CSI 333 – Programming at the Hardware/Software Interface – Fall 1999
Programming Project 3

WARNING: Read and follow the instructions herein about Revision Logs before you start this project. Otherwise, you might get ZERO credit or face more serious consequences.

1 Objectives:

1. Design, implement and debug software with multiple functionality by carefully planning the order in which the functions will be added. Analyze command line arguments and store the resulting information to control future operations the software performs.

2. Gain experience with simulation software and improve understanding of instruction set architecture (hardware) operation.

3. Process text “formatted” files and files of arbitrary contents (binary or “unformatted” files).

4. Continue to use the modularization standard which separates interfaces and implementations by using `.h` and `.cpp` files. Write “Makefiles” that automate safe and efficient builds and express the build dependencies.

5. Yes, you can use all standard library functions for this project, including those for C strings in `<string.h>` or C++ strings if you want to learn about them.

2 Advice

Before you attempt to write even one line of code, you must read and analyze the problem statement ("specification") to understand what your program is supposed to do, and think out design and implementation plans go about creating the program. Read the whole assignment and submit any questions to the lecture, the newsgroup (`sunya.classes.csi333`), a TA or the professor. Waldron’s chapter 6 and PH sections 3.4 and 3.8 may clarify the project’s subject.

Devise a collection of modules each of which solves a separate problem and make a plan for adding new functionality to partially implemented modules. Non-trivial programming work projects are best done incrementally. This means first write, test, debug and test again (after every change, no matter how little it is) a version that meets just a few of the specifications correctly. The first and subsequent versions enable you to add new functionality a little at a time, in such a way that each time new code is added, you can trust the previously completed code to help test and debug the new code.

Copies of all tested previous versions should be kept. They of course have NO SYNTAX ERRORS (the assembler or compiler accepts and successfully runs or compiles them) because they had to have been assembled or compiled for testing! This way, if you mess up the newer version beyond repair, you can start again with a previous version.
3 Problem

The problem is to write a simulator called tmips for a “toy” simplification of the MIPS computer called TMIPS. Your “makefile” should orchestrate commands to build an executable file named tmips. tmips will analyze strings from its command line, record the information from the analysis to determine what to do, and then perform a combination of one or more major operations according to the description of the command line arguments below.

TMIPS memory will be have one segment with byte addresses 0x0 to 0x3FF which therefore consists of 1024 (1K) bytes. Like typical hardware, the memory will be accessed in units of 32 bit words, thus it should be simulated by an array of 256 unsigned long ints because unsigned long int is the C/C++ datatype that provides (at least) 32 bit values to be manipulated as bitstrings.

TMIPS memory is to be initialized so every word contains 0xFFFFFFFF. This way you are likely to observe memory locations have not be modified by the loader functions or simulated TMIPS operation.

To determine the contents of the TMIPS memory before the simulation is run, the tmips program will read either a file of ASCII coded characters or a file 1K bytes of “unformatted” or “binary” data.

The command line options are accessed through the arguments to main:

```c
int main( int argc, char *argv[] )
```

See Stroustrup p.117-118 for the data structure description. Test test if the C string pointed to by argv[i] (where 1 ≤ i < argc) is the same as an option name such as --in-hex with code like:

```c
#include <string.h>
...
if( strcmp( argv[i], "--in-hex" ) == 0 )
{
    /* The C string pointed to by argv[i] equals --in-hex. */
}
```

String of decimal digits can be converted to integers with strtol() (See the man page man strtol in the ECL.)

The command line options can be given in any order. Of course, the next argument after each option that requires a file name or decimal number must be the file name or number. Such options cannot be repeated. Unspecified options will not be tested. (Therefore, you can invent your own options to facilitate control of testing/debugging features.)

The program should process all the options at the beginning and use an appropriately designed set of global variables or an “option information structure” to record options and their values. (You can simply use the pointer copied from argv[i] to a filename string to refer to the filename in the future; copying the string is unnecessary.) After processing the whole argument array, check the option settings for contradictions.

