The Assignment

Recall that an assembler translates code written in mnemonic form in assembly language into machine code.

You will implement an assembler that supports a subset of the MIPS32 assembly language (specified below). The assembler will be implemented in C and executed on Linux. Your assembler will take a file written in that subset of the MIPS32 assembly language, and write an output file containing the corresponding MIPS32 machine code (in text format).

Supported MIPS32 Assembly Language

The subset of the MIPS assembly language that you need to implement is defined below. The following conventions are observed in this section:

- The notation `(m:n)` refers to a range of bits in the value to which it is applied. For example, the expression

  `(PC+4)(31:28)`

  refers to the high 4 bits of the address `PC+4`.
- `imm16` refers to a 16-bit immediate value; no assembly instruction will use a longer immediate
- `offset` refers to a literal applied to an address in a register; e.g., `offset(rs)`; offsets will be signed 16-bit values
- `label` fields map to addresses, which will always be 16-bit values if you follow the instructions
- `sa` refers to the shift amount field of an R-format instruction; shift amounts will be nonnegative 5-bit values
- `target` refers to a 26-bit word address for an instruction; usually a symbolic notation
- Sign-extension of immediates is assumed where necessary and not indicated in the notation.
- C-like notation is used in the comments for arithmetic and logical bit-shifts: $>>_a$, $<<_1$ and $>>_1$
- The C ternary operator is used for selection: `condition ? if-true : if-false`
- Concatenation of bit-sequences is denoted by `||`.
- A few of the specified instructions are actually pseudo-instructions. That means your assembler must replace each of them with a sequence of one or more other instructions; see the comments for details.

You will find the *MIPS32 Architecture Volume 2: The MIPS32 Instruction Set* to be an essential reference for machine instruction formats and opcodes, and even information about the execution of the instructions. See the Resources page on the course website.

MIPS32 assembly .data section:

`.word` This is used to hold one or more 32 bit quantities, initialized with given values. For example:

```
var1:   .word 15      # creates one 32 bit integer called var1 and initializes
                      # it to 15.

array1: .word  2, 3, 4, 5, 6, 7  # creates an array of 6 32-bit integers,
                                 # initialized as indicated (in order).

array2: .word 2:10    # creates an array of 10 32 bit integers and initializes
                      # element of the array to the value 2.
```

`.asciiz` This is used to hold a NULL (0) terminated ASCII string. For example:

```
hello_w: .asciiz "hello world"   # this declaration creates a NULL
                                 # terminated string with 12 characters
                                 # including the terminating 0 byte
```

MIPS32 assembly .text section:

Your assembler must support translation of all of the following MIPS32 assembly instructions. You should consult the *MIPS32 Instruction Set* reference for opcodes and machine instruction format information. That said, we will evaluate your output by comparing it to the instruction specifications in the MIPS32 Instruction Set manual.

Your assembler must support the following basic load and store instructions:

```
lw      rt, offset(rs)          # Transfers a word from memory to a register
                                # GPR[rt] <-- Mem[GPR[rs] + offset]

sw      rt, offset(rs)          # Transfers a word from a register to memory
                                # Mem[GPR[rs] + offset] <-- GPR[rt]

lui     rt, imm16               # Loads a constant into the upper half of a register
                                # GPR[rt] <-- imm16 || 0x0000
```

Note that the constant offset in the load and store instructions (`lw` and `sw`) may be positive or negative.

Your assembler must support the following basic arithmetic/logical instructions:

```
add     rd, rs, rt              # signed addition of integers; overflow detection
                                # GPR[rd] <-- GPR[rs] + GPR[rt]

addi    rt, rs, imm16           # signed addition with 16-bit immediate;
                                # overflow detection
                                # GPR[rt] <-- GPR[rs] + imm16

addu    rd, rs, rt              # unsigned addition with 16-bit immediate;
                                # no overflow detection
                                # GPR[rt] <-- GPR[rs] + GPR[rt]

addiu   rt, rs, imm16           # unsigned addition with 16-bit immediate;
                                # no overflow detection
                                # GPR[rt] <-- GPR[rs] + imm16

mul     rd, rs, rt              # signed multiplication of integers;
                                # no overflow detection
                                # GPR[rd] <-- (GPR[rs] * GPR[rt])(31:0)

mult    rs, rt                  # signed multiplication of integers;
                                # no overflow detection
                                # REG[hi] <-- (GPR[rs] * GPR[rt])(63:32)
                                # REG[lo] <-- (GPR[rs] * GPR[rt])(31:0)

nop                             # no operation
                                # executed as:  sll  $zero, $zero, 0

nor     rd, rs, rt              # bitwise logical NOR
                                # GPR[rd] <-- !(GPR[rs] OR GPR[rt])

sll     rd, rt, sa              # logical shift left a fixed number of bits
                                # GPR[rd] <-- GPR[rs] <<l sa

slt     rd, rs, rt              # set register to result of comparison
                                # GPR[rd] <-- (GPR[rs] < GPR[rt] ? 0 : 1)

slti    rt, rs, imm16           # set register to result of comparison
                                # GPR[rt] <-- (GPR[rs] < imm16 ? 0 : 1)

sra     rd, rt, sa              # arithmetic shift right a fixed number of bits
                                # GPR[rd] <-- GPR[rt] >>a sa
```

```
srav    rd, rt, rs              # arithmetic shift right a variable number of bits
                                # GPR[rd] <-- GPR[rt] >>a GPR[rs]

sub     rd, rs, rt              # signed subtraction of integers
                                # GPR[rd] <-- GPR[rs] - GPR[rt]
```

