C Programming Disassembling MIPS32 Machine Programs

For this assignment, you will implement a C program that produces an assembly-language representation of a MIPS32 machine code program.

The input file will contain a valid MIPS32 machine code program, including both a .text segment and a .data segment, written in binary text format, with one word per line. Here's an example input file:

```
10001100000100000010000000000000
10001100000100010010000000000100
10001100000100100010000000001000
10001100000100110010000000001100
00000010001100001000100000100000
00000010010100101001000000100000
00010010010100110000000000000001
00010110001100101111111111111100
00100000000000100000000000001010
00000000000000000000000000001100

00000000000000000000000000000001
00000000000000000000000000000100
00000000000000000000000000010000
00000000000000000000000001000000
```

Each word consists of exactly 32 bits, represented by the characters '0' and '1', and is followed immediately by a Linux line terminator. The number of instructions in the text segment will vary, and so will the number of words in the data segment, and your solution must process all of them and halt correctly at the end of the input file. The text segment will be given first, followed by a single empty line, and then by the data segment.

For the purpose of this assignment, we will assume the text segment always begins at address 0x00000000, and the data segment always begins at address 0x00002000. These addresses are needed in order to determine the relationships between labels and immediates and the addresses of instructions and variables.

The machine program shown above should be translated to the following assembly-language representation:

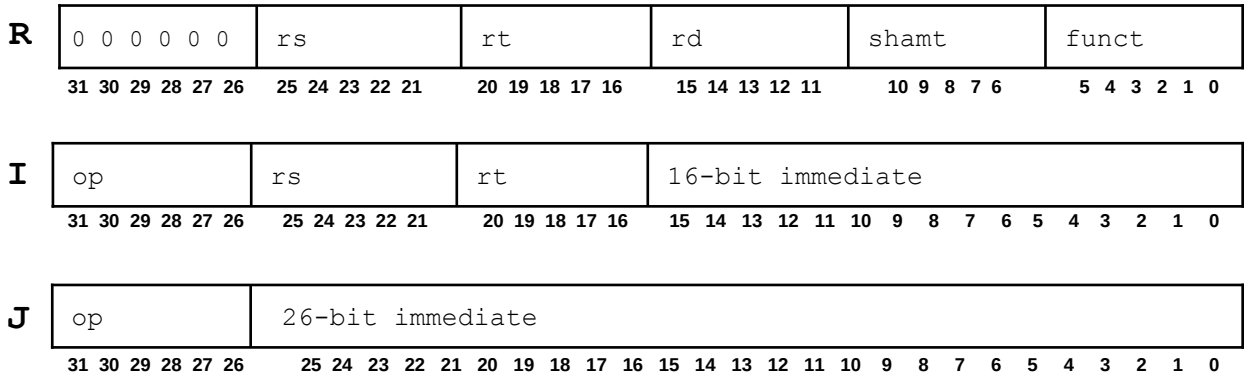
```
.text
main:   lw          $s0,    V01
        lw          $s1,    V02
        lw          $s2,    V03
        lw          $s3,    V04
L02:    add         $s1,    $s1,    $s0
        add         $s2,    $s2,    $s2
        beq         $s2,    $s3,    L01
        bne         $s1,    $s2,    L02
L01:    addi      $zero,    $v0,    10
        syscall

.data
V01:    .word   1
V02:    .word   4
V03:    .word   16
V04:    .word   64
```

The text segment must be shown first, followed by a single empty line, and concluding with the data segment. The instructions should be formatted as neatly as possible, with sensible indentation and separation of parameters. Code segment labels must be on the same line as the corresponding instruction; data segment labels must be on the same line as the variable initialization code.