Whenever an error is detected, the program must print (in CAPITAL letters, on a line by itself):

```
ERROR:
```
and, on the next line, a description of the error which you make up yourself. Then, it should exit. (Errors not reported this way will lose credit because of the automated grading system.)

---

1There is nothing special about the names argc and argv. The int number of arguments including the command name itself is main’s first parameter. main’s second parameter is a char ** pointing to an array of character pointers.

2The strtol function very useful for converting into binary integers strings representing integer numerals in various bases, which is done by assemblers and compilers. The Standard C++ library class istringstream also provides formatted conversions between strings and C++ data types in an extendable but more intricate way that is harder to learn to use.
Assuming there are no errors, tmips must print (in CAPITAL letters, on a line by itself):

PLAN:
followed by a one line description of each major operation that should be performed according to what the options specify. For example:

Read binary memory image file *filename*
Simulate *n* instructions with reg. tracing.
Output a post dump.
Quit

An error must be reported if opening of the specified input or output file fails.

1. **--in-hex** *filename*
   Open a file named *filename* containing TMIPS instructions and data to be loaded into memory. The “hex” file consists of lines of two hexadecimal numerals each: The first is a word address (number in the 0 to 3FC range which is a multiple of 4) and the second is the contents (number in the 0 to FFFFFFF range) to be loaded into memory at the given address.

   TMIPS memory (like MIPS) is byte addressed, but all instructions and data words are 32 bits long and must be aligned to 4 byte address boundaries.

   The addresses in the “hex” file will not necessarily be consecutive word addresses. If two or more lines with the same address are given, the last line should determine the addressed memory location’s contents. (This makes the program easier.)

   You can assume the input file will be in the correct format, but you can design your own specification for handling illegal input. (Some cases of illegal input include out of range or unaligned addresses, characters other than whitespace, digits or letters a-f, A-F, and missing data values.)

   To set the format state of the input stream to interpret all integer numerals in base Sixteen, use the standard manipulator hex (or the setf() member function):

   ```
   MyInputStream ifstream(/* pointer to *filename* char array ... */);
   if( !MyInputStream ) { /* file cannot be opened */ };
   MyInputStream >> hex;  // Use base Sixteen for future integer numeral
   // translation to binary.
   MyInputStream.setf( ios::hex, ios::basefield );
   // Alternative way to set input base.
   ```

2. **--in-bin** *filename*
   Open a file named *filename* that contains a 1Kbyte (1024 byte) TMIPS memory image and read it into memory. The memory image is to be read into simulated TMIPS memory beginning at address 0.

   **Exactly one of the --in-hex and --in-bin options must be given.**

   To read an unformatted file into an array of words, use the `istream read( char * POINTER, COUNT )` member function of the ifstream opened to read the file. See Stroustrup p.619 and your other C++ books if necessary, or consult emacs info. (In the ECL, give the “C-h i” command to emacs to enter the info system, select the “iostream” page, and type “g String Input”.)

   Our “struct memory” uses a “unsigned long int *pwords” (word) pointer to refer to the array that simulates memory. Since the unformatted `read` function takes a char pointer
because it reads in terms of bytes, you might (for some C++ library implementations) need to code a cast to tell the compiler to treat the word pointer as a char pointer. To do that, code

\begin{verbatim}
MyIStream.read( static_cast<char*>(words), COUNT )
\end{verbatim}

where COUNT (for this project) is 1024 (bytes).

3. --pre-dump
   After reading hex input or a memory image (but \textbf{before} any simulation), dump the “simulator state” to stdout. This means print the contents of all the registers and memory locations. The register contents must be printed first by calling the member function \texttt{registers::dump()} of \texttt{struct registers}. Second, the memory contents must be printed by calling the member function \texttt{memory::dump()}.

4. --out-bin \texttt{filename}
   After reading hex input or a memory image (but \textbf{before} any simulation), write a 1Kbyte memory image file containing a copy of the TMIPS memory contents produced by the assembly or the memory image read.

