CMSC 216 Project #2 Spring 2013

Date due: Wednesday, February 27, 10:00:00 p.m.

1 Administrative

The TAs' "extra" office hours for this project will run from Monday, February 25, through Friday, March 1. During that time they will hold the higher–level office hours shown on the office hours handout.

If you make **only one submission** for this project, and that submission **passes all of the public tests**, we will give you **5 extra–credit bonus points** when your project is graded. In order to make only one submission for the project, and have it pass all the public tests, you are going to have to read this assignment very carefully, and write extensive tests of your functions yourself, so that you are confident of doing well on the secret tests– as well as use good style all during your coding, because your single submission is of course the one that is going to be graded for style.

(Note that your submission can be on time, one day late, or two days late, and you can still get the extra credit, as long as it is your only submission for the project, and it passes all the public tests.)

On the other hand, as you will see below, if you make too many submissions for this project, you may lose credit, as in the prior project. Obviously the purpose of both the extra credit and the point deductions for excess submissions is to strongly encourage students to test their code very thoroughly themselves, since some students need much more practice in doing so.

The public tests may not be provided immediately when the project is first assigned, but if not they will be available soon. The project will be set up on the submit server after the public tests are provided.

Note that one item was added to the project grading policy posted on ELMS, regarding using a blank space to separate curly braces from other language constructs, that had been overlooked before (the seventh item in Section 4.2), so be sure to follow that as well.

2 Introduction and purpose

In this project you will write several functions that manipulate the instructions of a simple hypothetical computer named the "Z86." An architecture called Y86 is actually a discussed in detail in the Bryant & O'Hallaron text, in Chapter 4. It is a simplified version of the x86 assembly instruction set that Intel processors use. This project involves a simplified version of Y86 assembly language, with some changes from Y86 just to make the project easier. Although our Z86 architecture is closely related to the text's Y86, to avoid confusion, we are just calling the machine and assembly language used in this project Z86, to differentiate it from what is described in the text. We will be returning to discuss Y86 further later this semester. This assignment explains everything you need to know about assembly and machine language to write the project, although if you want to read ahead in Chapter 4 now, knowing more about Y86 and related concepts will be helpful.

The primary purpose of the project is to write a program using the features of C covered so far; in particular, besides scalar data and basic control structures the project requires use of arrays and C's bit operations. Another objective is to become familiar with assembly language concepts, which as mentioned we will be revisiting later.

3 Background concepts

A program written in any computer language is ultimately executed on a particular machine using the basic instructions that its processor provides. Instructions manipulate information in memory and also in the processor's registers. In modern systems memory is used to hold data, such as variables in a C or Java program, and programs themselves are also stored in memory. Registers are storage locations that can store values just 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 read an instruction from memory, figure out what it says to do, perform that operation, and go on to read the next instruction; this is called the fetch–decode–execute

cycle. (This is discussed in far greater depth in the Bryant & O'Hallaron text.) Although the instruction sets of different processors vary widely, many instructions can be grouped into categories by their basic functionality:

computation instructions: Processors have instructions to perform various arithmetic and logical operations.

data movement instructions: Processors have instructions to transfer data values between memory and registers, between different registers, and sometimes between different memory locations.

flow of control instructions: Processors have instructions to affect which instruction will be executed next, to implement conditionals and iteration, as well as function calls.

In addition, facilities must be available for tasks like reading and writing data and manipulating files. Processors may also have various miscellaneous instructions that don't fit into the categories above.

4 Machine details

Some of the machine details in this section are particular to this project, and are different from the Y86 as described in the Bryant & O'Hallaron text. In implementing the project, use the information here, rather than what the text says.

