C Programming Simple Parsing, Processing Binary Representations, Tables

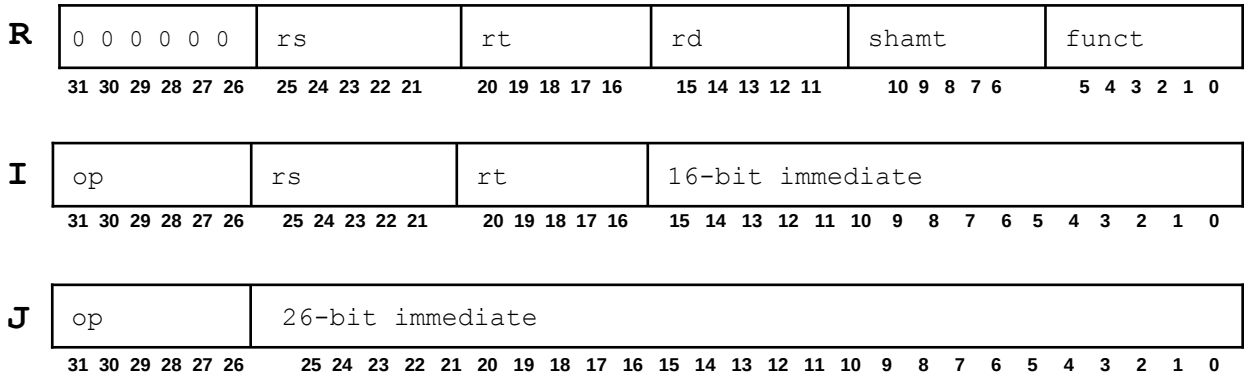
For this assignment, you will implement a C program that reads MIPS32 machine language instructions and computes various fields from them. More precisely, your program will read instructions from an input file (name supplied on the command line) and write representations of various fields to a second file (name also supplied on the command line).

The input file will contain a list of valid MIPS32 machine instructions, written in text format, with one instruction per line. Here's the beginning of an example input file:

```
00100000000010000010000000000000
10001101000010000000000000000000
00100000000010010010000000001000
. . .
```

Each machine instruction 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 will vary, and your solution must process all of them and halt correctly at the end of the input file.

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



In order to decide how to execute a machine instruction, we would need to know exactly which machine instruction we are dealing with. If the `opcode` field is `000000`, we have an R-format machine instruction, and the `funct` field tells us exactly which R-format machine instruction we have. Otherwise, the `opcode` field tells us which machine instruction we have.

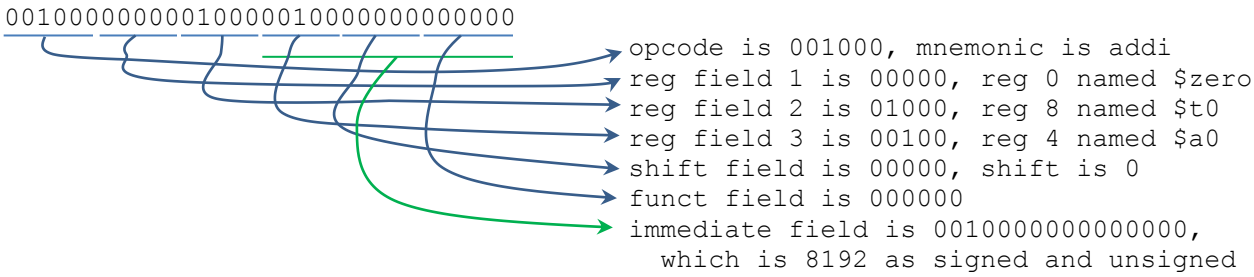
Then, we would know exactly how to parse the machine instruction to determine the relevant information, like register numbers, immediate fields, and so forth.

Those considerations would come into play if we wanted to "read" a machine instruction and determine how to execute it, or how to disassemble it and construct an equivalent assembly instruction (which will be a later project).