   You can test the --in-bin function by using the \texttt{diff} command to tell if the original image file and the file written by --out-bin have the same contents.

   You can check the output of your hex input reader by looking at the contents of the file written by --out-bin in two ways:

   \begin{itemize}
   \item Open the file with emacs and give the \texttt{ESC-x hex1-mode} command to the buffer viewing the file.
   \item Run the Unix utility \texttt{od} on the file. If you are using the FSF version (available on Linux systems), use the options:
   \begin{verbatim}
   od --format=x4 --address-radix=x \texttt{filename}
   \end{verbatim}
   In the ECL, use:
   \begin{verbatim}
   od -Ax -tx4 \texttt{filename}
   \end{verbatim}
   \end{itemize}

   To easily observe the operation of the \texttt{od} utility: (\texttt{od} stands for “octal dump.”)

\begin{verbatim}
echo "ABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890" | od -Ax -tx4
\end{verbatim}

5. --simulate \texttt{n}
   After loading the memory, and after any image file writes or state dumps requested, simulate the TMIPS processor until either \texttt{n} instructions have been simulated, a halt instruction was executed or an error is detected. \texttt{n} is expressed in decimal.

6. --post-dump
   After the simulation stops, dump the “simulator state” to stdout as specified for --pre-dump.

7. --trace-regs
   Dump the simulated register contents immediately after simulating each instruction.

4 Simulation

Simulator software must somehow represent as data those aspects of the “real world” that are being simulated. Thus, the first step in designing a simulator is to determine which aspects to simulate.
The next step is to design data structures to represent and store them. The collective content of those data structures (that is, the values stored in them) is called the state.

**State representation** and operations to change the current state are the key elements of simulation. Note that state is written first because conceptually, it is first.

The state of TMIPS consists of its memory contents plus the contents of “registers”. The registers consist of:

1. **PC** (Program Counter) The address of the instruction being simulated.
2. **IR** (Instruction Register) A copy of the instruction being simulated.
3. **regs[]** (Architectural Registers) The 32 general purpose (GPR[0] to GPR[31]) integer registers of MIPS. When the state is printed, the Architectural register display should show the register contents after the instruction at PC and displayed in IR has been simulated.³
4. **newPC** (New PC) The address of the next instruction to be simulated. Branch and jump instructions set newPC according to their operands. Other instructions set newPC to PC+4.
5. **CPU state** (Simulator Run State) **RUNNING** if the simulator should simulate the instruction pointed to by newPC after the current instruction simulation is finished. If an error is detected, the count runs out or current instruction halts the simulator, state should be set to HALTED.

To get everyone started with working and uniform state printing facilities, we have implemented two struct's with member functions for you to use in your project. The header, implementation, test driver/demonstration programs and a “makefile” are now available in the ECL under ~csi333/Project3/StateClasses

You can add your own function members to the “struct registers” and “struct memory” classes (remember a C++ “struct” IS a C++ class) and write their implementations in files separate from registers-dump-init.C and memory-dump.C respectively.

Our “dump()” member functions of “struct memory” and “struct registers” MUST be used to perform the dumps specified for the --pre-dump, --post-dump and --trace-regs options.

You must copy our registers-dump-init.C, memory-dump.C, registers.h and memory.h files into your project3 directory. After possibly modifying memory.h and registers.h, you must submit all these files together with your own files and the “Makefile” so that we can grade your work by building your software completely from the source files you submitted. (Except for system header files and libraries, of course).

## 5 Toy Output Device (TOD)

TMIPS has a toy output device connected to its memory bus in addition to its 1KByte memory. Writing a word to address 0xFFFFF8C (also known as -4), which is done with a SW instruction, causes the word to be printed (use cout << ) in decimal.

