, and if needed, ask about it in the TAs' office hours **before** starting to code. Lastly, read the **entire project completely and carefully before starting to code**. You may need to read it several times while writing the project.

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 over 475 students**. If many students wait until the last few days to write or finish projects– especially challenging ones– and have questions, the TAs will not be able to help everyone in office hours. If you are in office hours near when a project is due you may have to wait a long time, and may not even be able to get help at all. This would not be the TAs' fault; it would be because you waited until too late to do the project, given the size of the course. The only way to avoid this is starting on projects **early**, in case you do have to ask questions in office hours.

1.1 Extra credit and number of submissions

If you make **only one submission** for this project that passes **all the public tests** we will give you **5 extra–credit bonus points** on the project. Additionally, if your single submission is made **at least 48 hours before the on–time deadline** you will get **5 additional extra credit bonus points**, for 10 extra credit points total. (Obviously you can't get the second 5 extra credit points if you don't get the first 5 extra credit points, but you can get the first 5 without getting the second 5.)

In order to get any extra credit you will have to read this assignment very carefully and throughly test your functions yourself, so you are confident of doing well on the secret tests. If you find a bug later and submit again, you won't get any bonus points. You will also have to use good style all during your coding, because your single submission is of course the one that is going to be graded for style.

However, as before, if you make too many submissions you may again lose credit. To avoid making submissions that don't compile you should compile your code and fix problems **before** submitting. And to avoid making submissions that do not pass all the public tests you should run them yourself, check your output, and fix any problems, **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 Machine details and background concepts

This section explains many background concepts, while the following one describes the functions to be written.

2.1 Hardware concepts

For a program written in any computer language to be executed on a particular machine it must be translated to the instructions that the machine's processor can execute. These instructions are referred to as *machine instructions*, but we

usually just say *instructions*. Instructions manipulate information in memory and also in the processor's registers. Memory is used to hold data, such as variables, and programs themselves are also stored in memory in modern systems. Registers are locations that store values like memory, except they are internal to the CPU so they are accessed much more quickly. Registers temporarily store values that instructions operate upon, as well as the results of instructions.

The fundamental operation of any CPU is to get an instruction from memory, figure out what it says to do, perform that operation, and go on to get the next instruction; this is called the *fetch–decode–execute cycle*. (This is discussed in more detail in the Bryant & O'Hallaron reserve text.) Although the instruction sets of different processors vary widely, instructions can be categorized by functionality, for example:

computation instructions: These perform various arithmetic and logical operations.

flow of control instructions: By default instructions are executed sequentially in memory, but there are instructions that affect which instruction is executed next, for implementing conditionals and repetition, as well as function calls.

data movement instructions: These transfer data values between memory and registers, between different registers, and sometimes between different memory locations.

invoking the operating system: These instructions are used to do things like halt a program or to access parts of the hardware that user programs are not allowed to directly access, for example, I/O devices. Some instructions allow user programs to make *system calls* to get the operating system to perform these types of tasks for them.

(Section 2.3 says which Nelovo instructions fall into which categories.)

2.2 Machine specifications

The hypothetical Nelovo processor has a 32–bit (4 byte) word length, meaning that instructions, registers, and memory words are all 32 bits. (As mentioned in lecture, processors manipulate information in multibyte quantities called words.) Machines that use the Nelovo CPU are quite small, with very limited system resources. They have only 16384 bytes of memory, grouped as 4096 four–byte words ($4096 \times 4 = 16384$). The first byte in memory has address 0. Memory addresses always refer to bytes. Memory is only word–addressable, meaning that although each byte in memory has its own unique address, the CPU can only access memory in four–byte quantities, using the address of the first (or low–order) byte of a four–byte word. Consequently, the addresses that can be used to access memory are 0, 4, 8, etc., up to 16380. The value of a memory word can be interpreted as an instruction and executed, or treated as data.

The Nelovo CPU has 23 different instructions. It has fourteen registers, each of which as mentioned holds 32 bits. The first register has the number 0, through the last one that has number 13; the registers are referred to using the names `R0` through `R13`. The Nelovo has what is called a *load/store* architecture, which means that only some data movement instructions access memory. One of these instructions loads a value from a memory location into a register and another one stores a value from a register into a memory location, while all other instructions, including computations, operate upon values in registers and place their result into a register. So computation involves loading operands from memory into registers, doing calculations with them, and storing the results from registers back into memory.