4.1 Hardware and components of instructions

The Z86's processor can address 4096 bytes of memory, that are grouped into four–byte words; each word contains a value that can be interpreted either as an instruction or as data. Assume that any system using a Z86 chip will always contain this much memory. The first byte in memory has address 0, and subsequent bytes have sequential addresses (up to 4095). The Z86 architecture only has instructions to operate upon integer and character data. The Z86 has 36 different instructions, which use the following components as operands (although different instructions use different components, as discussed where the individual instructions are described in Section 4.2):

opcode: All instructions have an opcode, which is a value uniquely indicating that instruction (called an instruction specifier in the text). Its opcode is how an instruction is represented in assembly language programs. In the Z86, opcodes are one byte (eight bits) large. The term opcode is not only used to refer to the part of an instruction that indicates what operation should be performed, but to the short words that are used to identify instructions in assembly–language programs.

registers: Some instructions operate upon the contents of one or two registers, and some instructions place their results in one of those two registers. The Z86 CPU has eight registers, which have the numbers 0 through 7; they have the associated names below:

0	`%eax`
1	`%ecx`
2	`%edx`
3	`%ebx`
4	`%esp`
5	`%ebp`
6	`%esi`
7	`%edi`

memory address: Some instructions can operate upon a memory address, which, depending upon the instruction, may have the effect of using the value stored in the location with that address, storing a new value in the location with that address, or possibly causing execution to jump to that address.

immediate: Two of the Z86's instructions operate upon an immediate, which is a literal or constant value that is part of the instruction.

displacement: Two of the Z86's instructions use an immediate value as a displacement; its value is added to the contents of a register to refer to a memory address.

Like the Y86, the Z86 has what is called a *load/store* architecture. This means that the only instructions that access memory are ones that load a value from memory into a register, or one that store a value from a register into a memory location. All other instructions, such as arithmetic and bit instructions, operate upon value in registers and put their result into a register.

4.2 Effects of the instructions

The effects of the Z86's instructions are explained below. Although you don't actually need to know what they do in order to write the project, understanding them to some extent will make things much more comprehensible, and, as mentioned above, one of the purposes of the project is to start to get some familiarity with assembly–language concepts.

The number and the type of the operands used by each instruction are first given in a summary table. The exact syntax of instructions is given below, because it is necessary for one function you are to write.

opcode	value (in hex)	register_1	register_2	memory address	immediate	displacement
`nop`	0					
`halt`	10					
`rrmovl`	20	✓	✓			
`cmovle`	21	✓	✓			
`cmovl`	22	✓	✓			
`cmove`	23	✓	✓			
`cmovne`	24	✓	✓			
`cmovge`	25	✓	✓			
`cmovg`	26	✓	✓			
`irmovl`	30		✓		✓	
`rmmovl`	40	✓	✓			✓
`mrmovl`	50	✓	✓			✓
`addl`	60	✓	✓			
`subl`	61	✓	✓			
`andl`	62	✓	✓			
`xorl`	63	✓	✓			
`multl`	64	✓	✓			
`divl`	65	✓	✓			
`modl`	66	✓	✓			
`jmp`	70			✓		
`jle`	71			✓		
`jl`	72			✓		
`je`	73			✓		
`jne`	74			✓		
`jge`	75			✓		
`jg`	76			✓		
`call`	80			✓		
`ret`	90					
`pushl`	A0	✓				
`popl`	B0	✓				
`iaddl`	C0		✓		✓	
`leave`	D0					
`rdch`	F0	✓				
`wrch`	F1	✓				
`rdint`	F2	✓				
`wrint`	F3	✓				

Note that the same register may be used for both operands in instructions that operate upon two registers.

Many of the instructions' names end in 'l'; this refers to `long`, since they operate upon four–byte quantities or words (for historical reasons Intel calls a four–byte word a long).

`nop`: This instruction has no effect at all. Occasionally such an instruction is useful in assembly programming.

`halt`: This instruction has the effect of stopping the CPU.

`rrmovl`: This instruction has two register operands and copies the value in its first register operand to its second register operand. The first register is unchanged. For example, the instruction `rrmovl %edx, %ebx` would copy the value in register `%edx` to register `%ebx`. `rrmovl` stands for register–to–register move (and the 'l' stands for long, since each register stores a four–byte quantity.