Translating from ML to Assembly Code

There are three different formats for MIPS32 machine instructions (ignoring a small number of special cases):



Consider the following machine instruction:

```
00100000000010000010000000000000
```

The opcode field is `001000`, which corresponds to the `addi` instruction. The `addi` instruction takes three parameters: two registers and an immediate value*. The rs and rt fields are `00000` and `01000`, which correspond to `$zero` and `$t0`, respectively. The immediate field is `0010000000000000`, which is a signed value for `addi`, and corresponds to 8192. Finally, the first register is used to as an input operand, and the second is the destination for the result. So, the machine instruction would be represented in assembly language as:

```
addi   $t0, $zero, 8192
```

As another example:

```
00000011000010110100110110000010
```

The opcode field is `000000`, which means this is an R-format instruction; the funct field is `000010`, which corresponds to `srl`. An R-format instruction always takes three registers as parameters, except when it does not. In this case, we have a shift instruction, which takes two registers and a 5-bit immediate as parameters. The rs field, `11000`, does not specify a register number at all. The rt field, `01011`, corresponds to `$t3`, and is the input operand. The rd field, `01001`, corresponds to `$t1`, and is the destination for the result. Finally, the shamt field is `10110`, which represents 22. So, the machine instruction would be represented in assembly language as:

```
srl    $t3, $t1, 22
```

So, how do we know all this? Reading. The course notes and relevant discussions in P&H reveal much... but the MIPS32 Architecture for Programmers Volume II (see the Resources page) provides full details regarding the instruction set.

The `opcode` (together with the `funct` field for some instructions) gives you enough information to determine exactly how to handle the interpretation of the instruction, provided you've built static lookup tables that store sufficient information about the instructions.

Your solution must correctly deal with any register or shift field from `00000` to `11111` (i.e., all 32 general-purpose MIPS registers). Note that both are interpreted as unsigned integers. Your solution must correctly deal with any immediate field from `0000000000000000` to `1111111111111111`, whether the instruction calls for the immediate to be interpreted as a signed or unsigned value. It might be helpful to recall that signed integers are represented in 2's complement form.

Required MIPS Machine Instruction Features

The subset of the MIPS machine language that you need to implement is defined below; the instructions are described as assembly instructions because that provides you with some helpful information about how each instruction is formed. 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`.
- `rd`, `rs` and `rt` refer to bits 15:11, 25:21, and 20:16, respectively, of the relevant machine instruction
- `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 (bits 10:6); 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: $>>_a$, $<<_l$ and $>>_l$
- The C ternary operator is used for selection: `condition ? if-true : if-false`
- Concatenation of bit-sequences is denoted by `||`.

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

```
add   rd, rs, rt              # Signed addition of integers
                              # GPR[rd] <-- GPR[rs] + GPR[rt]

addi  rt, rs, imm16           # Signed addition with 16-bit immediate
                              # GPR[rt] <-- GPR[rs] + imm16

addiu rt, rs, imm16           # Unsigned addition with 16-bit immediate
                              # GPR[rt] <-- GPR[rs] + imm16

and   rd, rs, rt              # Bitwise logical AND of integers
                              # GPR[rd] <-- GPR[rs] AND GPR[rt]

andi  rt, rs, imm16           # Bitwise logical AND with 16-bit immediate
                              # GPR[rd] <-- GPR[rs] AND imm16

beq   rs, rt, offset          # Conditional branch if rs == rt
                              # PC <-- (rs == rt ? PC + 4 + offset <<l 2)
                              #                  : PC + 4)

blez  rs, offset              # Conditional branch if rs <= 0
                              # PC <-- (rs <= 0 ? PC + 4 + offset <<l 2)
                              #                 : PC + 4)

bltz  rs, offset              # Conditional branch if rs < 0
                              # PC <-- (rs < 0 ? PC + 4 + offset <<l 2)
                              #                : PC + 4)
```

```
bne   rs, rt, offset          # Conditional branch if rs != rt
                              # PC <-- (rs != rt ? PC + 4 + offset <<l 2)
                              #                  : PC + 4)

div   rs, rt                  # Integer division rs / rt
                              # LO <-- rs % rt
                              # HI <-- rs / rt

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

lb    rt, offset(rs)          # Copy a byte from memory to a register, with
                              #    sign-extension to 32 bits
                              # GPR[rt] <-- Mem[GPR[rs] + offset]

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

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

mfhi  rd                      # Copy register HI to GPR[rd]
                              # GPR[rd] <-- HI

mflo  rd                      # Copy register LO to GPR[rd]
                              # GPR[rd] <-- LO

mult  rs, rt                  # Signed multiplication of integers
                              # HI <-- (GPR[rs] * GPR[rt])(63:32)
                              # LO <-- (GPR[rs] * GPR[rt])(31:0)

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

or    rd, rs, rt              # Bitwise logical OR
                              # GPR[rd] <-- GPR[rs] OR GPR[rt]

ori   rt, rs, imm16           # Bitwise logical OR with 16-bit immediate
                              # GPR[rd] <-- GPR[rs] OR imm16

sb    rt, offset(rs)          # Copy a byte from a register to memory, with
                              #    sign-extension to 32 bits
                              # Mem[GPR[rs] + offset] <-- GPR[rt](7:0)

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[rd] <-- (GPR[rs] < imm16 ? 0 : 1)

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

