CS4121 Cminus Expression Evaluation Compiler Project
Due Date: Friday, Nov. 3, 2017 at *8am*

Purpose

The purpose of this project is to gain experience in giving meaning to a programming language by generating Mips assembly for a subset of Cminus. Specifically, you will be generating assembly for integer I/O operations, integer arithmetic and logical expressions and assignment statements.

Project Summary

In this project, you will add actions to a parser provided by me. You must add code to do the following:

1. Assign space for global and local integer variables declared in a Cminus program.
2. Generate assembly pseudo-ops for string constants used in a Cminus program.
3. Generate assembly to print string constants.
4. Generate assembly to print integers.
5. Generate assembly to read integers.
6. Generate assembly to compute integer expressions.
7. Generate assembly to compute logic expressions.
8. Generate assembly to assign values to integer variables.

Prologue and Epilogue Code

Since you are converting a Cminus program to Mips assembly, you must begin each assembly file with any data declarations and a declaration of where the `main` program begins. This is done with the following code:

```
        .data
.newl:  .asciiz "\n"
        .text
        .globl main
main:   nop
```

This code declares a data section with a string `.newl` that is just the newline character, followed by a text section (instructions) containing a declaration of the main routine. Each Mips assembly file should begin with this sequence. Any space that you allocate for strings or floating-point constants in the static data area may be allocated with directives after the ".data" directive and before the ".text" directive.

Assigning Variable Space

Memory for global variables declared in a Cminus program will be address as an offset off of `$gp`. The register `$gp` points to the middle of a 64K region in the static data area. You may address this area as a positive or negative offset off of `$gp`. I will guarantee that you will need no more than 64K of space in the static data area for any input program for any of the projects in this class.

Memory for local variables is allocated on the stack. Each integer requires four bytes of space. Local space is allocated by adjusting the stack pointer the requisite number of bytes. Since stacks grow in the negative direction in memory, space is allocated by subtracting from the stack pointer. The Cminus declarations

```
int i,j,k;
```

require 12 bytes of space. That space is allocated on the stack with the instructions

```
sw $fp, ($sp)
move $fp, $sp
sub $sp, $sp, 12
```

which should be placed *immediately* following the prologue code. The first instruction stores the old frame pointer. The second sets the new frame pointer, `$fp`, and the third instruction allocates the space for the 3 variables.

String Constants

A Cminus program may use string constants in write statements. These constants are declared in the data section using the `.asciiz` pseudo-op. For the Cminus statement,

```
write('Hello');
```

The following declaration must be added to the data section of the assembly file:

```
.string0: .asciiz "Hello"
```

The label `.string0` is implementation dependent. You may name your string constants however you wish.

Printing Strings

Printing strings requires using a system call. The system call service for printing strings is 4. Since a character string is stored in memory, you must pass the address of the string to the system call in register `$a0`. As an example, the code to implement the `write` statement in the previous section would be:

```
    la $a0, .string0
    li $v0, 4
syscall
```

Note that you will need to additionally print the newline character when printing any data.

Printing Integers

Printing integers is similar to printing strings except that the actual integer is passed to the system call rather than an address and the system call service is 1. As an example, to implement the statement:

```
write(7);
```

the following Mips assembly would need to be generated:

```
li $a0, 7
li $v0, 1
syscall
```

Reading Integers

To read an integer, the system call service is 5. The read value is returned in register `$v0`. Thus, to read an integer, the following instructions are needed:

```
li $v0, 5
syscall
```

Accessing Variables

You may access local variables by loading them from an offset of the frame pointer (`$fp`). As an example, assuming that the variable `a` is assigned the second 4 bytes of local space. The following code might be generated to access `a`:

```
sub $s0, $fp, 4
lw $s1, 0($s0)
```

Loading a global variable is similar except that we use `$gp` instead of `$fp`. If `a` is a global variable that is store 8 bytes (in the positive direction) off of `$gp`, the following code might be generated to access it:

```
add $s0, $gp, 8
lw $s1, 0($s0)
```

Integer Arithmetic Expressions

In Mips assembly, all operations are done on registers. The best way to generate code is to store all intermediate values in Mips registers. Using the registers `$s0, ..., $s7,$t0, ..., $t9` should be sufficient. You should not need any other temporary registers. For an operation, the operands should all be put into registers, a result register should be allocated, the operations should be performed and then the input registers should be released to be reused later. As an example, the statement

```
write(a+b);
```

might result in the code (if `a` is the first declared variable and `b` is the second)

```
lw $s1, 0($fp)
sub $s0, $fp, 4
lw $s2, 0($s0)
add $s3, $s1, $s2
move $a0, $s3
li $v0, 1
syscall
```

Logic Expressions

Logic expressions are similar to arithmetic expressions. For the mips, the value for `false` is 0 and the value for `true` is 1.