Writing a word to address 0xFFFFF88 (aka -8) causes the low order byte of the word to be printed as a character. Extract using an AND mask and cast to char:

```cpp
cout << static_cast<char>( 0x000000FF & word );
```

³YOU must implement the special behavior of register 0: It is not an error to write anything to it but all reads of register 0 read out value 0.
Writing a word to address \texttt{0xFFFFFFFF} (aka -12) causes the word to be printed in hexadecimal:
\begin{verbatim}
cout << hex << word << dec;
\end{verbatim}
The \texttt{dec} manipulator is to restore the default. Better practice is to use the format flag save/restore technique of Stroustrup page 626. Use \texttt{ios} instead of \texttt{ios\_base}.

Reading from the Toy Output Device is an error, of course.

6 TMIPS instructions

The TMIPS instructions are fashioned after real MIPS but do have simplifications appropriate to their toy nature. You can therefore use \texttt{(x)spin} to help generate test input.

A new PC value is obtained after the simulation of each instruction is finished. Except after branch and jump instructions, the new PC value will be \texttt{PC+4}. The PC value may become illegal as a result of control flowing beyond the 1KByte TMIPS memory or as the result of a branch or jump. Whenever a simulation step begins with PC containing a value that is not a legal TMIPS instruction address, the simulation should stop, the message:\n
ERROR:

should be printed on a single line, and a one line explanation of the error in your own words should be printed below it. An address is an illegal instruction address if it is not 4-byte aligned or if it is not in the 0 to \texttt{0x3FC} range.

Another error that causes simulation to stop is the attempted execution of an unrecognized instruction (unspecified \texttt{opcode} or \texttt{opcode}=0/\texttt{funct} combination). The most common unrecognized instruction will be \texttt{0xFFFFFFFF}.

Simulation should also stop after the \texttt{n}th instruction simulation, where \texttt{n} is the parameter of the \texttt{--simulate} option. In that case, instead of ERROR: print on one line: \texttt{COUNT FINISHED}.

Again, whenever the simulation stops, any post simulation operation requested by a command line action should be performed.

The values of the \texttt{funct} or \texttt{opcode} fields, which determine the instruction's identity, are written below in \texttt{decimal}.

6.1 Three Register ALU Instructions

Instructions: ADD, OR, SUB, AND
Simplification: TMIPS does not detect arithmetic overflow.

<table>
<thead>
<tr>
<th>bit field widths</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>000000</td>
</tr>
</tbody>
</table>

\begin{tabular}{|c|c|c|c|c|c|}
\hline
\text{opcode} & \text{source reg} & \text{source reg} & \text{destination reg.} & \text{function code} \\
\hline
\end{tabular}

bit field interpretations

1. \texttt{ADD}  \hspace{1cm} \texttt{funct = 32}

Retrieve the 32-bit values from the registers named in the \texttt{rs} and \texttt{rt} fields, add them, and
put the sum in the register named in the rd field. More succinctly expressed:

\[ \text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] + \text{GPR}[\text{rt}] \]

Note for all instructions, \( \text{rd} = 0 \) is legal but \( \text{GPR}[0] \) must always (seem to) contain 0.

2. SUB \quad \text{funct} = 34
   Similar to ADD:
   \[ \text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] - \text{GPR}[\text{rt}] \]

3. OR \quad \text{funct} = 37
   Similar to ADD:
   \[ \text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] \lor \text{GPR}[\text{rt}] \]
   The operation is bitwise OR. That is expressed in C++ with the \( \lor \) (single vertical bar) operator. (In C++, \( \| \) denotes “boolean OR” which returns 0 if both operands are 0 and the integer 1 (value one) otherwise.)

4. AND \quad \text{funct} = 36
   Similar to ADD:
   \[ \text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] \land \text{GPR}[\text{rt}] \]
   The operation is bitwise AND. That is expressed in C++ with the \( \& \) (single ampersand) operator. (In C++, \( \& \) denotes “boolean AND” which returns integer value 1 if both operands are 1 and 0 otherwise.)