Your assembler must support the following basic control-of-flow instructions:

```
beq     rs, rt, offset          # conditional branch if rs == rt
                                # PC <-- (GPR[rs] == GPR[rt] ? PC + 4 + offset <<1 2)
                                #                            : PC + 4)

blez    rs, offset              # conditional branch if rs <= 0
                                # PC <-- (GPR[rs] <= 0 ? PC + 4 + offset <<1 2)
                                #                      : PC + 4)

bgtz    rs, offset              # conditional branch if rs > 0
                                # PC <-- (GPR[rs] > 0 ? PC + 4 + offset <<1 2)
                                #                     : PC + 4)

bne     rs, rt, offset          # conditional branch if rs != rt
                                # PC <-- (GPR[rs] != GPR[rt] ? PC + 4 + offset <<1 2)
                                #                            : PC + 4)

j       target                  # unconditional branch
                                # PC <-- ( (PC+4)(31:28) || (target <<1 2))

syscall                         # invoke exception handler, which examines $v0
                                # to determine appropriate action; if it returns,
                                # returns to the succeeding instruction; see the
                                # MIPS32 Instruction Reference for format
```

Your assembler must support the following pseudo-instructions:

```
move    rd, rs                  # copy contents of GPR[rs] to GPR[rt]
                                # GPR[rd] = GPR[rs]
                                # pseudo-translation:
                                #   addu  rd, zero, rs


blt     rs, rt, offset          # conditional branch if rs < rt
                                # PC <-- (GPR[rs] < GPR[rt] ? PC + 4 + offset <<1 2)
                                #                           : PC + 4)
                                # pseudo-translation:
                                #   slt   at, rs, rt
                                #   bne   at, zero, offset

la      rt, label               # load address label to register
                                # GPR[rd] <-- label
                                # pseudo-translation for 16-bit label:
                                #   addi  rt, $zero, label

li      rt, imm16               # load 16-bit immediate to register
                                # GPR[rd] <-- imm
                                # pseudo-translation:
                                #   addiu  rt, $zero, imm16

lw      rt, label               # load word at address label to register
                                # GPR[rd] <-- Mem[label]
                                # pseudo-translation:
                                #   lw   rt, label[15:0]($zero)
```

MIPS32 assembly format constraints:

The assembly programs will satisfy the following constraints:

- Labels will begin in the first column of a line, and will be no more than 32 characters long. Labels are restricted to alphanumeric characters and underscores, and are always followed immediately by a colon character (`':'`).
- Labels in the **`.text`** segment will always be on a line by themselves.
- Labels in the **`.data`** segment will always occur on the same line as the specification of the variable being defined.
- Labels are case-sensitive; that actually makes your task a bit simpler.
- MIPS instructions do not begin in a fixed column; they are preceded by an arbitrary amount of whitespace (possibly none).
- Blank lines may occur anywhere; a blank line will always contain only a newline character.
- Whitespace will consist of spaces, tab characters, or a mixture of the two. Your parsing logic must handle that.
- Registers will be referred to by symbolic names (`$zero`, `$t5`) rather than by register number.
- Instruction mnemonics and register names will use lower-case characters.
- Assembly source files will always be in UNIX format.

You must be sure to test your implementation with all the posted test files; that way you should avoid any unfortunate surprises when we test your implementation.

Input

The input files will be MIPS assembly programs in ASCII text. The assembly programs will be syntactically correct, compatible with the MIPS32 Instruction Set manual, and restricted to the subset of the MIPS32 instruction set defined above. Example programs will be available from the course website.

Each line in the input assembly file will either contain an assembly instruction, a section header directive (such as `.data`) or a label (a jump or branch target). The maximum length of a line is 256 bytes.

Your input file may also contain comments. Any text after a `'#'` symbol is a comment and should be discarded by your assembler. Section header directives, such as **`.data`** and **`.text`** will be in a line by themselves. Similarly, labels (such as `loop:`) will be on a line by themselves. The input assembly file will contain one data section, followed by one text section.

Your assembler can be invoked in either of the following ways:

```
assemble <input file> <output file>
assemble <input file> <output file> -symbols
```

The specified input file must already exist; if not, your program should exit gracefully with an error message to the console window. The specified output file may or may not already exist; if it does exist, the contents should be overwritten.

Output

Output when invoked as: `assemble <input file> <output file>`

Your assembler will resolve all references to branch targets in the .text section and variables in the **`.data`** section and convert the instructions in the **`.text`** section into machine code.

To convert an instruction into machine code follow the instruction format rules specified in the class textbook. For each format (R-format, I-format or J-format), you should determine the opcode that corresponds to instruction, the values for the register fields and any optional fields such as the function code and shift amount fields for arithmetic instructions (R-format) and immediate values for I-format instructions.

The output machine code should be saved to the output file specified in the command line. The output file should contain the machine code corresponding to instructions from the `.text` section followed by a blank line followed by variables from the `.data` section in human readable binary format (0s and 1s). For example, to represent the decimal number `40` in 16-bit binary you would write `0000000000101000`, and to represent the decimal number -40 in 16-bit binary you would write `1111111111011000`.