`cmovle`, `cmovl`, `cmove`, `cmovne`, `cmovge`, **and** `cmovg`: These instructions are all conditional move instructions. They have two register operands and copy the value in their first register operand to their second register operand, like `rrmovl` described above, but only if a certain condition is true. They have no effect if the condition is false. We will not explain further here what the conditions are that the instructions depend upon. Note that `cmovle` performs the move if something (its condition) is less than or equal to something else, `cmovl` tests a less–than condition, `cmove` is a conditional move equal, and so on.

`irmovl`: The effect of this instruction is to store its first literal (constant) operand into the register specified by its second operand. Literals in Z86 and Y86 assembly programs are preceded by a dollar sign (`$`).

For example, the instruction `irmovl $216, %eax` would store the value 216 into the register `%eax`. From there it could be used for a later calculation or computation. `irmovl` stands for immediate–to–register move (long).

`rmmovl`: The effect of this instruction is to copy the value in its first register operand to the memory location specified by the address in its second register operand, offset by a displacement. (In other words, the sum of the register and the displacement is treated as a memory address, the location where the first operand's value is stored.) The first register is unchanged. The displacement is written as a number before the parenthesized second register operand. `rmmovl` stands for register–to–memory–move.

For example, the instruction `rmmovl %ebx, 12(%eax)` would store whatever value was in register `%ebx` to the memory location 12 bytes past the address that is in the register `%eax`.

Note that the `mrmovl` instruction described below has the opposite effect of the `rmmovl` instruction, in that it moves a value from a memory location to a CPU register, while `rmmovl` moves a value from a CPU register to a memory location.

`mrmovl`: This effect of this instruction is to copy the value in the memory location specified by the address in its first register operand, offset by a displacement, into its second register operand. The memory location's value is unchanged. `mrmovl` stands for memory–to–register–move.

For example, the instruction `mrmovl 12(%eax), %ecx` would store whatever value was in the memory address 12 bytes past the value in register `%eax` to `%ecx`.

`addl`: This instruction adds the values that are in its two register operands, and stores the sum into its second register operand. `addl` stands for add long.

For example, the instruction `addl %eax, %ecx` would store into `%ecx` the sum of the values in `%ecx` and `%eax`. And `addl %eax, %eax` would store into `%eax` the sum of `%eax` plus itself.

`subl`: This instruction subtracts the value that is in its first register operand from the value in its second register operand and stores the difference into its second register operand. For example, `subl %eax, %ecx` would store into `%ecx` the value `%ecx` − `%eax`.

`andl`: This instruction performs a bitwise "and" of the value in its two register operands and stores the result into its second register operand.

`xorl`: This instruction performs a bitwise exclusive "or" of the value in its two register operands and stores the result into its second register operand.

`multl`: This instruction multiplies the values in its two register operands and stores the result into its second register operand.

`divl`: This instruction divides the value that is in its second register operand by the value in its first register operand and stores the quotient into its second register operand. For example, `divl %eax, %ecx` would store into `%ecx` the value `%ecx` / `%eax`.

`modl`: This instruction divides the value that is in its second register operand by the value in its first register operand and stores the remainder of the division into its second register operand. For example, `modl %ebx, %eax` would store into `%eax` the value `%eax` % `%ebx`.

`jmp`: This instruction unconditionally (always) transfers execution, beginning with the next instruction to be executed, to the address specified by its memory address operand (it jumps to that location and starts executing instructions from there).

`jle`, `jl`, `je`, `jne`, `jge`, **and** `jg`: These instructions are all conditional jump instructions. Like `jmp` they can cause execution to be transferred to some other address (their memory address operand), but only if a certain condition is true. They have no effect if the condition is false, in that the next instruction to be executed will just be the next sequential following instruction in the program. We omit further explanation of what the conditions are that the instructions depend upon. Note that `jle` performs the jump if something (its condition) is less than or equal to something else, `jl` tests a less–than condition, `je` is a conditional jump equal, etc.

`call`: The effect of this instruction is to cause execution to jump to code comprising a subroutine (function), which is different from jumps of the types described above. The exact mechanism that is used to accomplish this is omitted here. It has one operand, which is the memory address to which execution should jump.

`ret`: The effect of this instruction is to cause execution to return from code comprising a subroutine, also not explained further here. It has no operands.