### 6.2 Memory Access Instructions (loads and stores)

Instructions: LW and SW

<table>
<thead>
<tr>
<th>opcode</th>
<th>base</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bit field widths</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bit field interpretations</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
</tr>
</tbody>
</table>

1. LW \quad \text{opcode} = 35
   The 16 bit offset is interpreted as an integer in the 16 bit 2-s complemented signed way. This integer is added with the contents of \( \text{GPR}[\text{base}] \) to form the effective address. The contents of memory at the effective address is retrieved and copied into \( \text{GPR}[\text{rt}] \).
   C++ code to sign extend the offset after extraction into a \textbf{signed long int} variable can be written in a portable way as follows:
   (a) Test if bit 15 is non-zero by ANDing with the mask \( 0x00008000 \).
   (b) If bit 15 is non-zero, use bitwise OR with \( 0xFFFFFFFF \) to insert the extended sign bits.
If the effective address is not a multiple of 4 (i.e., bottom 2 bits are non-zero) or it addresses a memory location that doesn’t exist in the simulation, print:

ERROR:
on a line by itself. Then, on the next line, print an informative description of the error in your own words. Make the CPU simulation stop by setting the “CPU state” to HALTED. Print the trace-state report and/or the post-dump if they had been requested. Finally, exit.

2. SW opcode = 43
Simulate behavior similar to LW except store the data from register rt into memory at the effective address instead of loading. Implement similar error detection and behavior.

6.3 Jump Instructions
Jump and Branch instructions cause the PC to get a new value different from PC+4. Like SPIM but not like real MIPS, TMIPS jumps and branches are not delayed. Also, (as in SPIM) branch offsets are relative to the address of the branch instruction, not its delay slot.

Instructions: J, JAL. (Format for JR below is different.)

\[
\begin{array}{|c|c|}
\hline
\text{bit field widths} & \text{target} \\
\hline
6 & 26 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{opcode} & \text{target} \\
\hline
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{bit field interpretations} \\
\hline
1. J \quad \text{opcode} = 2 \\
\text{Shift the target value left 2 bits and unconditionally jump to the address so obtained.} \\
\text{Simplification: Since TMIPS addresses use at most 9 bits, only the low 7 bits of the J instruction can be non-zero.} \\
\text{Make newPC contain the target address instead of PC+4.} \\
\hline
2. JAL \quad \text{opcode} = 3 \\
\text{In addition to the actions of Jump, the Jump and Link instruction copies the return address into GPR[31]. The return address is PC+4; here, PC contains the address of this JAL instruction.} \\
\hline
3. JR \\
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{bit field widths} & \text{target} \\
\hline
6 & 5 & 15 & 6 \\
\hline
000000 & rs & 00000000000000000000 & 001000 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{bit field interpretations} \\
\hline
\end{array}
\]
The function code is 8 (shown in binary above.) Jump to the instruction whose address is in GPR[rs]. Detect the usual errors for control transfer instructions.

6.4 Branches Conditional on Register Equality

<table>
<thead>
<tr>
<th>bit field widths</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>opcode</td>
</tr>
</tbody>
</table>

opcode data reg data reg offset/4 from PC for computing target addr.

bit field interpretations

1. BEQ opcode=4
If GPR[rs] == GPR[rt], the offset is sign extended, shifted left 2 bits, and added to the PC value to obtain the address of the next instruction to execute.

2. BNE opcode=5
Similar to BEQ except the branch is taken if GPR[rs]! = GPR[rt].

6.5 Branches Conditional on Register Sign

<table>
<thead>
<tr>
<th>bit field widths</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>opcode</td>
</tr>
</tbody>
</table>

opcode data reg offset/4 from PC for computing target addr.

bit field interpretations

1. BLEZ opcode=6
Similar to BEQ except the branch is taken if GPR[rs] <= 0. The “rt” field should be 0. However, for TMIPS, don’t bother checking.

2. BGTZ opcode=7
Similar to BEQ except the branch is taken if GPR[rs] > 0. The “rt” field should be 0. However, for TMIPS, don’t bother checking.