The last Nelovo register (`R13`) has a special purpose. It is the *program counter*, which always contains the memory address of the next instruction to be fetched and executed. A flow of control instruction can modify the program counter to cause execution to jump to somewhere else in a program. All other instructions cause the program counter to be incremented by 4. So unless specified by a flow of control instruction, the processor executes instructions sequentially in memory. The value of the program counter register can be used by instructions but it cannot be directly modified, for example, by storing the result of a computation in it. The other registers may be used or modified by the various instructions.

2.3 Hardware, components of instructions, and instruction format

Since the Nelovo's processor has a 32–bit word length, all instructions, just like all data, must fit into a 32–bit word. When a 32–bit word is interpreted as an instruction, it is considered to consist of fields representing(a) an "opcode", i.e., operation of the instruction, (b) up to three registers used as operands by some instructions, and (c) a memory address or constant/immediate field that holds either a memory address or a constant (literal) numeric operand for some instructions. The instruction format on the Nelovo processor is as follows:

6 bits	4 bits	4 bits	4 bits	14 bits
opcode	register_1	register_2	register_3	memory address or constant/literal value

The fields have the following uses and functionality:

opcode: An opcode uniquely identifies the operation that an instruction performs. Short names are used in assembly language to refer to each type of instruction. (Recall that assembly instructions are just human–readable versions of machine instructions.) The Nelovo has 23 different opcodes, and 6 bits are used to represent an opcode. Note that 6 bits can store 64 different values and only 5 bits are actually needed to store 23 unique values, but the Nelovo company has other CPU models that use the same architecture that have more instructions, so even their CPU model in this project uses 6 bits for an opcode. Consequently, there are 41 different six–bit values that can be in the opcode field that do not represent valid opcodes.

register$_1$, register$_2$, register$_3$: As Section 2.4 says, some instructions operate upon one register, others operate on two registers, and some have three register operands. These three fields of instructions contain numbers indicating the registers that instructions use as operands. Since the Nelovo CPU has 14 registers, 4 bits can indicate any of them.

Since 4 bits can store sixteen different values but the Nelovo only has 14 registers, there are two values that can be in a register field that do not correspond to valid registers.

address or constant: Some of the Nelovo's instructions have the address of a memory location as an operand. For example, an instruction may copy the value in memory location 216 to one of the registers, so 216 would be stored in this field of the instruction. One instruction (named `li`) has a constant or literal value, typically called an *immediate* in assembly language, as an operand. For example, an instruction may store the numeric value 250 in one of the registers, so 250 itself would be encoded in the instruction.

This field is 14 bits, which is why the Nelovo has 16384 bytes of memory (with addresses $0 \ldots 16383$)– because $2^{14} = 16384$. This also means that the Nelovo CPU only has 16384 different literal (constant) values that can appear in an `li` instruction, although values up to $4,294,967,295$ can be stored in a 32–bit word.

For example, say an instruction has opcode value 3 and is an instruction that uses all three register operands, and this particular instruction is going to operate upon registers R1, R2, and R4. Also suppose this instruction doesn't use the address/constant field, and just has zeros in those 14 bits. Then this instruction would be represented in a 32–bit memory word as $0x0c490000_{16}$, which is 206110720_{10}. (Suggestion– convert the hexadecimal value to binary and match the bits against the diagram of the instruction format above.) Or suppose an instruction has opcode value 18 and uses one register operand, which is R5, and uses the memory address field to store 216_{10}. If it also had 0s in all of the unused fields it would be represented in a memory word as $0x494000d8_{16}$ (which is 1228931288_{10}).