`pushl` **and** `popl`: These instructions will push and pop something on and off of a stack (the complete mechanism is omitted here). Both have one register operand; its value is what `pushl` will push onto the stack, and it is where the value removed from the stack will be stored by `popl`.

`iaddl`: This instruction adds a constant or literal value to the value in its register operand, leaving the sum in the register operand. For example, the instruction `iaddl $10, %eax` would increment the value in register `%eax` by 10.

`leave`: This instruction has no operands and is just a convenience; it has to do with returning from a subroutine, and using it just allows two instructions that would otherwise be required to be replaced by one instruction. Since the exact method of subroutine call and return is not described here, we omit further details.

`rdch`, `wrch`, `rdint`, `wrint`: These instructions respectively read a character from the input, write a character to the output, read an integer from the input, and write an integer to the output. They all have one register operand; its value is what will be printed for `wrch` and `wrint`, and it is where the value read by `rdch` and `rdint` will be stored.

4.3 Instruction formats

In the hypothetical Z86 instruction set **all instructions occupy a four–byte word**. (In the functions you are to write below, this will be modeled by an `unsigned int`, which on the Grace machines is four bytes.) Some instructions don't use all of the bits of their word; different instructions use between one byte and all four bytes of their instruction word. As mentioned above, the opcode is the part of each instruction that indicates which instruction it is, and on the Z86 opcodes are one byte large.

no–operand instructions: The four instructions with no operands (`nop`, `halt`, `ret`, and `leave`) just consist of their opcode, which is one byte, in the leftmost byte of their word. The remaining three bytes of these instructions are just unused.

one–register–operand instructions: The six instructions that have one register as an operand (`pushl`, `popl`, `rdch`, `wrch`, `rdint`, and `wrint`) consist of an opcode in their leftmost byte, followed by a four–bit field (the next four bits of the instruction word) whose value indicates which register is the operand. The remaining $2\frac{1}{2}$ bytes of the instruction word are just unused.

two–register–operand instructions: The fourteen instructions that have two registers as operands (`rrmovl`, `cmovle`, `cmovl`, `cmove`, `cmovne`, `cmovge`, `cmovg`, `addl`, `subl`, `andl`, `xorl`, `multl`, `divl`, and `modl`) consist of an opcode in their leftmost byte, followed by a four–bit field whose value indicates which register is the first operand, and a second four–bit field that indicates which register is the second operand. The remaining two bytes of the instruction word are just unused.

register–and–immediate instructions: The two instructions `irmovl` and `iaddl` have one register and an immediate (constant or literal) value as operands. In these instructions the one–byte opcode appears first, followed by four unused bits, then the next four bits indicate the number of the register operand (these instructions use just one register, but they use the second register bit field in the instruction). The remaining two bytes of the instruction word are the value of the immediate operand.

two–register–and–displacement instructions: The two instructions `rmmovl` and `mrmovl` have two registers and a literal (constant or immediate), that is used as a displacement, as operands. Their one–byte opcode appears first, followed by four bits for the first register operand, four bits for the second register operands, and the remaining two bytes are the value of the displacement.

memory–address–operand instructions: The eight instructions `jmp`, `jle`, `jl`, `je`, `jne`, `jge`, `jg`, and `call` have one operand, which is a memory address. Their opcode appears first, and the remainder of the instruction word is the memory address.

5 Functions to be written

The header file `machine.h` contains the following prototypes of the three functions you are to write, which are described below. As mentioned, the functions use `unsigned int` values to represent Z86 instructions, since on the Grace machines an `unsigned int` is four bytes, the same size as a Z86 instruction.

```
int print_instruction(unsigned int memory_word);
int disassemble(const unsigned int program[], int program_size);
int valid_instruction(unsigned int memory_word);
```

5.1 `int print_instruction(unsigned int memory_word)`

This function is passed an `unsigned int` as a parameter, which represents a Z86 instruction. Assuming the instruction is valid (discussed below) it should print the instruction's components on a single output line, in the format described below. Note that four instructions are exceptions to the format that should be followed for all of the other instructions; the way they should be printed is described after the general output format.