The output file is a text file, not a binary file; that's a concession to the need to evaluate your results.

Your output file should match the machine code file posted with the grading harness. A sample showing the assembler's translation of the `adder.asm` program is given at the end of this specification.

Output when invoked as: `assemble <input file> <output file> –symbols`

Your assembler will write (to the specified output file) a well-formatted table, listing every symbolic name used in the MIPS32 assembly code and the address that corresponds to that label. Addresses will be written in hex. Note: when invoked this way, your assembler will not write any other output.

We will make the following assumptions about addresses and program segments:

- The base address for the text segment is 0x00000000, so that's the address of the first machine instruction.
- The base address of the data segment is 0x00002000, so that's the address of the first thing declared in the data segment.

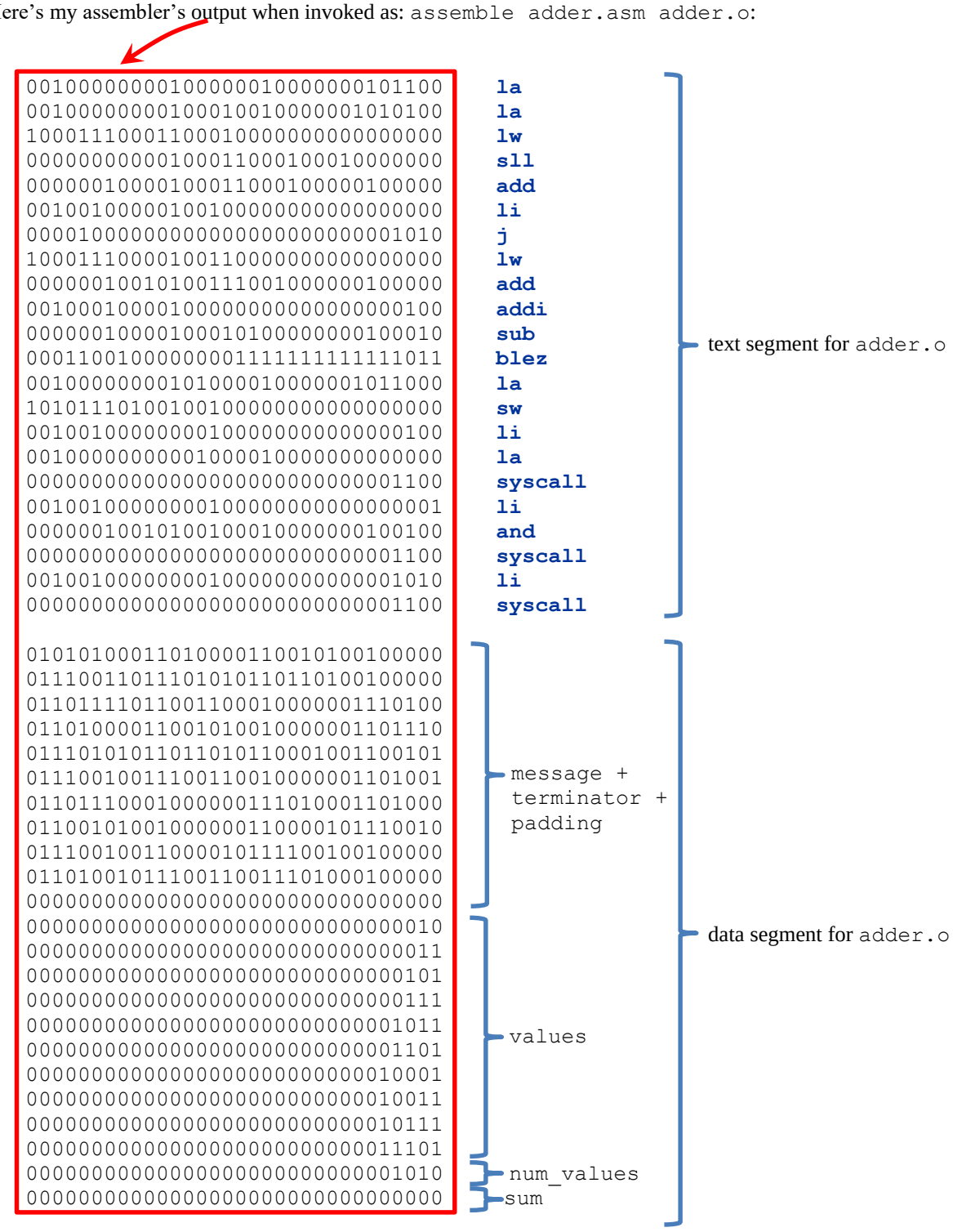
The second fact above implies that the text segment cannot be longer than 8 KiB or 2048 machine instructions. You don't need to do anything special about that fact.