2.4 Names, operands, and effects of the machine instructions

You need to have a basic understanding of the effects of the Nelovo's machine instructions to write the project, so after the table below, which indicates which operands are used by each instruction, their effects are briefly described. Each instruction is listed with the (decimal) number between 0 and 22 that represents its opcode, according to the `enum Operation` defined in `machine.h`. (To save space on the page the table is in two columns.) Note that the column headings reg$_1$, reg$_2$, and reg$_3$ are the three register operand fields, and "addr./imm." refers to the memory address/immediate operand field.) A checkmark means that an instruction uses that field.

opcode	value	reg$_1$	reg$_2$	reg$_3$	addr./imm.
`halt`	0				
`add`	1	✓	✓	✓	
`sub`	2	✓	✓	✓	
`mul`	3	✓	✓	✓	
`div`	4	✓	✓	✓	
`rem`	5	✓	✓	✓	
`neg`	6	✓	✓		
`and`	7	✓	✓	✓	
`or`	8	✓	✓	✓	
`not`	9	✓	✓		
`eql`	10	✓	✓	✓	
`neq`	11	✓	✓	✓	

opcode	value	reg$_1$	reg$_2$	reg$_3$	addr./imm.
`lt`	12	✓	✓	✓	
`gt`	13	✓	✓	✓	
`le`	14	✓	✓	✓	
`ge`	15	✓	✓	✓	
`test`	16	✓			✓
`move`	17	✓	✓		
`li`	18	✓			✓
`lw`	19	✓			✓
`sw`	20	✓			✓
`read`	21	✓			
`prt`	22	✓			

Different instructions use between 6 bits and 24 bits of their word (there are no instructions that use all 32 bits). Unused fields may just contain any bits at all. Also note that not every 32–bit word represents a valid Nelovo instruction.

halt: This instruction simply stops executing a Nelovo program and returns control to the operating system.

add: This instruction adds the contents of register$_2$ and register$_3$ and puts the result in register$_1$.

sub: This instruction similarly does subtraction (computes the difference of register$_2$ and register$_3$)

mul: The `mul` instruction does multiplication.

div: The `div` instruction performs division.

rem: This instruction computes the remainder of a division (modulus).

`add`, `sub`, `mul`, `div`, and `rem` all apply the operation to their second and third register operands and put the result in their first register operand. In fact, all instructions that modify a register store their result into their first register operand, so this is not always repeated below. Since a register can only store one value at a time, any instruction that modifies its first register operand's value has the effect of overwriting the previous value in that register.

For instance, the instruction `div R4 R3 R9` divides the value stored in register `R3` by the value stored in register `R9`, and puts the result in register `R4`. And `sub R10 R12 R11` subtracts the contents of `R11` from the contents of `R12` and puts the result into `R10`.

neg: This instruction negates the value in its second register operand (if it is positive it becomes negative, and vice versa), and it puts the result in the register indicated by its first register operand.

and: The `and` instruction performs logical conjunction.

or: The `or` instruction performs logical disjunction.

`and` and `or` operate upon their second and third register operands and leave the result in the register indicated by their first register operand,

not: This instruction performs logical negation; it is applied to its second register operand and puts its result in the register indicated by its first register operand.

eql: The `eql` instruction compares its second two register operands for equality.

neq: The `neq` instruction compares its second two register operands for inequality.

lt: The `lt` instruction compares its second two register operands for less than.

gt: The `gt` instruction compares its second two register operands for greater than.

le: The `le` instuction compares its second two register operands for less than or equal.

ge: The `ge` instruction compares its second two register operands for greater than or equal.

`eql`, `neq`, `lt`, `gt`, `le`, and `ge`, in conjunction with the `test` instruction explained below, allow execution to go to another location in a program. If the relationship is true about the values in their register operands they will store 1 in the register indicated by the value of their first register operand, otherwise they will store 0 in the register indicated by their first register operand. For example, suppose that register `R1` has 132 and register `R2` has 216. Then if the instruction `lt R10 R1 R2` were executed, since the value in register `R1` is less than the value in register `R2`, the effect would be that 1 would be stored in register `R10`. But `eql R10 R1 R2` would have the effect of storing 0 in `R10`.