- Each instruction's opcode should be printed using the names given in Section 4.2 ("`halt`", "`nop`", "`rrmovl`", etc.), spelled **exactly** as shown there. Note that the function must extract the leftmost byte of its parameter `memory_word`, and print the proper opcode name based on the value of that byte, treated as an unsigned integer value (e.g., if the byte has the value 0 `nop` should be printed, if its value is 10_{16} `halt` should be printed, etc.).
- Following the opcode, the register operands used by that instruction, are to be printed, using the names in Section 4.1. For example, an `rdint` instruction has one register operand so just that register name should be printed, while an `addl` instruction has two register operands so both their names should be printed; the first register operand's name must be printed first. Register names are always printed with

a preceding percent sign (`%`). The function must extract the bits representing the register operand fields and use their values to determine what names to print (e.g., it should print `%eax` if a field has the value 0, while if it has the value 1 then `%ecx` should be printed, etc.).

- If an instruction has a memory address as an operand it is to be printed last, in decimal, with exactly five digits, padded with zeros on the left if its value has fewer than five digits. (The format specifier `%05d` can be used to do this.)

The four exceptions to the above are the following instructions:

`irmovl` and `iaddl`: These instructions are the only ones that have an immediate (literal) operand, and they have one register operand (which in both cases is the second register operand). The immediate operand is to be printed after the opcode, followed by the register operand, which is to be printed regularly, meaning the same as register names in any other instructions. The immediate is to be printed in decimal, with exactly five digits, padded with zeros on the left if its value has fewer than five digits, and beginning with a dollar sign (`$`). An example instruction would be `iaddl $39030, %edx`.

`rmmovl` and `mrmovl`: These instructions are the only ones that have a displacement operand, and they both have two register operands. The two register operands are to be printed, with the operand that the displacement applies to printed differently. In the `mrmovl` instruction, the displacement applies to the **first** register operand, while the displacement applies to the **second** register operand for `rmmovl`. The name of the register that the displacement does not apply to should be printed regularly, meaning the same as register names in any other instructions. For the register operand that the displacement does apply to, the register name must be printed immediately surrounded by parentheses, as in `(%eax)`. **If** the value of the replacement is **nonzero**, it is to be printed immediately **before** the opening parentheses, in decimal, with exactly five digits, padded with zeros on the left if its value has fewer than five digits. (If the displacement is zero, however, it is **not** to be printed; only the parentheses surrounding the register operand are printed.)

For example, two sample instructions are `rmmovl %edi, (%esp)` (where the displacement is zero), and `mrmovl 00004(%ebp), %esi` (where the displacement is 4).

If an instruction has two operands (this could be two registers, an immediate and a register, or a register/displacement and another register), they are to be separated by a comma followed by a single blank space (as was illustrated in various places above). The printed instruction should **not** end with any trailing whitespace or a newline character.

If the instruction is valid, the function should print it as described above, and return 1. If it is invalid, however, (the `valid_instruction()` function below describes the conditions that would cause an instruction to be invalid) the function should not print anything, and just return 0.

Note: typically in projects where output has to be produced, students sometimes lose credit for minor misspellings or formatting mistakes that cause their output to appear wrong. Unfortunately, in a class of 246 students, it is impossible for us to check the output of submitted programs manually. The submit server checks your output automatically, but this means that even trivial misspellings, differences in punctuation, etc. would cause your output to be incorrect. Due to the size of the course we will not be able to give back any credit lost due to these types of errors. even though they are minor, because that would necessitate manual checking of potentially many students' output. Therefore it will be essential for you to check carefully that your output is correct yourself, that register names and opcodes are spelled exactly, and that the output format above is followed scrupulously. It's even more important to write thorough tests yourself that exercise any functionality that is not tested in the public tests, so you can ensure that things are spelled and formatted correctly in those cases yourself (cases that might be tested in secret tests.)

5.2 `int disassemble(const unsigned int program[], int program_size)`

A disassembler converts a machine language program to assembly language, which is roughly what this function simulates, by applying `print_instruction()` to the instructions in the array `program`, in order beginning

with the first element. The second parameter `program_size` indicates how large the program is, or how many instructions in the array should be printed.

