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 machine code input file, and the corresponding disassembly file:

```
10001100000100000010000000000000
10001100000100010010000000000100
10001100000100100010000000001000
10001100000100110010000000001100
00000010001100001000100000100000
00000010010100101001000000100000
00010010010100110000000000000001
00010110001100101111111111111100
00100000010000000000000000001010
00000000000000000000000000001100

00000000000000000000000000000001
00000000000000000000000000000100
00000000000000000000000000010000
00000000000000000000000001000000
```

```
00000000:  .text
00000000:  main:   lw      $s0,   <.data+0x0>
00000004:          lw      $s1,   <.data+0x4>
00000008:          lw      $s2,   <.data+0x8>
0000000C:          lw      $s3,   <.data+0xC>
00000010:  L01:    add     $s1,   $s1,   $s0
00000014:          add     $s2,   $s2,   $s2
00000018:          beq     $s2,   $s3,   L02
0000001C:          bne     $s1,   $s2,   L01
00000020:  L02:    addi    $zero, $v0,   10
00000024:          syscall

00002000:  .data
00002000:   0   0   0   1
00002004:   0   0   0   4
00002008:   0   0   0  10
0000200C:   0   0   0  40
```

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, which will list one word (32 bits) per line.

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 text segment labels and instructions, and between the data accesses (by instructions) and data values.

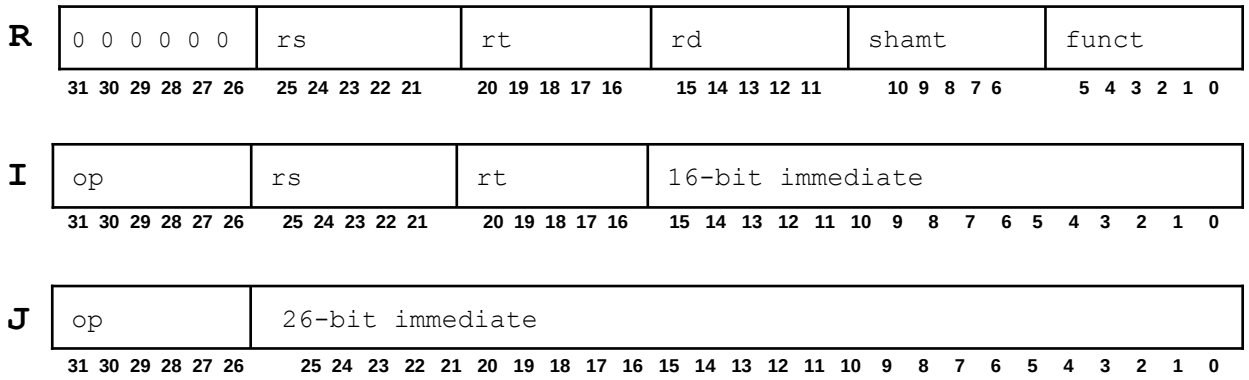
The disassembled text segment begins with the `.text` directive. Each instruction must be preceded by its address (in hex). The instructions should be formatted as neatly as possible, with sensible indentation and separation of parameters. You are advised to use spaces, rather than tabs, to achieve a neat alignment of your output. Text segment labels must be on the same line as the corresponding instruction.

The disassembled data segment simply lists the values of the bytes that make up each word (in hex), along with the address of that word (in hex).

The disassembled text segment must be shown first, followed by a single empty line, followed by the disassembled data segment.

Translating from Machine Langugage to Assembly Code

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



The Basics

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.

Disassembling Memory-Access Instructions

Memory-access instructions are those that read from or write to the data memory (i.e., to the data segment of the program).

Consider the machine instruction: `10001100000100000010000000001100`

The opcode, `100011`, corresponds to the I-format `lw` instruction, which causes a word (4-byte value) to be transferred from data memory into a register. The register number fields, `00000` and `10000`, correspond to registers 0 (`$zero`) and 16 (`$s0`). The immediate field, `0010000000001100`, corresponds to the value 8204 (0x200C).

Now, the `lw` instruction accesses memory at the address that equals sum of the value in the `rs` register and the immediate field in this instruction. Since the `rs` register is `$zero`, we know its value is 0, and hence the address being accessed is 0x200C. (If the `rs` field was any other register, we could not determine the exact address.) But, in this case, we do know the exact address, and we can specify that address in our translation of the instruction.

Now, the data value being accessed lies in the data segment, which begins at address 0x2000, so this data value is at the offset 0x0C from the beginning of the data segment. We can express this address (in *relative address* form) as `.data+0xC`.

Therefore, we disassemble this machine instruction as: `lw  $s0, <.data+0xC>`

On the other hand, suppose we had the similar machine instruction: `10001110001100000000000000001100`

Now, the `rs` register is 10001 or 17 (`$s1`). The immediate field is 12 or 0xC. Since that register could store any value, we cannot determine a specific address; the most we can say is that the address is `$s1 + 0xC`, which is expressed in MIPS assembly syntax as a register-relative address: `0xC($s1)`.

So, we would disassemble this machine instruction as: `lw  $s0, 0xC($s1)`

Disassembling Conditional Branch and Jump Instructions

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

First of all, a label is just an alphanumeric string. Labels occur in two contexts. A label may be *referred to* within an assembly code instruction, or be *defined* by being used preceding an assembly code instruction:

```
L01:    add   $s1,   $s1,   $s0
        . . .
        bne   $s1,   $s2,   L01
```

When being defined, labels are followed by a colon (`':'`).