```
srl   rd, rt, sa              # Arithmetic shift left a fixed number of bits
                              # GPR[rd] <-- GPR[rs] <<a sa

sub   rd, rs, rt              # Signed subtraction of integers

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

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

xor   rd, rs, rt              # Bitwise logical XOR
                              # GPR[rd] <-- GPR[rs] XOR GPR[rt]

xori  rt, rs, imm16           # Bitwise logical XOR with 16-bit immediate
                              # GPR[rd] <-- GPR[rs] XOR imm16
```

MIPS32 assembly .data section:

The assembly language specification for the data segment will be limited to the following elements:

```
var1:   .word 8192  # Creates one 32-bit integer called var1 and initializes
                    # it to 8192.

var2:   .byte 15    # Creates one 8-bit integer called var1 and initializes
                    # it to 15.
```

These declarations cannot entirely be reconstructed from a machine language representation. We can reasonably infer the existence of a variable, if we find an instruction that makes an access to the data segment, and we can infer the address of the data that is accessed. We can infer whether the access is to an 8-bit or a 32-bit value from the nature of the instruction that performs the access. Alas, we cannot infer the presence of an array in the data segment without seeing something like a linear access pattern at runtime.

Input

The input files will be MIPS32 machine language programs in binary text format. The machine language programs will be syntactically correct, compatible with the MARS MIPS simulator, and restricted to the subset of the MIPS32 instruction set defined above. Example input files will be available from the course website.

Each line in the input file will contain the representation of a single 32-bit word. The text segment will precede the data segment, and they will be separated by a single empty line.

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

