C Programming — String Parsing and Table Lookup

For this assignment, you will implement a C function that parses a restricted subset of MIPS assembly instructions and prints out information relevant to translating those instructions into machine code. In particular, you will be concerned with MIPS assembly instructions, expressed in one of the following forms:

```
R-format:  mnemonic   reg1, reg2, reg3        (mul, sub)
I-format:  mnemonic   reg1, reg2, immediate   (addi)
           mnemonic   reg1, immediate         (lui)
           mnemonic   reg1, offset(reg2)      (lw)
SPECIAL2:  mnemonic   reg1, reg2              (mult)
```

where `mnemonic` is one of the following MIPS32 mnemonics:

`addi`   `mul`   `mult`   `lui`   `lw`   ~~`sw`~~   `sub`

and `reg1`, `reg2` and `reg3` are each one of the following MIPS registers:

`$t0, $t1, $t2, $t3, $s0, $s1, $s2, $s3`

and `immediate` and `offset` are integers in the range -32768 to 32767. The elements of each instruction are separated by whitespace and commas, as shown. Whitespace can be any mixture of spaces and tab characters.

The instructions you are concerned with in this assignment can be classified as shown above; but `mult`, `lui`, and `lw` break the ideal format rules the MIPS designers proposed for the instruction set architecture. You should consult the *MIPS32 Architecture Volume 2: the MIPS32 Instruction Set*, which is available on the Resources page of the course website for details of the machine language representations of these instructions. They can be viewed as illustrating the notion that a good design often requires (hopefully good) compromises from the pure design goals you might start with.

In order to decide how to interpret an assembly instruction, we would need to know exactly which assembly instruction we are dealing with, since different assembly instructions follow different patterns. For example, if the mnemonic is `add`, we have an R-format machine instruction, and we know that the instruction has three parameters that are all register names.

How do we know these things? From consulting the available MIPS32 references, chiefly the *MIPS32 Architecture Volume 2: the MIPS32 Instruction Set*, which is available on the Resources page of the course website. From there, we find that `add` is an R-format instruction, and that the `add` assembly instruction always has the form:

```
add   $rd, $rs, $rt
```

We also find that executing this instruction results in the assignment: `$rd = $rs + $rt`.

Moreover, we find that this is expressed in binary machine format as:

R	0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	1 0 0 0 0 0
	31 30 29 28 27 26	**25 24 23 22 21**	**20 19 18 17 16**	**15 14 13 12 11**	**10 9 8 7 6**	**5 4 3 2 1 0**

We also find, from other MIPS32 references, how the symbolic register names map to integer register numbers, so if we are given a specific instance of the `add` assembly instruction, we can determine all the components of the binary representation.

A later assignment will require you to implement a C program that translates complete MIPS32 assembly programs into machine code. For now, we will focus on the narrower problem of determining the pieces that make up the representations of a small selection of assembly instructions.

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

Given one of the MIPS32 assembly instructions mentioned earlier, you will create a C **struct** variable that contains information relevant to the specific assembly instruction and its representation in machine code. We will use the following user-defined C type to represent your analysis of the given instruction:

```
/**  Represents the possible field values for a MIPS32 machine instruction.
 *
 *   A ParseResult object is said to be proper iff:
 *
 *     - Each of the char* members is either NULL or points to a zero-
 *       terminated C-string.
 *     - If ASMInstruction is not NULL, the contents of the array represent
 *       a MIPS32 assembly instruction.
 *     - If ASMInstruction is not NULL, the other fields are set to properly
 *       represent the corresponding fields of the MIPS32 assembly instruction
 *       stored in ASMInstruction.
 *     - Each field that is not relevant to the MIPS32 assembly instruction
 *       is set as described in the comments below.
 */
struct _ParseResult {
   // Each char* member will be NULL or point to dynamically-allocated char array
   // holding a zero-terminated C string.

   // The assembly code portion
   char* ASMInstruction;     // the assembly instruction, as a C-string
   char* Mnemonic;           // the symbolic name of the instruction
   char* rdName;             // the symbolic names of the registers, as C-strings;
   char* rsName;             //     NULL if the register field is not specified
   char* rtName;             //     in the assembly instruction

   //   The following are integer values
   int16_t  Imm;             // the immediate field, as a signed integer;
                             //    0 if not relevant for the assembly instruction
   uint8_t  rd;              // the three register fields, as small unsigned integers;
   uint8_t  rs;              //    255 if not relevant for the assembly instruction
   uint8_t  rt;

   // The computed machine code portion
   //   These are malloc'd zero-terminated C-strings
   char* Opcode;             // the opcode field bits
   char* Funct;              // the funct field bits
                             //    NULL if not relevant for the assembly instruction
   char* RD;                 // the bit representations of the register numbers;
   char* RS;                 //    NULL if not relevant for the assembly instruction
   char* RT;
   char* IMM;                // 2's complement bit representation of the immediate;
                             //    NULL if not relevant for the assembly instruction
};
```

This type includes every possible component related to any of the assembly instructions in which we are interested. However, no particular MIPS32 assembly instruction actually has all of the possible components defined in this type, so we will stipulate that are unused will be set to default values, given in the comments above.