Storing Integer Variables

To store a value in a variable, first compute the address and then store the value into that location. For example, the statement

```
a = 5;
```

could be implemented with

```
li $s0, 5
sub $s1, $fp, 4
sw $s0, 0($s1)
```

Exiting the Program

The `exit` statement in Cminus can be implemented in Mips assembly as follows:

```
li $v0, 10
syscall
```

These instructions call the system routine that exits a program. Every `main` routine in a Cminus program will end in an `exit` statement.

Requirements

Write all of your code in C or C++ . It will be tested on a CS machine and MUST work there. You will receive no special consideration for programs which "work" elsewhere, but not on a CS machine.

Input. The file `CminusProject2.tgz` contains the parser need to begin this project. You will need to modifiy the actions in the project file `parser/CminusParser.y` to do this project. Currently, the actions just emit the rules that are reduced. Sample input for this project is provided in the project directory `input`. To run your compiler, use the command

```
cmc <file>.cm
```

To execute the resulting assembly file, use the Mars simulator(http://courses.missouristate.edu/KenVollmar/mars/).

Submission. Your code should be well-documented. You will submit all of your files, by tarring up your project directory using the command

```
tar -czf CminusProject2.tgz CminusProject2
```

Submit the file `CminusProject2.tgz` via the CS4121 Canvas page. Make sure you do a 'make clean' of your directory before executing the tar command. This will remove all of the '.o' files and make your tar file much smaller.

Data Structures and Documentation I have provided several C data structures for those who will be programming in C. There are doubly linked list, symbol table and string manipulation routines in the workspace directory `CminusProject2/util`. The HTML Doxygen documentation for the provided code is in `CminusProject1/Documentation/html/index.html`. You may ask me any questions regarding these routines. You will not likely need any of these structures now, but you may want to familiarize yourself with them. For those coding in C++, you may use STL.

codegen directory I have created the codegen directory where the codegen.*, reg.* and other files are store. You will add fuctions to generate instructions for different productions in parser and these functions are in codegen.c. In addition, the register allocation management untily implimenation files are reg.c and reg.h, which makes you use registers simple. Before generatging the compiler, you need issue make command under the codegen direcetory.

Makefile Structure The Makefiles for the project are set up to automatically generate make dependences. In a particular directory (*e.g.*, `parser`), you may add new files for compilation by adding the source file name to the `SRCS` variable declaration on the first line of that directory's `Makefile`. For example, to add the file `newfile.c` to be compiled in the `parser` directory, change the first line of `parser/Makefile` from

```
SRCS = CminusScanner.c CminusParser.c
```

to

```
SRCS = CminusScanner.c CminusParser.c newfile.c
```

Nothing else needs to be done. Do not add source files to the root directory `CminusProject2` as the make files assume there are no source files in that directory.

If you would like to add your own subdirectory (*e.g.*, `newdir`) to `CminusProject2`, then change the line

```
DIRS = parser util
```

in `CminusProject1/Makefile` to

```
DIRS = parser util newdir
```

and the line

```
LIBS = parser/libparser-g.a util/libutil-g.a
```

in `CminusProject1/Makefile` to

```
LIBS = parser/libparser-g.a util/libutil-g.a newdir/libnewdir-g.a
```

Then, copy `util/Makefile` to `newdir/Makefile`. Finally, change the `SRCS` declaration in `newdir/Makefile` to contain only the source files in that directory and change the line

```
ARCHIVE = libutil$(ENV).a
```

to

```
ARCHIVE = libnewdir$(ENV).a
```

An Example

Given the following Cminus program:

```
    int main () {
    int i,j,k,l;

        write(10+20);
        i=1;  k=3; l=4;
        j = i + k + l;
        write(j);
        exit;
    }
```

it may be implemented with the following Mips assembly.

```
        .data
.newline: .asciiz "\n"
        .text
        .globl main
main:   nop
        move $fp, $sp
        sub $sp, $sp, 16
        li $s0, 10
```

```
li $s1, 20
add $s2, $s0, $s1
move $a0, $s2
li $v0, 1
syscall
li $v0, 4
la, $a0, .newline
syscall
li $s1, 1
sw $s1, 0($fp)
sub $s0, $fp, 8
li $s1, 3
sw $s1, 0($s0)
sub $s0, $fp, 12
li $s1, 4
sw $s1, 0($s0)
sub $s0, $fp, 4
lw $s2, 0($fp)
sub $s1, $fp, 8
lw $s3, 0($s1)
add $s1, $s2, $s3
sub $s2, $fp, 12
lw $s3, 0($s2)
add $s2, $s1, $s3
sw $s2, 0($s0)
sub $s0, $fp, 4
lw $s1, 0($s0)
move $a0, $s1
li $v0, 1
syscall
li $v0, 4
la, $a0, .newline
syscall
li $v0, 10
syscall
```