test: This instruction examines the value of the register indicated by the value of its first register operand. If it is nonzero, the next instruction executed will be the one that is at the memory address in the instruction's address/constant field, so execution will jump to and continue from there. If the register indicated by the value of the `test` instruction's first register operand is zero though, execution just drops down instead to continue with the next sequential instruction in memory after the test instruction. Note that the `test` instruction's first register operand may have a value that was computed by one of the six comparison instructions `eql`, `neq`, `lt`, `gt`, `le`, and `ge`, but this is not required– the `test` instruction's first register operand can have been computed by any other instructions as well.

move: This instruction copies its second register operand's value to its first register operand.

li: This instruction loads a constant or literal value (`li` stands for *load immediate*) into a register. For instance, `li R4 236` would load the numeric value 236 (**not** the contents of memory location 236) into register number `R4`. (Assuming 236 is in decimal, this instruction would be stored in a word or register as 0x490000ec$_{16}$, which is 1224737004$_{10}$.)

lw: The lw instruction ("load word") copies a value from a memory location to a register. It uses the first register operand and a memory address, and its effect is to copy the value of the word in memory that has that address into the register specified by the register operand. For example, `lw R3 200` would load or copy the value that is in memory location 200 into register R3. (Assuming 200 is in decimal, this instruction would be stored in a register or memory word as $0x4cc000c8_{16}$, which is 1287651528_{10}.)

sw: The sw instruction ("store word") does the opposite of lw, copying a value from a register to the indicated memory location.

lw and sw are the only instructions that access or modify values in memory locations.

read: This instruction reads a value from the input into a register (the register indicated by the value of its register operand).

prt: This instruction prints the value stored in the register indicated by the value of its register operand to the output.

All of the instructions are computation instructions except move, li, lw, and sw, which are data movement instructions; test, which is a flow of control instruction, and halt, read, and prt, which invoke the operating system.

The same register may be used for multiple operands of an instruction, such as in `add R3 R2 R3` or `add R1 R1 R1`.

Any operand fields that aren't indicated in the table above are ignored by that instruction. For example, move has two register operands, therefore they will be contained in the first two register fields $register_1$ and $register_2$; neither the third register field $register_3$, or the address or immediate field are used by a move instruction.

Of the instructions that use the address or constant field, only li uses that field to store an immediate (i.e., constant) operand; all others use that field to store a memory address.

Note the Nelovo is such a simple machine it can only manipulate integer data, not characters, floats, or other types.

3 Functions to be written

The header file machine.h contains definitions and the prototypes of the four functions you must write. The functions use a typedef name Mach_word to represent a Nelovo word (typedef was explained in lecture recently). Mach_word is an unsigned int, since on the Grace machines an int is four bytes.

3.1 unsigned short print_instruction(Mach_word instruction)

This function should interpret its parameter as a Nelovo instruction, use the bit operators to extract its fields (see the description and diagram in Section 2.4), and print the instruction's components on a single output line. However, if the instruction is invalid as discussed in Section 3.3 below, nothing at all should be printed and the function should just return 0. Otherwise, if the instruction is valid, the function should return 1 after printing the instruction, in this format:

- The instruction's opcode (the opcode field of the parameter, meaning its leftmost six bits) should be printed using its name in the table in Section 2.4 (halt, add, sub, mul, etc.), spelled **exactly** as shown there. For example, if the value of the opcode field is the enum value HALT (which has the value 0), then halt should be printed. If the opcode field is ADD then add should be printed, etc.
- Following the opcode, the register operands **that are actually used by the instruction** should be printed, in the order $register_1$, $register_2$, and $register_3$. For example, a not instruction uses the first two registers as operands, so those two should be printed, with the first register operand followed by the second one. A read instruction uses the first register as its only operand, so just the first register operand should be printed.

  Register names should be printed in decimal with an R immediately preceding the register number. For example, registers 0, 1, and 12 would be printed as R0, R1, and R12.
- If an instruction uses a memory address operand or an immediate operand that operand should be printed last, in **decimal**. If it is a **memory address** it should be printed using exactly five places, with addresses less than 10000_{10} printed using as many leading zeros as necessary so they occupy exactly five places. For example, the address 216_{10} should be printed as 00216. (This can **trivially** be performed with printf() formatting options covered in discussion.) Note that immediate operands of li instructions should **not** be printed with any leading zeros, except zero itself should be printed as 0. Only **memory address** operands should have leading 0s if needed.