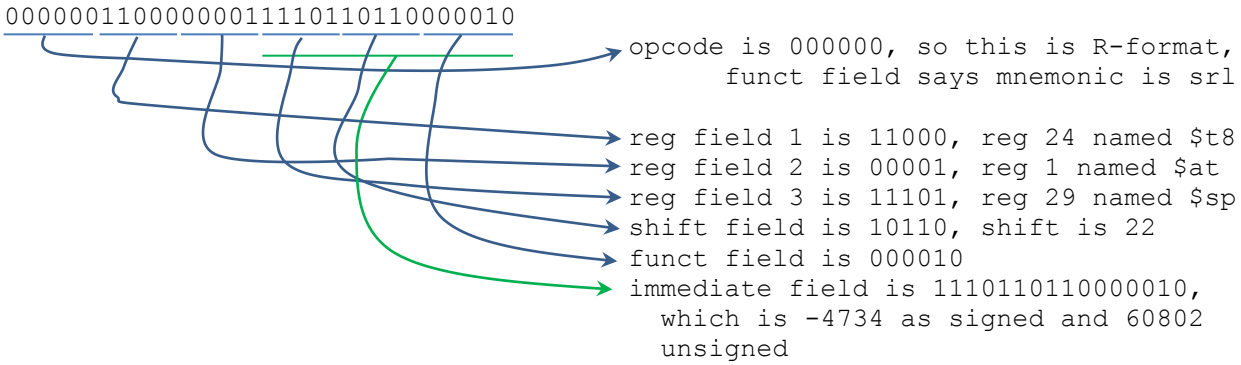
But, for now, we are only interested in determining how to identify all the possible fields, and to interpret them in useful ways. So, for each machine instruction, you must compute the following information (and store it in a `struct` variable, as shown later in this specification):

- the bits that form the instruction
- the bits in the opcode field and the mnemonic for the instruction
- for each possible register field, the register number in base-10 and the register name
- the base-10 value of the shift amount field
- the bits in the funct field
- the immediate field, as a signed integer and as an unsigned integer

For example, given the first machine instruction above, you should obtain the following information:



As another example:



So, how do we know all that? Reading. The course notes and relevant discussions in P&H reveal all...

You will implement a C function that takes a representation of a MIPS32 machine instruction as input and parses it, as illustrated above. For communication, you will have the function return a C object (`struct` variable) of the following type:

```
struct _ParseResult {
   // Raw machine code portion
   // These are malloc'd zero-terminated C-strings
   char* MInstruction;      // the complete machine instruction
   char* Opcode;            // the extracted opcode field bits
   char* Funct;             // the extracted funct field bits

   // Interpreted asm code portion
   char*    Mnemonic;       // the symbolic name of the instruction
   uint8_t  rd;             // the three register fields, as small unsigned
   uint8_t  rs;             //    integers
   uint8_t  rt;
   char*    rdName;         // the names of the registers, as C-strings
   char*    rsName;
   char*    rtName;
   uint8_t  shft;           // the shift field, as a small integer
   uint16_t unsignedImm;    // the immediate field, as an unsigned integer
   int16_t  signedImm;      // the immediate field, as a signed integer
};
typedef struct _ParseResult ParseResult;
```

Be careful of the fields that are identified as zero-terminated C-strings. You must allocate the necessary arrays dynamically, and make sure that each of the C-strings is, indeed, properly terminated. Also be sure to understand the definition of a *proper* `ParseResult` object (from the posted header file), and to pay attention to where that term occurs in pre- and post-conditions.

The instruction parsing function will have the following interface:

```
/**  Breaks up given MIPS32 machine instruction and displays information about
 *   all possible fields.
 *
 *   Pre:  MI points to an array holding the bits (as chars) of one of the
 *         MIPS32 instructions listed in the project specification.
 *
 *   Returns:
 *         A pointer to a proper ParseResult object, whose fields have
 *         been correctly initialized to correspond to the target of MI.
 */
ParseResult* parseMI(const char* const MI);
```

Some Coding Requirements

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

This assignment is, in some ways, a warm-up for the disassembler project. Therefore, if you give careful thought to your design, you can produce lots of C code that can be plugged into the disassembler. And, if you choose to do this in minimalist fashion, you'll gain little or nothing towards implementing the disassembler.

You are required* to implement static lookup tables and use them to determine instruction mnemonics, and to map register numbers to register names. See the discussion of static lookup tables later in 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.