Sample Assembler Input

```
# adder.asm
#
# The following program computes the sum of all elements in an array.
# The program then prints the sum of all the entries in the array.

.data
message:    .asciiz "The sum of the numbers in the array is: "
values:     .word   2, 3, 5, 7, 11, 13, 17, 19, 23, 29  # array of 10 words
num_values: .word   10           # size of array
sum:        .word   0            # running total

.text
main:
        la   $s0, values         # load address of the array;
                                 # $s0 will point to the current element
        la   $s1, num_values     # load address of num_values
        lw   $s1, 0($s1)         # load num_values
        sll  $s1, $s1, 2         # compute size of array in bytes
        add  $s1, $s0, $s1       # compute one-past-end address
        li   $s2, 0              # set running total to 0

        j    chk                 # check for empty array
loop:
        lw   $s3, 0($s0)         # fetch current data element
        add  $s2, $s2, $s3       # update running total
        addi $s0, $s0, 4         # step pointer to next array element
chk:
        sub  $t0, $s0, $s1       # compute distance beyond one-past-end
        blez $t0, loop           # if ( distance <= 0 ) then loop

        # done processing array elements; store total to memory
        la   $s4, sum            # get address for sum
        sw   $s2, 0($s4)         # write sum to memory

        # report results to user
        li   $v0, 4              # system call to print string
        la   $a0, message        # load address of message into arg register
        syscall                  # make call

        li   $v0, 1              # system call to print an integer
        and  $a0, $s2, $s2       # load value to print into arg register
        syscall                  # make call

        li   $v0, 10             # system call to terminate program
        syscall                  # make call
```

Sample Assembler Output

Here's my assembler's output when invoked as: `assemble adder.asm adder.o`:



Data segment notes:

The character string variable `message` is stored as a sequence of one-byte ASCII codes, with the characters in ascending order by address.

The next value in the data segment is a 32-byte integer, which must be aligned on an address that is a multiple of 4; therefore, we must add "padding" NULL bytes before storing the bytes of the integer. The integer variable `values` is stored as a sequence of 32-bit values, which are the 2's complement representations of the values shown in the assembly code. In the data segment display above, the first value, 2, is displayed in big-endian byte order:

```
                00000000 00000000 00000000 00000010
          Low address                      High address
```

The value of `num_values` is 10, which is expressed in hex as `0x0000000A`.

```
                00000000 00000000 00000000 00001010
```

You must be sure that you write the bytes of integers in big-endian order as well.

Handling of strings in the data segment:

For example, suppose we have the data segment:

```
.data
S1:    .asciiz    "abc"
I1:    .word      1
S4:    .asciiz    "CDEFG"
I4:    .word      4
```

The corresponding data segment representation should be:

```
01100001011000100110001100000000     S1:   'a'   'b'   'c'   '\0'
00000000000000000000000000000001     I1:   0x00000001
01000011010001000100010101000110     S2:   'C'   'D'   'E'   'F'
01000111000000000000000000000000           'G'   '\0'  padding bytes
00000000000000000000000000000100     I2:   0x00000004
```

The ASCII codes for the characters in a string are stored in front-to-back order (the opposite of the usual MIPS convention). Since MIPS requires a 32-bit integer to be aligned to an address that is a multiple of 4, we must insert padding bytes after the string variable `S2`. We will place those padding bytes as shown above, after the last byte of the string variable (`'\0'`).

Output for the `-symbols` option:

Here's my assembler's output when invoked as: `assemble adder.asm adder.sym -symbols`:

```
0x00002000   message
0x0000202C   values
0x00002054   num_values
0x00002058   sum
0x00000000   main
0x0000001C   loop
0x00000028   chk
```

The order in which you display the symbols is up to you. My implementation writes them in the order they occur in the source file, but that is not a requirement.

How can I verify my output or test my code manually?

Download the posted test/grading tar file and unpack it. This will provide you with a collection of MIPS assembly (`.asm`) files and corresponding assembled object files (`.o`), placed in a subdirectory `testData`. For example, if you've compiled your assembler, you could run it on the first test case by using the command:

```
assemble ./testData/test01.asm myobj01.o
```

You can then compare your assembler's output to the reference output by using the supplied `compare` utility:

```
compare 1 ./testData/test01.o myobj01.o
```

This will generate detailed output showing any mismatches, and a score. We will use exactly this approach in evaluating the correctness of your submission. The first parameter, `1` in the example above, is just a dummy one that has no significance until it is used by the grading script.

Note: the test harness described later provides the actual test data, and reference solutions, that we will use to grade your solutions (milestones and final). You should make good use of that in your own testing. However, the grading script we've supplied is a rather blunt instrument, and it not really suitable for testing individual cases. You cannot use gdb or Valgrind with the grading harness. So, you should use the instructions above if you're trying to diagnose why you are failing on a particular test case.

Milestones

In order to assess your progress, there will be two milestones for the project. Each of these will require that you submit a partial solution that achieves specified functionality. Each milestone will be evaluated by using a scripted testing environment, which will be posted on the course website at least two weeks before the corresponding milestone is due. We will not perform any Valgrind-based evaluation of the milestones, although the test harness may include relevant output.

Your scores on the milestones will constitute 6% and 9%, respectively, of your final score on the project.

Milestone 1

The first milestone will be due approximately two weeks before the final project deadline. Your submission must support translation of a MIPS assembly program that consists of a `.text` segment including the following instructions:

```
add     rd, rs, rt              # signed addition of integers; overflow detection
                                # GPR[rd] <-- GPR[rs] + GPR[rt]

addi    rt, rs, imm16           # signed addition with 16-bit immediate;
                                # overflow detection
                                # GPR[rt] <-- GPR[rs] + imm16

nor     rd, rs, rt              # bitwise logical NOR
                                # GPR[rd] <-- !(GPR[rs] OR GPR[rt])

slti    rt, rs, imm16           # set register to result of comparison
                                # GPR[rt] <-- (GPR[rs] < imm16 ? 0 : 1)

syscall                         # invoke exception handler, which examines $v0
                                # to determine appropriate action; if it returns,
                                # returns to the succeeding instruction; see the
                                # MIPS32 Instruction Reference for format
```

Your submission for milestone 1 must support references to the `$s*` and `$v0` registers. The test files for this milestone will not include a `.data` segment, and there will be no symbolic labels in the `.text` segment.