For example, suppose you have the assembly instruction: `add   $s3, $t1, $t0`

The corresponding `ParseResult` object should contain the following information, parsed directly from the given instruction:

```
ASMInstruction --> "add  $s3, $t1, $t0"
Mnemonic --> "add"
rdName   --> "$s3"
rsName   --> "$t1"
rtName   --> "$t0"
```

The following information can be obtained from properly-constructed static lookup tables:

```
rd == 19
rs ==  9
rt ==  8
Opcode --> "000000"
Funct  --> "100000"
```

The following information can be computed from the values above:

```
RD --> "10011"
RS --> "01001"
RT --> "01000"
```

Finally, the remaining fields are just set to their specified defaults:

```
Imm == 0
IMM == NULL
```

Given the assembly instruction `"addi   $t0, $s2, -42"`, we find it has the form:

```
addi   $rt, $rs, immediate
```

And, `addi` is an I-format instruction, whose binary representation is:

I	0 0 1 0 0 0	rs	rt	16-bit immediate
	31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

We would obtain the following values:

```
ASMInstruction --> "addi  $t0, $s2, -42"
Mnemonic --> "addi"
rdName == NULL
rsName --> "$s2"
rtName --> "$t0"
rd == 255
rs ==   8
rt ==  18
Opcode --> "001000"
Funct  == NULL
RD == NULL
RS --> "01000"
RT --> "10010"
Imm --> -42
IMM == "1111111111010110"
```

Coding Requirements

You will implement the following C function:

```
/**  Breaks up given the MIPS32 assembly instruction and creates a proper
 *   ParseResult object storing information about that instruction.
 *
 *   Pre:  pASM points to an array holding the representation of a
 *         syntactically valid assembly instruction, whose mnemonic is
 *         one of the following:
 *
 *             add  addi  and  andi  lui  lw  sub
 *
 *         The instruction will be formatted as follows:
 *
 *             <mnemonic><ws><operand1>,<ws><operand2>,<ws>...
 *
 *         where <ws> is an arbitrary mixture of space and tab characters.
 *
 *   Returns:
 *         A pointer to a proper ParseResult object whose fields have been
 *         correctly initialized to correspond to the target of pASM.
 */
ParseResult* parseASM(const char* const pASM);
```

The stated precondition will be satisfied whenever the testing code calls your implementation. Your implementation must satisfy the stated return specification, and must not violate `const` or create memory violations of any kind.

You are required* to implement static lookup tables and use them to determine instruction opcodes from 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 mnemonics to opcodes, etc.

You are also 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 evaluated by running the supplied grading script. The grading script will certainly not check for modularity, but it will perform a Valgrind check. Failure to satisfy these requirements will result in a deduction of up to 10% from your autograding score.

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

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", ..., CITY, 33},
   {"Pikes Peak", "CO", ..., MOUNTAIN, 14115},
   ...
   {"McBryde Hall", "VA", ..., BUILDING, 2080}
};
```

We place the table in the `.c` file and make it **`static`** so it's protected from direct access by user code in other files. 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.

Memory Management Requirements and 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 result you want to see when you try your solution with Valgrind.

On the other hand, here’s what I got from a student submission:

```
==8596== Memcheck, a memory error detector
==8596== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8596== Using Valgrind-3.14.0-353a3587bb-20181007X and LibVEX; rerun with -h for copyright info
==8596== Command: c02driver tests.txt results.txt -rand
==8596== Parent PID: 8595
. . .
==8596== Invalid read of size 1
==8596==    at 0x4E84029: vfprintf (in /usr/lib64/libc-2.17.so)
==8596==    by 0x4E8A3B6: fprintf (in /usr/lib64/libc-2.17.so)
==8596==    by 0x402860: refprintResult (in /home/wmcquain/2506/c02/broken/c02driver)
==8596==    by 0x401E19: scoreResult (in /home/wmcquain/2506/c02/broken/c02driver)
==8596==    by 0x40176D: main (in /home/wmcquain/2506/c02/broken/c02driver)
==8596==  Address 0x5205c15 is 0 bytes after a block of size 5 alloc'd
==8596==    at 0x4C2BFB9: calloc (vg_replace_malloc.c:762)
==8596==    by 0x4010F6: parseASM (in /home/wmcquain/2506/c02/broken/c02driver)
==8596==    by 0x401727: main (in /home/wmcquain/2506/c02/broken/c02driver)
. . .
==8596== 21 bytes in 3 blocks are definitely lost in loss record 3 of 17
==8596==    at 0x4C2BFB9: calloc (vg_replace_malloc.c:762)
==8596==    by 0x403FB1: copyOf (in /home/wmcquain/2506/c02/broken/c02driver)
==8596==    by 0x401417: parseASM (in /home/wmcquain/2506/c02/broken/c02driver)
==8596==    by 0x401727: main (in /home/wmcquain/2506/c02/broken/c02driver)
==8596==
==8596== 21 bytes in 3 blocks are definitely lost in loss record 4 of 17
==8596==    at 0x4C2BFB9: calloc (vg_replace_malloc.c:762)
==8596==    by 0x4035D6: refcopyOf (in /home/wmcquain/2506/c02/broken/c02driver)
==8596==    by 0x40346E: refparseASM (in /home/wmcquain/2506/c02/broken/c02driver)
==8596==    by 0x401DE7: scoreResult (in /home/wmcquain/2506/c02/broken/c02driver)
==8596==    by 0x40176D: main (in /home/wmcquain/2506/c02/broken/c02driver)
==8596==
==8596== 35 bytes in 5 blocks are definitely lost in loss record 5 of 17
==8596==    at 0x4C2BFB9: calloc (vg_replace_malloc.c:762)
==8596==    by 0x403FB1: copyOf (in /home/wmcquain/2506/c02/broken/c02driver)
```