Printing of each instruction should be preceded by its memory address. Since each instruction is four bytes, and the first instruction begins at address 0, the second instruction's address is 4, the third instruction's address is 8, etc. The address should be printed in hexadecimal, with exactly three digits, padded with zeros on the left if its value has fewer than three digits; the format specifier `%03x` can be used here. (Any address on the Z86 can be printed in three hexadecimal digits.) The address should be preceded by `0x`, and followed by a colon and a single blank space. Then the instruction is to be printed, as described above by `print_instruction()`. Printing of each instruction, including the last one, must be terminated by a newline character.

If the function's parameter `program_size` has an invalid value (negative or larger than the largest possible Z86 program), the function should not produce any output, and instead should simply return 0. Otherwise the function should print the **valid** instructions in the array, beginning with its first element and attempting to print the first `program_size` elements; then return the number of instructions that were actually printed. If any element of that range of the array is an invalid instruction (the `valid_instruction()` function below describes the conditions that would cause an instruction to be invalid) it should **not** be printed– its memory address should be printed, just followed by a blank space and newline. Therefore the return value should be the number of valid instructions printed (which may range between 0 and the value of `program_size`, inclusive).

5.3 `int valid_instruction(unsigned int memory_word)`

This function should return 1 if the instruction represented by `memory_word` is valid, and 0 if it is incorrect in any way. The possible reasons that an instruction would be invalid are:

- It contains an invalid opcode. One byte is used for the opcode in instructions, but the Z86 has only 36 instructions, so some bit patterns that could be in the opcode byte just do not represent any instruction.
- It has an invalid number for a register operand, namely one outside of the range 0–7. There are eight registers, with numbers 0–7 (that have the names given in Section 4.1), but four bits are used for register operands, so some bit patterns that could be in a register operand field just do not represent a valid register number.
- It is an instruction that uses the memory address field and it has an address in that field that is larger than the largest memory address on the Z86. As mentioned in Section 4.1 the Z86 has 4096 bytes, with addresses between 0 and 4095, but values outside of that range can be represented in the bits of instructions that use a memory address.
- It is an instruction that uses the memory address field, and it has an address in that field that is not evenly divisible by 4. As will come up later in the semester, machines often have various alignment requirements, meaning that any data item must begin at a memory address that is a multiple of the size of that data item. Since every instruction and data item on the Z86 occupies exactly 32 bits, every data item must begin at a byte address which is divisible by 4. Therefore, the memory address operand of any instruction that refers to a memory location must be a byte address divisible by 4 in order to be valid.

Note that the values of fields that are **not used** by instructions have **no effect on validity**, as it's immaterial what they contain. For instance, an `addl` instruction may have any bits at all in its rightmost two bytes; so long as the opcode and the values in the two register fields are correct the instruction is valid. But instructions that **do use** fields, but have incorrect values in them, are invalid. So if an `addl` instruction had an invalid value in either of its register operand fields it would be invalid.

6 Example

The program on the left below illustrates the use of two of the functions you are to write. Its output, if the functions above are correctly implemented, when compiled and run, is shown on the right below.


```
#include <stdio.h>
#include "machine.h"

#define SZ 5

int main() {
  unsigned int program[SZ]= {0x20340000,
                             0x62340000,
                             0x30050084,
                             0x50560004,
                             0x10000000};

  printf("Illustrating print_instruction():\n");
  print_instruction(0x60320000);
  printf("\n");
  print_instruction(0x800000c8);
  printf("\n\n");

  printf("Illustrating disassemble():\n");
  disassemble(program, SZ);

  return 0;
}
```

```
Illustrating print_instruction():
addl %ebx, %edx
call 00200

Illustrating disassemble():
0x000: rrmovl %ebx, %esp
0x004: andl %ebx, %esp
0x008: irmovl $00132, %ebp
0x00c: mrmovl 00004(%ebp), %esi
0x010: halt
```

This program is an example but will not necessarily be one of the public tests. Of course this one example does not show all of the circumstances that the functions must work in.