Milestone 2

The second milestone will be due approximately one week before the final submission. Your submission for this milestone must support translation of a MIPS assembly program that includes both a `.data` and a `.text` segment. The `.data` segment may include:

`.word` This is used to hold one or more 32 bit quantities, initialized with given values. For example:

```
    var1:    .word 15       # creates one 32 bit integer called var1 and initializes
                            #   it to 15.

    array2:  .word 2:10     # creates an array of 10 32 bit integers and initializes
                            #   all the elements of the array to the value 2.
```

The `.text` segment may include any of the instructions from the first milestone, and also:

```
lw      rt, offset(rs)          # Transfers a word from memory to a register.
                                # GPR[rt] <-- Mem[GPR[rs] + offset]

la      rd, label               # load address label to register
                                # GPR[rd] <-- label
                                # pseudo-translation for 16-bit label:
                                #   addi  rd, $zero, label
```

```
mul     rd, rs, rt              # signed multiplication of integers;
                                # no overflow detection
                                # GPR[rd] <-- (GPR[rs] * GPR[rt])(31:0)

mult    rs, rt                  # signed multiplication of integers;
                                # no overflow detection
                                # REG[hi] <-- (GPR[rs] * GPR[rt])(63:32)
                                # REG[lo] <-- (GPR[rs] * GPR[rt])(31:0)

nor     rd, rs, rt              # bitwise logical NOR
                                # GPR[rd] <-- !(GPR[rs] OR GPR[rt])

beq     rs, rt, offset          # conditional branch if rs == rt
                                # PC <-- (GPR[rs] == GPR[rt] ? PC + 4 + offset <<1 2)
                                #                            : PC + 4)

bne     rs, rt, offset          # conditional branch if rs != rt
                                # PC <-- (GPR[rs] != GPR[rt] ? PC + 4 + offset <<1 2)
                                #                            : PC + 4)
```

Your submission for milestone 2 must deal with all requirements for milestone 1, support the `$s*`, `$v0` and `$zero` registers, and symbolic labels in both the **`.text`** and **`.data`** segments.

NO LATE SUBMISSIONS WILL BE GRADED FOR EITHER MILESTONE!

What should I turn in, and how?

For both the milestone and the final submissions, create an uncompressed tar file containing:

- All the `.c` and `.h` files which are necessary in order to build your assembler.
- A GNU makefile named "`makefile`". The command "`make assembler`" should build an executable named "`assemble`". The makefile may include additional targets as you see fit.
- A `readme.txt` file if there's anything you want to tell us regarding your implementation. For example, if there are certain things that would cause your assembler to fail (e.g., it doesn't handle **`la`** instructions), telling us that may result in a more satisfactory evaluation of your assembler.
- A `pledge.txt` file containing the pledge statement from the course website.
- Nothing else. Do not include object files or an executable. We will compile your source code.

Submit this tar file to the Curator, by the deadline specified on the course website. Late submissions of the final project will be penalized at a rate of 10% per day until the final submission deadline.

A *flat tar file* is one that includes no directory structure. In this case, you can be sure you've got it right by performing a very simple exercise. Unpack the posted test harness, and follow the instructions in the `readme.txt` file. If the two shell scripts fail to build an executable, and test it, then there's something wrong with your tar file (or your C code).

You can also tell the difference by simply doing a table-of-contents listing of your tar file (use the switches `-tf`). The listing of a flat tar file will not show any path information. The one on the left below is not flat; the one on the right is flat:

```
Linux > tar tf C3Test.tar
C3TestFiles/
C3TestFiles/mref01.asm
C3TestFiles/mref01.o
C3TestFiles/mref02.asm
C3TestFiles/mref02.o
C3TestFiles/mref03.asm
C3TestFiles/mref03.o
compare
prepC3.sh
testC3.sh
```

```
Linux > tar tf C3Source.tar
assembler.c
Instruction.c
IWrapper.c
MIParser.c
ParseResult.c
Registers.c
SymbolTable.c
Instruction.h
MIParser.h
ParseResult.h
Registers.h
SymbolTable.h
SystemConstants.h
makefile
pledge.txt
```

Some General Coding Requirements

Your solution will be compiled by a test/grading harness that will be supplied along with this specification.

You are required* to implement your solution in logically-cohesive modules (paired `.h` and `.c` files), where each module encapsulates the code and data necessary to perform one logically-necessary task. For example, a module might encapsulate the task of mapping register numbers to symbolic names, or the task of mapping mnemonics to opcodes, etc. There are many reasonable ways to organize the code for a system as large as this; my solution employs about a dozen modules, and involves more than 2300 lines of C code.

The TAs are instructed that they are not required to provide help with solutions that are not properly organized into different files, or with solutions that are not adequately commented.