The printed fields must be separated with **one or more** tabs or spaces; the exact number of tabs or spaces (or both) is up to you. The printed instruction should **not** have any tabs or spaces before the opcode or following the last field printed. A newline should **not** be printed after the instruction.

As mentioned above, some values that can be stored in a 32–bit Nelovo word or register don't represent valid Nelovo instructions, and if its parameter represents an invalid instruction, according to the criteria in Section 3.3 below, this function should just return 0 without printing anything.

For example, the call `print_instruction(0x316ac000)` should print something like `lt R5 R10 R11`, where the exact number of spaces or tabs separating the four things printed is up to you (one or more).

(Remember that there is no such thing as a hexadecimal, decimal, or octal number in memory. There are just numbers in binary, although programmers can use constants in different bases, and programs can read or write numbers in different bases. So even though the argument passed into this function above is in hexadecimal, it is stored in memory in binary, and you can still perform bit operations on it to extract its fields, regardless of what base it was expressed in.)

Note: in projects where output has to be produced, students sometimes lose credit for minor spelling or formatting mistakes. The submit server checks your output automatically, consequently even trivial typos would cause tests to fail. Due to the size of the course we would not be able to give back any credit lost for failed tests due to spelling or formatting errors, even if they are extremely minor, because that would necessitate manual checking of every output for over 475 students' projects, which is not possible. Therefore it's essential for you to check carefully that your output is correct, register names and opcodes are spelled exactly, and that the output format described above is followed.

3.2 `unsigned short disassemble(const Mach_word program[], unsigned short max_addr)`

A disassembler converts machine language to assembly language, which is the reverse of what an assembler does. This function will do that for a Nelovo program stored in its first parameter `program`, which is an array of words representing Nelovo instructions. The function may assume that `program` is a valid C array. The parameter `max_addr` gives the Nelovo memory address of the last instruction in the array. (Note that `max_addr` is a byte address, **not** an array subscript or a number of array elements. For example, if there are three instructions in `program` the value of `max_addr` will be 8.)

The function's return value is described below. Assuming its parameter `max_addr` is valid (also described below) it should attempt to print all the instructions in the program, in this format (see some of the public test outputs for examples):

- Preceding each instruction in the array, its starting Nelovo memory address should be printed in **hexadecimal** using as many leading zeros as necessary so the address occupies exactly four places. A Nelovo program always begins at memory address 0. A colon and blank space should follow the address. In printing the memory address, remember that each array element corresponds to a four–byte word of the machine's memory.
- Each element of the array should be printed exactly how `print_instruction()` prints instructions, with each one followed by a newline (including the last one).

This is what the function should return:

- If its parameter `max_addr` is larger than the address of the last word in the Nelovo it should not produce any output at all, and just return 0.
- As discussed above, some values that can be stored in a Nelovo word or register don't represent valid Nelovo instructions. (They could represent data though.) So another possibility is that the function's parameters are valid, but some elements of the `program` array are invalid machine instructions (according to the criteria in Section 3.3). In this case all valid instructions **before** the first invalid instruction should be printed, but nothing at all should be printed for the invalid one, and the function should return 0 without printing anything after that.

3.3 `unsigned short is_valid(Mach_word word)`

As mentioned, some values that can be stored in a Nelovo word or register don't represent valid machine instructions. This function should return 1 if the value of its parameter `word` is a valid Nelovo machine instruction, and 0 if it is not valid. The reasons that a word could be invalid as a machine instruction are:

- The value in its six–bit opcode field is invalid. There are 23 instructions (hence opcodes) on the Nelovo (with enum values between `HALT` and `PRT`), so any value outside of that range doesn't represent an actual opcode.
- If a register operand that is **actually used by the instruction** has an invalid number. There are only fourteen registers on the Nelovo with numbers 0–13 (and symbolic constant names `R0` through `R13` defined in `machine.h`), so the values 14 and 15 don't represent valid register numbers.
- If it is an instruction that uses the address or constant field to store a **memory address** and the value of the field is

not evenly divisible by 4. Since everything on the Nelovo occupies a 4–byte word, and all data and instructions are accessed only in whole–word units, the only valid memory addresses are divisible by 4.