You are required* to provide a "destructor" function for the `ParseResult` type; that is, a function that deallocates all the dynamic content from a `ParseResults` object. The interface for this function must be:

```
/**  Frees the dynamic content of a ParseResult object.
 *
 *   Pre:  pPR points to a proper ParseResult object.
 *   Post: All of the dynamically-allocated arrays in *pPR have been
 *         deallocated.
 *         *pPR is proper.
 *
 *   Comments:
 *     -  The function has no information about whether *pPR has been
 *        allocated dynamically, so it cannot risk attempting to
 *        deallocate *pPR.
 *     -  The function is intended to provide the user with a simple
 *        way to free memory; the user may or may not reuse *pPR.  So,
 *        the function does set the pointers in *pPR to NULL.
 */
void clearResult(ParseResult* const pPR);
```

The test harness will call `clearResult()` at appropriate times during testing. If you have correctly implemented the function, and otherwise coded your solution correctly, tests run on `valgrind` will not indicate any memory leaks.

We will require* your solution to achieve a "clean" run on `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).

Required MIPS Machine Instruction Features

Your solution must correctly parse machine instructions with the following opcode fields:

```
000000    (with funct fields: 000000   000010   001100   100000   100100)
000001    000100   000101   000110   001000   001001
001010    001100   001111   100011   101011
```

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. For this assignment, you will be doing a very general parse of each machine instruction, but you will still have to take into account what the specific instruction is in order to determine the mnemonic.

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 it is interpreted as a signed or unsigned value. It might be helpful to recall that signed integers are represented in 2's complement form.

Static Lookup Tables in C

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
...
struct _GData {
   char* Name;
   char* State;
   ...
   uint16_t Elevation;
};
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 50

static GData GISTable[NUMRECORDS] = {
   {"New York", "NY", ..., 33},
   {"Los Angeles", "CA", ..., 305},
   ...
   {"Chicago", "IL", ..., 594}
};
```

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.

Testing

A test/grading harness, posted on the course website, includes the following files, and corresponding header files (some in subdirectories):

```
// test drivers
driver.c       // driver for running all the tests and scoring; see embedded comments

// shell files for solution (you complete these)
ParseResult.c
MIParser.c

// 64-bit testing code
Generate.h
Generate.o
Grading.h
Grading.o
```

Unpack the posted tar file, and put your completed implementation files (`MIParser.c` and `ParseResult.c`) into the top-level directory. You can then compile with the following commands:

```
gcc -o driver -std=c99 -m64 -Wall -ggdb3 *.c *.o
```

This will probably not work on CentOS 6 due to the presence of old libraries, but it does work on rlogin.

You should also note that the posted code will, indeed, compile. And, if you execute it as is it will segfault because the given implementation of `parseMI()` merely returns `NULL`.

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 parseMI]$ valgrind --leak-check=full --show-leak-kinds=all --log-file=vlog.txt
--track-origins=yes -v driver instr.txt parse.txt -rand
```

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

```
==10669== Memcheck, a memory error detector
==10669== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==10669== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==10669== Command: driver instr.txt parse.txt -rand
==10669==
==10669== HEAP SUMMARY:
```

```
==10669==     in use at exit: 0 bytes in 0 blocks
==10669==   total heap usage: 275 allocs, 275 frees, 6,904 bytes allocated
==10669==
==10669== All heap blocks were freed -- no leaks are possible
==10669==
==10669== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
==10669==
==10669== 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. That's the sort of results you want to see when you try your solution with Valgrind.

What to Submit

You will submit a "flat", uncompressed tar file containing your completed versions of `MIParser.c` and `ParseResult.c`; do not alter the names of those files. In addition, include any other C source files (`.c` and/or `.h`) that you have written as part of your solution.

This assignment will be graded automatically, using the supplied test harness. Your submitted files will be copied into a directory, along with the files from the posted tarball (aside from the original versions of `MIParser.c` and `ParseResult.c`). Note that any files you submit that have the same names as files contained in the posted tarball will be overwritten.

Your submission will be compiled with the following `gcc` command:

```
gcc -o driver-std=c99 -m64 -Wall -ggdb3 *.c
```

Test your solution thoroughly before submitting it. Make sure that your solution produces correct results for every logically valid test case you can think of.

The *Student Guide* and other pertinent information, such as the link to the proper submit page, can be found at:

http://www.cs.vt.edu/curator/

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
```

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted 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>
```

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/Spring2016/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.