We will require* your solution to achieve a "clean" run on `valgrind`. A clean run should report the same number of allocations and frees, that zero heap bytes were in use when the program terminated, that there were no invalid reads or writes, and that there were no issues with uninitialized data. We will not be concerned about suppressed errors reported by `valgrind`. See the discussion of `valgrind` below, and the introduction to `valgrind` that's posted on the course website. The discussion of `valgrind` later in this specification shows the report for my solution.

In line with the requirement above, we require that you make sensible use of dynamic memory allocation in your solution. Just how and where you allocate objects dynamically is up to you, but failure to use dynamic allocation sufficiently will result in a deduction, as will excessive memory usage.

Finally, this is not a requirement, but you are strongly advised to use `calloc()` when you allocate dynamically, rather than `malloc()`. This will guarantee your dynamically-allocated memory is zeroed when it's allocated, and that may help prevent certain errors.

* "Required" here means that this will be scored by a human being after your solution has been autograded. The automated evaluation will certainly not adjust your score for these things. Failure to satisfy these requirements will result in deductions from your autograding score; the potential size of those deductions may not be specified in advance (but you will not be happy with them).

Grading

The evaluation of your solution will be based on its ability to correctly translate programs using the specified MIPS32 assembly subset to MIPS32 machine code, your proper use of dynamic memory allocation, and a somewhat crude evaluation of the modularity of your implementation. That is somewhat unfortunate, since there are many other issues we would like to consider, such as the quality of your design, your internal documentation, and so forth. However, we do not have sufficient staff to consider those things fairly, and therefore we will not consider them at all.

We have released a tar file containing a testing harness (test shell scripts and test cases). You can use the contents of this tar file to evaluate your milestone submissions and your final submission, in advance, in precisely the same way we will evaluate it. We are posting the test harness as an aid in your testing, but also so that you can verify that you are packaging your submission according to the requirements given above. Submissions that do not meet the requirements typically receive extremely low scores.

The posted test harness contains the following files:

`readme.txt`	instructions for running the grading script
`gradeC03.sh`	grading script; see the readme for instructions
`compare`	tool used by the grading script to check the .o files produced by your assembler
`symcompare`	tool used by the grading script to check the symbol tables produced by your assembler
`milestone1/`	assembly code files and reference .o files for testing your milestone 1 solution
`  m1testxx.asm`	
`  m1testxx.o`	
`milestone2/`	assembly code files and reference .o files for testing your milestone 1 solution
`  m2testxx.asm`	
`  m2testxx.o`	
`final/`	assembly code files and reference .o files for testing your final solution
`  ftestxx.asm`	
`  ftestxx.o`	
`symtabs/`	reference symbol tables for checking your symbol table output
`  refsymsxx.txt`	

Our testing of your milestone and final assembler submissions will be performed using the test harness files we will make available to you. We expect you to use each test harness to validate your solution.

Valgrind will be employed, on one test case, to check how well your solution manages memory:

- A penalty of up to 10% will be assessed if Valgrind shows any memory leaks, invalid reads, invalid writes, or uses of uninitialized data. The exact penalty will be proportional to the number of bytes leaked, number of invalid accesses, and number of uses of uninitialized data.
- A penalty of up to 5% will be assessed if Valgrind shows that your solution performed excessive dynamic allocations (even if all are properly freed). The reason for this is that some students will attempt to sidestep memory-related issues by simply enlarging the arrays they allocate; that is unprofessional and unacceptable. The term "excessive" is deliberately vague, but the likely interpretation would be that "excessive" is a fair judgment if your solution uses 5-10 times as much memory as the reference solution, on the same test.

We will also consider the modularity of your implementation. Again, we will not give precise criteria for this. The reference solution consists of 13 modules; we do not expect your solution to match that, but having fewer than 3-5 modules will likely draw a penalty of up to 5%.

We will not offer any accommodations for submissions that do not work properly with the corresponding supplied test harness.

Test Environment

Your assembler will be tested on the rlogin cluster or the equivalent, running 64-bit CentOS 7 and gcc 4.8. There are many of you, and few of us. Therefore, we will not test your assembler on any other environment. So, be sure that you compile and test it there before you submit it. Be warned in particular, if you use OS X, that the version of gcc available there has been modified for Apple-specific reasons, and that past students have encountered significant differences between that version and the one running on Linux systems.

Maximizing Your Results

Ideally you will produce a fully complete and correct solution. If not, there are some things you can do that are likely to improve your score:

- Make sure your assembler submission works properly with the posted test harness (described above). If it does not, we will almost certainly not be able to evaluate your submission and you are likely to receive a score of 0.
- Make sure your assembler does not crash on any valid input, even if it cannot produce the correct results. If you ensure that your assembler processes all the posted test files, it is extremely unlikely it will encounter anything in our test data that would cause it to crash. On the other hand, if your assembler does crash on any of the posted test files, it will certainly do so during our testing. We will not invest time or effort in diagnosing the cause of such a crash during our testing. It's your responsibility to make sure we don't encounter such crashes.
- If there is a MIPS32 instruction or data declaration that your solution cannot handle, document that in the `readme.txt` file you will include in your submission.
- If there is a MIPS32 instruction or data declaration that your solution cannot handle, make sure that it still produces the correct number of lines of output, since we will automate much of the checking we do. In particular, if your assembler encounters a MIPS32 instruction it cannot handle, write a sequence of 32 asterisk characters (`'*'`) in place of the correct machine representation (or multiple lines for some pseudo-instructions). Doing this will not give you credit for correctly translating that instruction, but this will make it more likely that we correctly evaluate the following parts of your translation.