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 disassembled. Labels will be assigned names of the form `Lxx`, where `xx` is a sequence number, starting at `01`. As shown in the example on page 1, labels are to be numbered sequentially, according to where they are defined (not necessarily in the order they are referred to).

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 MIPS32 assembly language specification for the data segment will have no restrictions (not that this really matters to the logic of the disassembler.

Data type information disappears during the translations from high-level language to assembly language and from assembly language to machine language. There is, in general, no reliable way to determine data type information from machine code. At best, we could reliably infer whether data was being accessed in chunks of 1 or 2 or 4 bytes. (To be fair, some machine instructions are integer-specific and others are floating-point-specific, but all of those can be used in creative ways that do not conform to the original data declarations in a high-level language.) Therefore, your disassembler will make no attempt to infer data types, or even pay attention to whether accesses are to 1 or 2 or 4 byte values.

So, what can your disassembler do? Obviously, we can parse the bytes given in the data segment and display the values of those bytes, and the addresses at which they occur. We might 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 actual address of the data that is accessed. However, even in that case we cannot be sure whether this is an access to a variable or to a piece of a larger structure, such as an array element. So, your disassembler will not assign names (labels) to elements of the data segment.

Command-line Interface

Your disassembler will be invoked in following way:

```
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.

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 14 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, 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.

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 is available on the course website. 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!

You should carefully examine the contents of the log files the test harness writes when it's applied to your solution. These not only contain score information, but they often indicate details that may relate to errors that have been found.

If you experience a segmentation fault, or some other runtime error, it's best to execute your solution directly on an input file for which the error occurs, and use `gdb` to analyze exactly what is going on.

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 this 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 disassem C3TestFiles/ref07.o stu_ref07.asm
==27447== Memcheck, a memory error detector
==27447== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==27447== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==27447== Command: disassem C3TestFiles/ref07.o stu_ref07.asm
==27447== Parent PID: 3523
==27447==
==27447==
==27447== HEAP SUMMARY:
==27447==     in use at exit: 0 bytes in 0 blocks
==27447==   total heap usage: 226 allocs, 226 frees, 6,720 bytes allocated
==27447==
==27447== All heap blocks were freed -- no leaks are possible
==27447==
==27447== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
==27447==
==27447== 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 a different solution to this project, and the results were less satisfactory:

```
[wdm@centosvm temp]$ valgrind --leak-check=full ./driver t01.o t01.txt -rand
. . .
==7962== HEAP SUMMARY:
==7962==     in use at exit: 4,842 bytes in 331 blocks
==7962==   total heap usage: 331 allocs, 0 frees, 4,842 bytes allocated
==7962==
==7962== Searching for pointers to 331 not-freed blocks
==7962== Checked 111,584 bytes
==7962==
==7962== 7 bytes in 1 blocks are definitely lost in loss record 1 of 26
==7962==    at 0x4C2B974: calloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==7962==    by 0x4020B3: parseJTypeInstruction (ParseInstructions.c:178)
==7962==    by 0x400F51: main (Disassembler.c:137)
. . .
==7962== LEAK SUMMARY:
==7962==    definitely lost: 3,116 bytes in 233 blocks
==7962==    indirectly lost: 590 bytes in 96 blocks
==7962==      possibly lost: 0 bytes in 0 blocks
==7962==    still reachable: 1,136 bytes in 2 blocks
==7962==         suppressed: 0 bytes in 0 blocks
==7962==
==7962== ERROR SUMMARY: 116 errors from 34 contexts (suppressed: 2 from 2)
```

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.

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.

The requirement for a clean run on Valgrind will not be applied during grading of the milestone, but you are encouraged to make sure that your milestone submission does satisfy that requirement, since that will make it much easier to do so later.

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).

You can also tell the difference by simply doing a table-of-contents listing of your tar file. 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
disassem.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
```

Grading

The evaluation of your milestone and final solutions will be based primarily 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.

As mentioned earlier, we do expect your solution to run cleanly on Valgrind. We will check that by running your solution on a single test file. A penalty of up to 20% will be assessed, based upon the number of invalid reads and writes, the percentage of dynamic allocations that were deallocated, and the number of heap bytes reported to still be in use when the program exits. (If you sidestep this by not making sufficient use of dynamic allocation, we will just assess the maximum penalty.)

At least two weeks before the due date for the 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 a score of zero. It is your responsibility to make sure your submission works correctly with the supplied testing harness. We will not accommodate any deviations.

The testing of your final disassembler submission will simply add test cases to cover the full range of specified MIPS32 instructions and data declarations. 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.

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 being ignored.

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 somewhat sparse comments, runs about 2100 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.

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.

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 Department 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 Department of Computer Science at Virginia Tech. Any errors, ambiguities, eccentricities, and omissions should be attributed to him.

Change Log

Version	Posted	Pg	Change
2.00	Jan 19		Base document.
2.01	Mar 21	3	Corrected literal value in disassembly output.
2.02	Mar 24	1	Added missing colons (`':'`) following instruction/data addresses
2.03	Mar 28	1	Fixed order of register numbers in machine code for `addi` instruction