Note: if the parameters represent an instruction that uses the address or constant field for storing a **memory address** its value must be divisible by 4. But if the instruction is an `li` instruction, which is using the field to represent a **immediate** (constant) value, it does **not** have to be divisible by 4.

- If the parameters represent an instruction that has the effect of **modifying** (storing a new value into) its first register operand and `reg1` is the special unmodifiable program counter register `R13`.

  The instructions that use and modify the first register operand are all instructions **except for** `test`, `sw`, and `prt`.

  `R13` can be used as the **second or third** register operand of any instructions that use two or three registers, or as the first operand of `test`, `sw`, or `prt` instructions, but not as the **first** operand of instructions that would modify their first register operand's value.

Note: the values of fields that are **not used** by machine instructions have **no effect on validity**, and it's immaterial what they contain. For instance, a `move` instruction may have anything at all in its rightmost 18 bits (the third register operand and memory address fields). As long as the opcode, register_1, and register_2 are correct, a `move` instruction is valid. But instructions that **do** use fields, and have incorrect values in them, are invalid.

3.4 `short check_branches(const Mach_word program[], unsigned short max_addr)`

The `test` instruction is an example of what in assembly language is called a branch, specifically a conditional branch, because it can cause execution to jump to someplace else in a Nelovo program depending on the result of a condition. It is possible that the value in the memory address field of a Nelovo instruction is invalid for the program it is in, which likely represents a program bug. A programmer using the Nelovo might want to know this before trying to run the program. This function should examine all the instructions in `program` and report whether any of them are `test` instructions that have invalid addresses. The function may assume that `program` is a valid C array. `max_addr` gives the Nelovo memory address of the last instruction in the `program` array.

The function should return -1 if `max_addr` is larger than the address of the last word in the Nelovo. If the value of `max_addr` is valid it should examine the instructions in `program` that have addresses between 0 and `max_addr` inclusive, and determine whether any of them are `test` instructions that have invalid values in their memory address field:

- A `test` instruction has an invalid address if its address is not divisible by 4.
- A `test` instruction also has an invalid address if its address is larger than `max_addr`, because in this case, if the test instruction's condition (the value of its first register operand) is true (nonzero) when it is executed and it tries to jump to the address stored in its memory address field, it will be jumping beyond the last instruction in the program.

If any of the instructions is a `test` instruction that has an invalid address the function should return the memory address of the **first** `test` instruction in `program` that has an invalid address. If the program's `test` instructions are all valid, including if it doesn't have any `test` instructions at all, the function should return 1. Note that the possible values that the function can return are -1 if `max_addr` is invalid, 1 if it is valid and all of the program's `test` instructions have valid addresses, and any value 0, 4, 8, 16, etc., up to 16380, if the program has any `test` instructions with invalid addresses.

Note that **only** `test` instructions should be examined by this function. Even if any other instructions are invalid for any reason the function should not even detect this. (If the user wants to know whether other instructions are valid they can call `is_valid()` described above.)

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/project03/project03.tgz
```

This will create a directory named `project03` that contains the necessary 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 (the `cd` command above), otherwise your submission will **not** be accepted. After extracting the files from the tarfile, `cd` to the `project03`

directory, create a file named `machine.c` (spelled **exactly** that way) that will `#include` the header file `machine.h`, and in it write the functions whose prototypes are in `machine.h`.

As in Project #1 your code will not be a standalone executable; you are writing functions that will be compiled with and called from our tests, which contain `main()` functions. Consequently each public test is a different C source file, and will be compiled to form a different executable program. You can compile the tests with commands similar to those in Project #1, either from the command line or under Emacs. For example, to compile your code with the first public test, use:

```
gcc public01.c machine.c -o public01.x
```

(Or you could use separate compilation and link the resulting object files together if you want.) Adjust the command to use `public02` instead to compile your code for the second public test, etc.

Commands similar to those in Project #2 can be used to run your code and determine whether it passes a public test, except there are no input files in this project, because the tests do not read any data. For example:

```
public01.x | diff -b - public01.output
```

If no differences exist between your output and the correct output then `diff` itself will not produce any output, meaning that your code passed the test. The `-b` argument to `diff` causes it to ignore differences only in the amount of whitespace, because it's up to you how much whitespace `print_instruction()` prints between components of instructions.

As before, running `submit` from the project directory will submit your project. **Before** you submit you **must** make sure you have passed the public tests, by compiling and running them yourself, using commands like the one right above.

A.2 Grading criteria

Your grade for this project will be based on:

public tests	35 points
secret tests	45 points
programming style	20 points

Almost half of your score will come from secret tests. The public tests check only a small subset of the functionality of your code. If you don't write extensive tests yourself you could lose substantial credit on the secret tests.

Style will still be a significant part of your score for this project. If you want to avoid losing credit for style in this project and the remaining ones, carefully read the course project style guide handout. It describes the style criteria in detail. The only file that will be graded for style is `machine.c`. Keep in mind that that if you submit more than once **your last submission may not be the one that is graded** (see details in the project policies handout), so **use good style** (according to the style guide) **from the beginning** when you start working on the project, and **throughout all of your coding**.

Make sure none of your program lines are longer than 80 characters, by running your Project #2 line length check programs on `machine.c`! The Project #2 secret tests will be put on Grace so you can fix any bugs in your tab expand and length check programs and ensure their results are accurate.

B Project–specific requirements, suggestions, and other notes

- 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, like we did in the skeleton `functions.c` in Project #1.)
- 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`.

  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.)

- 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 for your project. The UNIX tutorial explains how to do this.
- As the project grading policy handout on ELMS says you should use only the features of C that have been covered so far, up through the time the project is assigned. (As in the previous project, this includes 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, which were not covered in class because they are also similar to Java, except you **cannot** use the `goto` statement.) Note 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.
- One purpose of the project is to use and get experience with C's bit operators and to understand bit masking. To avoid losing credit you **must** use the bit operators anywhere you have to access or manipulate parts of words. **Do not multiply numbers, or divide them, or use exponentiation, to manipulate the bits of numbers– use the bit operators instead.** Similarly, when you need to extract or manipulate a part of a number **do <u>not</u> shift it in both directions to isolate only some of its bits– use bit masking instead**. If you have questions about how to perform bit operations on words or to extract parts of words then ask in the TAs' office hours. (Read this item again.)
- 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 also.)
- If your code compiles on Grace but not on the submit server either you modified `machine.h` or your account setup may be wrong. To check the way your account is set up just run `check-account-setup`, and come to the TAs' office hours for help if you can't fix any problems that it identifies yourself.

  If your code passes tests on Grace but not on the submit server, remember that the very first lecture of the semester emphasized that uninitialized variables can cause C programs to work differently on different machines or different times they're compiled, and that uninitialized data is the most common source of bugs in the first part of this course. I recently showed a program `check-vars` that can say whether your program is using any variables that have not been previously initialized, which you can use to see if this is the case.
- When you are writing your own tests of your functions you will need to create Nelovo instructions. Suggestion: do not use decimal to write instructions. Write the binary representation of an instruction according to the instruction format in Section 2.3, then convert it to either octal or hex. It is straightforward to convert binary to octal or hex, but it is inconvenient to convert large binary numbers to decimal.

  It is good practice (and not difficult) to convert manually from binary to hexadecimal. But note that if you use a conversion program to convert between bases, some of them will drop leading zeros, which can be confusing.
- For this project you will **lose one point** from your final project score for every submission that you make in excess of five submissions. You will also **lose one point** for every submission that does not compile, in excess of three 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.
- 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.
- (If you don't know anything about different UNIX shells or changing your shell you can ignore this item.) As the UNIX tutorial says, the Grace systems and the account setup procedure assume you are using the `tcsh` shell. If you change your shell to another one such as `bash`, or if you write any scripts using another shell, it's up to you to make sure what you're doing is correct. The TAs cannot help with this in office hours, and we can not make any allowances in grading for errors caused by using a shell other than `tcsh`. We recommend using `tcsh` for everything in this course. (There is nothing you have to do in this course where the shell version makes any difference.)

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.