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include a file, named `pledge.txt`, containing the following pledge statement in the submitted tar file:

```
//  On my honor:
//
//  - I have not discussed the C language code in my program with
//    anyone other than my instructor or the teaching assistants
//    assigned to this course.
//
//  - I have not used C language code obtained from another student,
//    the Internet, or any other unauthorized source, either modified
//    or unmodified.
//
//  - If any C language code or documentation used in my program
//    was obtained from an authorized source, such as a text book or
//    course notes, that has been clearly noted with a proper citation
//    in the comments of my program.
//
//  - I have not designed this program in such a way as to defeat or
//    interfere with the normal operation of the grading code.
//
//    <Student Name>
//    <Student's VT email PID>
```

We reserve the option of assigning a score of zero to any submission that is undocumented or does not contain this statement.

Advice

The following observations are purely advisory, but are based on my experience, including that of implementing a solution to this assignment. These are advice, not requirements.

First, and most basic, analyze what your assembler must do and design a sensible, logical framework for making those things happen. There are fundamental decisions you must make early in the process of development. For example, you could represent the machine instructions in a number of ways as you build them. They can be represented as arrays of individual bits (which could be integers or characters), or they can be represented in binary format, which would be the expected format for a "real" assembler's final output. I am not convinced that either of those approaches is inherently better, or that there are not reasonable alternatives. But, this decision has ramifications that will propagate throughout your implementation.

It helps to consider how you would carry out the translation from assembly code to machine instructions by hand. If you do not understand that, you are trying to write a program that will do something you do not understand, and your chances of success are reduced to sheer dumb luck.

Second, and also basic, practice incremental development! This is a sizeable program, especially so if it's done properly. My solution, including comments, runs something over 2300 lines of code. It takes quite a bit of work before you have enough working code to test on full input files, but unit testing is extremely valuable.

Record your design decisions in some way; a simple text file is often useful for tracking your deliberations, the alternatives you considered, and the conclusions you reached. That information is invaluable as your implementation becomes more complete, and hence more complex, and you are attempting to extend it to incorporate additional features.

Write useful comments in your code, as you go. Leave notes to yourself about things that still need to be done, or that you are currently handling in a clumsy manner.

A preprocessing phase is helpful; for example, it gives you a chance to filter out comments, trim whitespace, and gather various pieces of information. Do not try to do everything in one pass. Compilers and assemblers frequently produce a number of intermediate files and/or in-memory structures, recording the results of different phases of execution.

Consider how you would carry out the translation of a MIPS32 assembly program to machine code if you were doing it manually. If you don't understand how to do it by hand, you cannot write a program to do it!

Take advantage of tools. You should already have a working knowledge of gdb. Use it! The debugger is invaluable when pinning down the location of segfaults; but it is also useful for tracking down lesser issues if you make good use of breakpoints and watchpoints. Some memory-related errors yield mysterious behavior, and confusing runtime error reports. That's especially true when you have written past the end of a dynamically-allocated array and corrupted the heap. This sort of error can often be diagnosed by using valgrind.

Enumerated types are extremely useful for representing various kinds of information, especially about type attributes of structured variables. For example, if implementing a GIS system, we might find the following type useful:

```
// GData.h
enum _FeatureType {CITY, RIVER, MOUNTAIN, BUILDING, . . . , ISLAND};
typedef enum _FeatureType FeatureType;
...
struct _GData {
   char* Name;
   char* State;
   ...
   FeatureType FType;
   uint16_t Elevation;
};
typedef struct _GData GData;
...
```

Think carefully about what information would be useful when analyzing and translating the assembly code. Much of this is actually not part of the source code, but rather part of the specification of the assembly and machine languages. Consider using static tables of structures to organize language information; by static, I mean a table that's directly initialized when it's declared, has static storage duration, and is private to the file in which it's created. For example:

```
// GData.c
#define NUMRECORDS 50

static GData GISTable[NUMRECORDS] = {
   {"New York", "NY", ..., CITY, 33},
   {"Pikes Peak", "CO", ..., MOUNTAIN, 14115},
   ...
   {"McBryde Hall", "VA", ..., BUILDING, 2080}
};
```

Obviously, the sample code shown above does not play a role in my solution. On the other hand, I used this approach to organize quite a bit of information about instruction formats and encodings. It's useful to consider the difference between the inherent attributes of an instruction, like its opcode, and situational attributes that apply to a particular occurrence of an instruction, like the particular registers it uses. Inherent attributes are good things to keep track of in a table. Situational attributes must be dealt with on a case-by-case basis.

Also, be careful about making assumptions about the instruction formats… Consult the manual *MIPS32 Architecture Volume 2*, linked from the Resources page. It has lots of details on machine language and assembly instruction formats. I found it invaluable, especially in some cases where an instruction doesn't quite fit the simple description of MIPS assembly conventions in the course notes (e.g., `sll` and `syscall`).

Feel free to make reasonable assumptions about limits on things like the number of variables, number of labels, number of assembly statements, etc. It's not good to guess too low about these things, but making sensible guesses let you avoid (some) dynamic allocations.

Write lots of "utility" functions because they simplify things tremendously; e.g., string trimmers, mappers, etc.