6.6 ALU Instructions with an Immediate Operand

Instructions: ADDI, ANDI, ORI
Simplification: TMIPS ADDI does not detect arithmetic overflow.
bit field widths

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
</tr>
</tbody>
</table>

opcode source reg dest reg Immediate value used for one source operand.

bit field interpretations

1. ADDI  opcode=8
   The 16-bit immediate value is sign extended to 32 bits, added with the contents of GPR[rs], and the result is copied into GPR[rt]

2. ANDI  opcode=12
   The 16-bit immediate value is zero extended to 32 bits, bitwise ANDed with the contents of GPR[rs], and the result is copied into GPR[rt]

3. ORI   opcode=13
   The 16-bit immediate value is zero extended to 32 bits, bitwise ORed with the contents of GPR[rs], and the result is copied into GPR[rt]

6.7 Logical Shifts

Instructions: SLL and SRL

bit field widths

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>funct</td>
</tr>
</tbody>
</table>

opcode source reg dest reg shift amount function code

bit field interpretations

1. SLL  funct=0
   Shift the contents of GPR[rt] LEFT (use the C++ operator <<) by sa bits (inserting zeros into the low order bits) and copy the result into GPR[rd].
   Note that SLL with rt=rd=sa=0 is the instruction with value 0 and it functions as an operation that does nothing.

2. SRL  funct=2
   Shift the contents of GPR[rt] RIGHT (use the C++ operator >>) by sa bits inserting zeros into the high order bits and copy the result into GPR[rd]. To make this happen in C++, apply the shift to an variable of type unsigned long int.
7 Revision Log and Submission Checklist

You must maintain a “revision log”. The revision log is to be maintained in a file named revisions.txt which must be submitted with all your source files (every .cpp and every .h plus the Makefile) needed for us to build the software for testing.

At the beginning of the revisions.txt, after your name, student ID number and TA name, write a comment for each time you (1) began writing a new version, (2) fixed a bug, and (3) concluded by testing that the current version is correct; the comment must specify the version number, date and approximate time.

A submission without a revision log will receive ZERO credit, and logs found to be faked will be treated as evidence of CHEATING.

A word to the wise: Run make and do a final test immediately before submitting your work using turnin. **If you edit anything, DO NOT TURN IT IN** until after you remake the software and test it again. The remake will verify your edits (or attempts to restore old code, etc) did not introduce syntax errors.

**IF WE CANNOT BUILD YOUR SOFTWARE from sources only (no objects or executables) BECAUSE OF SYNTAX OR UNDEFINED SYMBOL ERRORS, YOU WILL GET NO CREDIT FOR PROGRAM FUNCTIONS**

8 Grading etc.

1. The due time is Monday, December 6, 9:00PM.

2. Late submissions will be accepted until Friday, December 10, 9:00PM 9:00PM (4.000 days). However, a lateness penalty factor equal to (10.0 - ND)/10.0 where ND is the amount of time late, measured in days, will be multiplied into your score, computed using floating point arithmetic. This means the amount deducted for lateness will begin at 0% and rise (almost) continuously to 40% during the late turnin period. Early submissions will be accepted but will not earn any bonus.

3. 70% of the score will come from evaluation of outputs on test cases. To get any of these points, your software must build to an executable named tmips when we run “make tmips” in a directory with all the .c, .cpp and .h files that you had submitted. All object (.o) files and executable files will be deleted before this is done; and points will be deducted if any such files were submitted. Give it a final test before you submit it; do NOT open it with an editor AT ALL after the final test.

4. 30% of the score will come from a physical organization that reflects a logical design in which separate problems are solved by separate modules, and internal documentation:

- Quality (clarity, accuracy, completeness, etc.) of revision history.
- Consistent indentation.
- Procedures/functions: What each does in terms of parameter register contents, return value, and action on any other data it uses.
- Header files containing interface definitions and documentation only, and other CS1333 software engineering standards in the lecture notes.
- The dependencies are accurately coded in the Makefile.
- The Makefile must be “simple” in the sense of Lab Exercise 3. DO NOT SUBMIT a “template” from a previous or other course! If you are a real Make expert, creating a simple makefile for this project will be easy. If not, follow the directions of Lab Exercise 3.