```
==8596==    by 0x401267: parseASM (in /home/wmcquain/2506/c02/broken/c02driver)
==8596==    by 0x401727: main (in /home/wmcquain/2506/c02/broken/c02driver)
. . .
==8596==
==8596== LEAK SUMMARY:
==8596==    definitely lost: 1,508 bytes in 105 blocks
==8596==    indirectly lost: 0 bytes in 0 blocks
==8596==      possibly lost: 0 bytes in 0 blocks
==8596==    still reachable: 0 bytes in 0 blocks
==8596==         suppressed: 0 bytes in 0 blocks
==8596==
==8596== ERROR SUMMARY: 25 errors from 18 contexts (suppressed: 0 from 0)
```

Now, that solution needs some work… but the Valgrind output gives good clues.

Testing and Grading

Download the posted tar file, `c02Files.tar` from the course website and unpack it on a CentOS 7 system. You should receive the following files, organized in two subdirectories:

```
README                 - usage instructions
dev:
   c02driver.c         - test driver
   ASMParser.h         - "public" interface for parser; do not modify!
   ASMParser.c         - implement the functions here, as needed
   ParseResult.h       - "public" interface for results type; do not modify!
   ParseResult.c       - implement the functions here, as needed
   Grader.h            - declaration for grading function
   Grader.o            - 64-bit CentOS binary for grading function
   Generate.h          - declaration for test data generator
   Generate.o          - 64-bit CentOS binary for test data generator

grading:
   gradeC02.sh         - this script file
   c02Grader.tar       - grading code, including:
      c02driver.c      - test driver
      ASMParser.h      - supplied C header file
      ParseResult.h    - supplied C header file
      Grader.h         - declaration for grading function
      Grader.o         - 64-bit CentOS binary for grading function
      Generate.h       - declaration for test data generator
      Generate.o       - 64-bit CentOS binary for test data generator
```

Unpack the posted tar file, and complete the implementation files (`ASMParser.c` and `ParseResult.c`) in the `dev` directory. See the `README` file for instructions.

You should also note that the posted code will, indeed, compile. And, if you execute it as is it will not perform correctly, because the given implementation of `parseASM()` merely returns `NULL`.

If you want to use gdb (and you do), you need to avoid the grading script. On the other hand, you should definitely use the grading script to verify the tar file you plan to submit is complete, and in the proper format.

The supplied grading script will produce a raw score, shown in the results log. The following deductions may be applied when the TAs assess your solution manually:

up to 10% memory management or access errors found by Valgrind
up to 5% poor modularity in your implementation

What to submit

You will submit an uncompressed tar file containing your completed C implementation files (`ASMParser.c` and `ParseResult.c` and any supporting `.c` and `.h` files you have written), and nothing else. Do not include any of the supplied files from `C02Grader.tar` (see above), object files, or executable files.

In a week or so, we will post a shell script that will automate the grading process. Once your solution passes the testing code above, you should test your tar file with that shell script to be sure everything meets our requirements.

We will grade your submission with the posted test/grading harness, and we will make no allowances for submissions that do not operate correctly with that. Be warned: we will use the original, posted versions of the header and `.o` files in grading your submission, so you may encounter problems if you've modified any of those.

Make your submission to the posted link for the Curator system. Late submissions will be assessed a penalty of 10% per diem.

Suggested resources

Aside from a good C language reference (see the Resources page on the course website), the following sources of information should prove useful:

Computer Organization and Design: the Hardware/Software Interface
good overall description of MIPS32 architecture, register names/numbers, etc.

MIPS32 Architecture Volume 2: the MIPS32 Instruction Set (on the Resources page)
good details on specific MIPS32 assembly instructions and their representations

From the CS 2505 course website at:

http://courses.cs.vt.edu/~cs2505/fall2019/

you should consider the following notes:

Intro to Pointers	T11.IntroPointers.pdf
C Strings	T12.CStrings.pdf
C String I/O	T12b.CStrings.pdf (link is on the CS 2505 Assignments page with C04)
C Pointer Finale	T14.CPointerFinale.pdf
C struct Types	T18.CstructTypes.pdf

Some of the other notes on the basics of C and separate compilation may also be useful.

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

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

Change Log Relative to Version 5.00

Version	Posted	Pg	Change
5.00	Feb 2		Base document.