Data structures play a role because there's a substantial amount of information that must be collected, represented and organized. However, I used nothing fancier than arrays.

Data types, like the structure shown above, play a major role in a good solution. I wrote a significant number of them.

Explore `string.h` carefully. Useful functions include `strncpy()`, `strncmp()`, `memcpy()` and `strtok()`. There are lots of useful functions in the C Standard Library, not just in `string.h`. One key to becoming proficient and productive in C, as in most programming languages, is to take full advantage of the library that comes with that language.

When testing, you should create some small input files. That makes it easy to isolate the various things your assembler must deal with. Note that the assembler is not doing any validation of the logic of the assembly code, so you don't have to worry about producing assembly test code that will actually do anything sensible. For example, you might use a short sequence of R-type instructions:

```
.text
        add   $t0, $t1, $t2
        sub   $t3, $t1, $t0
        xor   $s7, $t4, $v0
```

Appendix Valgrind

Valgrind is a tool for detecting certain memory-related errors, including out of bounds accessed to dynamically-allocated arrays and memory leaks (failure to deallocate memory that was allocated dynamically). A short introduction to Valgrind is posted on the Resources page, and an extensive manual is available at the Valgrind project site (www.valgrind.org).

For best results, you should compile your C program with a debugging switch (`-g` or `-ggdb3`); this allows Valgrind to provide more precise information about the sources of errors it detects. For example, I ran my solution for a related project, with one of the test cases, on Valgrind:

```
[wdm@centosvm C3]$ valgrind --leak-check=full --show-leak-kinds=all --log-file=vlog.txt
--track-origins=yes -v assemble ftest10.asm ftest10.o
==12926== Memcheck, a memory error detector
==12926== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12926== Using Valgrind-3.14.0-353a3587bb-20181007X and LibVEX; rerun with -h for copyright info
==12926== Command: assemble ftest10b.asm ftest10.o
==12926== Parent PID: 12925
==12926==
==12926==
==12926== HEAP SUMMARY:
==12926==     in use at exit: 0 bytes in 0 blocks
==12926==   total heap usage: 140 allocs, 140 frees, 38,016 bytes allocated
==12926==
==12926== All heap blocks were freed -- no leaks are possible
==12926==
==12926== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==12926== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

And, I got good news... there were no detected memory-related issues with my code. On the other hand, I ran Valgrind on a mildly incomplete solution to the same project, and the results were less satisfactory:

```
==9321== HEAP SUMMARY:
==9321==     in use at exit: 12 bytes in 3 blocks
==9321==   total heap usage: 248 allocs, 245 frees, 57,666 bytes allocated
==9321==
==9321== Searching for pointers to 3 not-freed blocks
==9321== Checked 70,024 bytes
==9321==
==9321== 4 bytes in 1 blocks are definitely lost in loss record 1 of 2
==9321==    at 0x4C2BFB9: calloc (vg_replace_malloc.c:762)
==9321==    by 0x404F0A: copyString (Util.c:47)
==9321==    by 0x401C60: Instruction_Init (Instruction.c:21)
==9321==    by 0x402626: ParseTextSegment (Parser.c:163)
==9321==    by 0x400FF8: main (assemble.c:88)
==9321==
==9321== 8 bytes in 2 blocks are definitely lost in loss record 2 of 2
==9321==    at 0x4C2BFB9: calloc (vg_replace_malloc.c:762)
==9321==    by 0x404F0A: copyString (Util.c:47)
==9321==    by 0x401C39: Instruction_Init (Instruction.c:19)
==9321==    by 0x402626: ParseTextSegment (Parser.c:163)
==9321==    by 0x400FF8: main (assemble.c:88)
==9321==
==9321== LEAK SUMMARY:
==9321==    definitely lost: 12 bytes in 3 blocks
==9321==    indirectly lost: 0 bytes in 0 blocks
==9321==      possibly lost: 0 bytes in 0 blocks
==9321==    still reachable: 0 bytes in 0 blocks
==9321==         suppressed: 0 bytes in 0 blocks
==9321==
==9321== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
==9321== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

It's worth noting that Valgrind not only detects the occurrence of memory leaks, but it also pinpoints where the leaked memory was allocated in the code. That makes it much easier to track down the logical errors that lead to the leaks.

Valgrind can also detect computations that use uninitialized variables, and invalid reads/writes (to memory locations that lie outside the user's requested dynamic allocation).

Do note that Valgrind has its limitations. False positives are uncommon, as are false negatives, but both are possible. More importantly, Valgrind depends on using its own internal memory allocator in order to detect memory leaks and out-of-bounds memory accesses. Valgrind is not particularly good at detecting errors related to statically-allocated memory.

Credits

The original formulation of this project was created by Dr Srinidhi Vadarajan, who was then a member of the Dept of Computer Science at Virginia Tech. His sources of inspiration for this project are lost in the mists of time.

The current modification was produced by William D McQuain, as a member of the Dept of Computer Science at Virginia Tech. Any errors, ambiguities, and omissions should be attributed to him.

Change Log Relative to Version 9.00

If changes are made to the specification, details will be noted below, the version number will be updated, and an announcement will be made on the course Forum board.

Version	Posted	Pg	Change
9.00	Mar 21		Base document.
9.01	April 20	8	Fixed placement of "-symbols" switch.