```
disassem <input file> <output file>
```

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

The output will be an ASCII text file containing your translation of the given machine language program. As stated earlier, the text segment must precede the data segment, and they must be separated by a single empty line. The output file must be syntactically valid MIPS assembly code; that can be tested by loading it into MARS and assembling it.

One of the more interesting issues you will encounter is in identifying labels from the machine code. Your assembler will resolve all references to branch and jump targets in the text segment, and to variables in the data segment, and supply appropriate names in the generated assembly code. The rules for naming labels and variables follow.

First of all, the first instruction in the text segment will always carry the label `main`.

Aside from that, each label in the text segment must be identified when the first machine instruction that refers to it is parsed. Labels will be assigned names of the form `Lxx`, where `xx` is a sequence number, starting at `01`. See the example assembly output on page 1 for examples.

Each label in the data segment (variable names) must also be identified when the first machine instruction that refers to it is parsed. However, you will encounter some machine instructions that clearly access data memory, but for which you will not be able to assign a label; this is because it is not always possible to infer a numeric address when given a machine instruction that accesses data memory. Therefore, the rule will be that data segment labels will only be created when a machine instruction accesses data memory, and it's possible to compute a precise numeric address from that instruction. The naming convention for data segment labels will be the same as for text segment labels, except that data segment labels will be of the form `Vxx`.

Finally, variables declared in the data segment also have a type directive (`.asciiz`, `.word`, etc.). We will use the following imperfect convention for determining the correct type directive. If the first memory access instruction that implies a label performs a word-size access, we will assume the type directive `.word`, and similarly for `.byte`. We will not generate any other type directives.

Ultimately, for each posted input machine language file, your output file should match the corresponding assembly file, also posted on the course website.

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 opcodes to mnemonics, etc. There are many reasonable ways to organize the code for a system as large as this; my solution employs 15 `.c` files and corresponding header files and involves nearly 2000 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, and that zero heap bytes were in use when the program terminated. We will not be concerned about suppressed errors reported by Valgrind. See the discussion of Valgrind below.

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 checked by a human being after your solution has been autograded. The automated evaluation will certainly not check for these things. Failure to satisfy these requirements will result in

deductions from your autograding score; the potential size of those deductions is not being specified in advance (but you will not be happy with them).

Testing

A tar file containing a testing harness will be posted shortly. You should download and unpack the tar file on a CentOS 7 system, and follow the instructions in the `readme.txt` file. The test harness is precisely the tool we will use to evaluate your milestone and final submissions for this assignment; if your submitted tar file does not work properly with the posted test harness, you are very likely to receive a score of 0 for this project!

Until then, you will find a sample object file and corresponding disassembly file posted on the website, and you can use those for testing.

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 test harness for this project, with my solution, on Valgrind:

```
[wdm@centosvm ParseMI]$ valgrind --leak-check=full ./driver t01.o t01.txt -rand
==4976== Memcheck, a memory error detector
==4976== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4976== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==4976== Command: ./driver t01.o t01.txt -rand
==4976==
==4976==
==4976== HEAP SUMMARY:
==4976==     in use at exit: 0 bytes in 0 blocks
==4976==   total heap usage: 19 allocs, 19 frees, 1,996 bytes allocated
==4976==
==4976== All heap blocks were freed -- no leaks are possible
==4976==
==4976== For counts of detected and suppressed errors, rerun with: -v
==4976== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

And, I got good news... there were no detected memory-related issues with my code.

On the other hand, I ran Valgrind on my test harness for the DList project, and that did not go as well:

```
[wdm@centosvm DList]$ valgrind --leak-check=full ./driver t01.o t01.txt -rand
. . .
==5991==
==5991== HEAP SUMMARY:
==5991==     in use at exit: 120 bytes in 5 blocks
==5991==   total heap usage: 41 allocs, 36 frees, 4,248 bytes allocated
==5991==
==5991== 24 bytes in 1 blocks are definitely lost in loss record 3 of 5
==5991==    at 0x4C29BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==5991==    by 0x40337E: testEmpty (DListTestHarness.c:816)
==5991==    by 0x403219: testReporters (DListTestHarness.c:782)
==5991==    by 0x4010DF: testDList (DListTestHarness.c:102)
==5991==    by 0x403E48: main (driver.c:59)
==5991==
==5991== 48 (24 direct, 24 indirect) bytes in 1 blocks are definitely lost in loss record 4 of 5
==5991==    at 0x4C29BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==5991==    by 0x4034FF: testFront (DListTestHarness.c:854)
==5991==    by 0x40323C: testReporters (DListTestHarness.c:785)
==5991==    by 0x4010DF: testDList (DListTestHarness.c:102)
==5991==    by 0x403E48: main (driver.c:59)
==5991==
```

```
==5991== 48 (24 direct, 24 indirect) bytes in 1 blocks are definitely lost in loss record 5 of 5
==5991==    at 0x4C29BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==5991==    by 0x40374A: testBack (DListTestHarness.c:907)
==5991==    by 0x40326C: testReporters (DListTestHarness.c:788)
==5991==    by 0x4010DF: testDList (DListTestHarness.c:102)
==5991==    by 0x403E48: main (driver.c:59)
==5991==
==5991== LEAK SUMMARY:
==5991==    definitely lost: 72 bytes in 3 blocks
==5991==    indirectly lost: 48 bytes in 2 blocks
==5991==      possibly lost: 0 bytes in 0 blocks
==5991==    still reachable: 0 bytes in 0 blocks
==5991==         suppressed: 0 bytes in 0 blocks
==5991==
==5991== For counts of detected and suppressed errors, rerun with: -v
==5991== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 2 from 2)
```