A Development procedure review

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

You can obtain the necessary project files by logging into one of the Grace machines using commands similar to those used in Project #1:

```
cd ~/216
cp ~/216public/project2/project2.tgz .
tar -zxvf project2.tgz
```

The tarfile will create a directory `project2` that contains the necessary files for the project, including the header file `machine.h` and the public tests. You **must** do your coding in your extra course disk space (using the `cd` command above) for this class, otherwise we **will not accept** your submission. After extracting the files from the tarfile, `cd` to the `project2` directory, create a file named `machine.c` that will `#include machine.h`, and write the functions whose prototypes are in `machine.h` in that source file.

Compile your code using commands similar to those used in Project #1:

```
gcc public1.c machine.c -o public1.x
```

Of course, modify the command to use `public2.c` instead to compile your code for the second public test, etc. This command will work right only if you set up your account properly in discussion section. If you did, and followed the steps to check your setup and everything worked, an alias was created for `gcc` that adds the required options `-ansi`, `-pedantic-errors`, `-Wall`, and `-Werror` to the command.

Commands just like those in Project #1 can be used to run your code and determine whether it passes a public test or not, for example:

```
public1.x | diff -u - public1.output
```

If no differences exist between your output and the correct output diff will produce no output, and your code passed the test.

A.2 Submitting your program

As before, the command `submit` will submit your project. **Before** you submit, however, you must first check to make sure you have passed all the public tests, by running them yourself. And after you submit, you **must** then log onto the submit server (linked to from ELMS under Pages, or use the link in the Project #1 PDF), and check whether your program worked right on the public tests there.

As mentioned in Project #1, unless you have versions of all required functions that will at least compile– whether they are used in the public tests or not– your program will fail to compile at all on the submit server. (You don't have to have complete working versions of all the functions to submit, but you do have to have versions that will at least compile, for example skeleton functions that just contain an appropriate `return` statement for any non–`void` functions.)

Do **not** submit programming assignments using the submit server's mechanism for uploading a jarfile or zipfile or individual files. Do not wait until the last minute to submit your program, because the submission server enforces deadlines to the exact second.

Project extensions will not be given to individual students as a result of hardware problems, network problems, power outages, etc., so you are urged to finish **early** so any such situations will not affect you even if they do arise.

A.3 Grading criteria

Your project grade will be determined according to the following weights:

Results of public tests	35%
Results of secret tests	50%
Code style grading	15%

The public tests will not be comprehensive, and you will need to do testing on your own to ensure the correctness of your functions.

A.3.1 Style grading

The only file that will be graded for style is `machine.c`. Be sure to **carefully read** the course project grading policy posted on ELMS, which contains all details about the criteria that will be used for style grading, so you can follow them and avoid losing credit unnecessarily.

B Project–specific requirements

- You **cannot** modify the declaration of 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.

- As the project grading policy says, you should use only the features of C that have been covered in class, or that are in the chapters covered, up through the time the project is assigned.

- In addition to the items that good style is considered to consist of discussed in the project grading policy, when you need to extract or manipulate parts of numeric types in this project you should not shift them in both directions; use bit masking instead.

- For this project you will **lose one point** from your final project score for every submission that you make in excess of 10 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. We hope everyone will check their code themselves carefully, and no one will incur these penalties.

C Other notes

- You may assume that the array parameter of `disassemble()` is a valid C array, that has at least `program_size` elements.
- **Write your own tests, and test each function as you write it, before going on!**
- 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 that illustrate the problem (tests that you wrote in addition to the public tests). In particular you will need to show the smallest test you were able to write that illustrates the problem, so whatever the cause is can be narrowed down as much as possible before the TAs even start helping you. You must have also used the `gdb` debugger, explained recently in discussion section, and be prepared to show the TAs how you attempted to debug your program using it and what results you got.
- Recall that the course syllabus says that all your projects must work on **at least half of the public tests** (by the end of the semester) in order for you to be eligible to pass the course. See the project grading policy for full details. (The compilation penalties mentioned above only apply to submissions made before the end of the project late submission period.)

D Academic integrity

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, 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 project assignments. 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. Full information is found in the course syllabus– please review it at this time.