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

Milestone

In order to assess your progress, there will be a milestone submission for the project, which will require that you submit a partial solution that achieves specified functionality. The milestone will be evaluated by using a scripted testing environment, which will be posted on the course website at least two weeks before the milestone is due. Your score on the milestone will constitute 10% of your final score on the project.

The milestone will be due approximately two weeks before the final project deadline. For the milestone, the text segment will be limited to the following instructions:

```
add   rd, rs, rt              # Signed addition of integers
                              # GPR[rd] <-- GPR[rs] + GPR[rt]

beq   rs, rt, offset          # Conditional branch if rs == rt
                              # PC <-- (rs == rt ? PC + 4 + offset <<l 2)
                              #                  : PC + 4)

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

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

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

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

addi  rt, rs, imm16           # Signed addition with 16-bit immediate
                              # GPR[rt] <-- GPR[rs] + imm16

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
```

Also, for this milestone, the data segment will be limited to no more than 4 words.

NO LATE SUBMISSIONS WILL BE ACCEPTED FOR THE MILESTONE!

What should I turn in, and how?

For both the milestone and the final submission, create a flat** uncompressed tar file containing:

- All the `.c` and `.h` files which are necessary in order to build your disassembler.
- A GNU makefile named "`makefile`". The command "`make disassembler`" should build an executable named "`disassem`". 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 disassembler to fail (e.g., it doesn't handle **`la`** instructions), telling us that may result in a more satisfactory evaluation of your disassembler.
- 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).

Grading

The evaluation of your solution will be based entirely on its ability to correctly translate programs using the specified MIPS32 assembly subset to MIPS32 machine code. 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.

At least two weeks before the due date for each milestone, we will release a tar file containing a testing harness (test shell scripts and test cases). You can use this tar file to evaluate your milestone 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 testing of your final disassembler submission will simply add test cases to cover the full range of specified MIPS32 instructions and data declarations, and to evaluate any extra-credit features that may be specified for this assignment. We will release an updated test harness for the final submissions at least two weeks before the final deadline.

Our testing of your disassembler and milestone 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. We will not offer any accommodations for submissions that do not work properly with the corresponding supplied test harness.

Test Environment

Your disassembler will be tested on the rlogin cluster, running 64-bit CentOS 7 and gcc 4.8. There are many of you, and few of us. Therefore, we will not test your disassembler 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 disassembler 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 disassembler does not crash on any valid input, even if it cannot produce the correct results. If you ensure that your disassembler 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 disassembler 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 machine instruction or data segment specification that your solution cannot handle, document that in the `readme.txt` file you will include in your submission. That may help in the evaluation of your solution, but do not expect a lot of special treatment.
- If there is a MIPS32 instruction or data segment specification 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 disassembler encounters a MIPS32 instruction it cannot handle, write a one-line descriptive message in place of the translation of that element. 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,
//      or any other unauthorized source, either modified or unmodified.
//
//    - If any C language code or documentation used in my program
//      was obtained from another source, such as a text book or course
//      notes, that has been clearly noted with a proper citation in
//      the comments of my program.
//
//    <Student Name>
```

Failure to include the pledge may result in your submission not being graded.

Advice

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

First, and most basic, analyze what your disassembler 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 assembly instructions in a number of ways as you build them. This decision has ramifications that will propagate throughout your implementation.

It helps to consider how you would carry out the translation from machine code to assembly code 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 about 1900 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. If you did a good job designing and implementing the earlier machine language parsing assignment, that should be a useful starting point.

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.

The disassembly process can be completed in a single pass through the input file, if you organize your work correctly.

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.

Static lookup tables are essential. Enumerated types are also extremely useful for representing various kinds of information, especially about type attributes of structured variables.

Consider implementing a program that will organize and support searches of a fixed collection of data records. For example, if the data records involve geographic features, we might employ a `struct` type:

```
// GData.h
...
enum _FType {CITY, BRIDGE, SUMMIT, BUILDING, ...};
typedef enum _FType FType;

struct _GData {
   char* Name;
   char* State;
   ...
   FType Category;
};
typedef struct _GData GData;
...
```

We might then initialize an array of `GData` objects by taking advantage of the ability to initialize **`struct`** variables at the point they are declared:

```
// GData.c
#define NUMRECORDS 500000
```

```
static GData GISTable[NUMRECORDS] = {
   {"New York", "NY", ..., CITY},
   {"Mount Evans", "CO", ..., SUMMIT},
   ...
   {"McBryde Hall", "VA", ..., BUILDING}
};
```

We place the table in the `.c` file and make it **`static`** so it's protected from direct access by user code. There's also a slight benefit due to the fact that **`static`** variables are initialized when the program loads, rather than by the execution of code, like a loop while the program is running.

Then we could implement any search functions we thought were appropriate from the user perspective, such as:

```
const GData* Find(const char* const Name, const char* const State);
```

Since **`struct`** types can be as complex as we like, the idea is applicable in any situation where we have a fixed set of data records whose contents are known in advance.

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 mnemonic, 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.

Here's a hint: I found it very useful to consider the "pattern" of the instruction in addition to the "format". I'll leave it to you to decide how to interpret the word "pattern".

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 instructions, 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 doubly-linked lists.

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.

Suggested resources

From the CS 2505 course website at:

http://courses.cs.vt.edu/~cs2505/summer2015/

you should consider the following notes:

Intro to Pointers	http://courses.cs.vt.edu/cs2505/summer2015/Notes/T14_IntroPointers.pdf
C Pointer Finale	http://courses.cs.vt.edu/cs2505/summer2015/Notes/T17_CPointerFinale.pdf
C struct Types	http://courses.cs.vt.edu/cs2505/summer2015/Notes/T24_CstructTypes.pdf
C Strings	http://courses.cs.vt.edu/~cs2505/summer2015/Notes/T15_CStrings.pdf

Some of the other notes on the basics of C and separate compilation may also be useful. The MIPS32 Instruction Set reference can be found on the CS 2506 Resources page:

http://courses.cs.vt.edu/cs2506/Fall2015/MIPSDocs/MD00086-2B-MIPS32BIS-AFP-02.50.pdf

This has detailed descriptions of all the supported instructions, with information about the machine code representation that will be invaluable on this assignment.

Credits

This project was inspired by the original formulation of a MIPS assembler project by Dr Srinidhi Vadarajan, who was then a member of the Dept of Computer Science at Virginia Tech. His sources of inspiration for that project are lost in the mists of time.

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

Change Log

Version	Posted	Pg	Change
1.00	Oct 13		Base document.
1.01	Oct 15	1	Added hint regarding base addresses for text and data segments.
		3	Added information about the use of the names `rd`, `rs`, `rt` and `sa` when describing instructions.
		4	Fixed parameter ordering for `slti`.
		5	Deleted array declaration syntax from requirements.
		6	Added a rule for determining type directives for data segment labels, and a clarification of what is meant by a "clean" run on Valgrind.
		7	Added more details about the upcoming test harness.
		9	Updated explanation of what a "flat" tar file is.