CS4121 Cminus Compiler Project 2: An Expression Evaluator

Announced: Friday, Jan. 31, 2020
Due Date: Friday, Feb. 14, 2020 at *5:00pm*

Purpose

The purpose of this project is to gain experience in giving meaning to a programming language by generating x86-64 assembly for a subset of Cminus. Specifically, you will be generating assembly for string output operations, integer I/O operations, integer arithmetic and logical expressions and assignment statements. Completely read this document and the Cminus language specification.

Project Summary

In this project, you will add/change actions to the provided parser that will do the following.

1. Assign space for global 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 logical expressions.
8. Generate assembly to assign values to integer variables.

Prologue and Epilogue Code

Since you are converting a Cminus program to x86-64 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:

```
      .section        .rodata
.int_wformat: .string "%d\n"
.str_wformat: .string "%s\n"
.int_rformat: .string "%d"
        .text
        .globl main
        .type main,@function
main:   nop
        pushq %rbp
        movq %rsp, %rbp
```

This code declares a data section with the read and write format strings, followed by a text section (instructions) containing a declaration of the main routine and instruction to set up the frame pointer (`%rbp`). Each x86-64 assembly file should begin with this sequence. If you assign space in the static data area for global variables, then the space may be allocated with directives after the “.section’ directive and before the “.text” directive.

To end an x86-64 assembly routine, add the code

```
        leave
        ret
```

These instructions exit a program.

Assigning Variable Space

Memory for global variables declared in a Cminus program may be allocated in the static area. The Cminus declarations

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

require 12 bytes of space. We can add the following x86-64 declaration to the data section

```
.comm _gp, 12, 4
```

Variables may be addressed as a positive offset off of the global pointer, _gp. The first variable is located at the global pointer with offset 0, and each successive variable is located 4 bytes from that point. For example, one may load the value of `j` above with the following instructions:

```
movq $_gp,%rbx
addq $4, %rbx
movl (%rbx), %eax
```

String Constants

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

```
write("Hello");
```

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

```
.string_const0: .string "Hello"
```

The label `.string_const0` is implementation dependent. You may name your string constants however you wish.

Printing Strings

Printing strings requires using the C library function `printf`. As an example, the code to implement the `write` statement in the previous section would be:

```
movl $.string_const0, %ebx
movl %ebx, %esi
movl $.str_wformat, %edi
movl $0, %eax
call printf
```

Printing Integers

Printing integers is similar to printing strings except that the actual integer is passed to `printf` rather than an address. As an example, to implement the statement:

```
write(7);
```

the following x86-64 assembly would need to be generated:

```
movl $7, %ebx
movl %ebx, %esi
movl $.int_wformat, %edi
movl $0, %eax
call printf
```

Reading Integers

To read an integer, use the C library routine `scanf`. To read the integer variable `j` declared above, the following instructions are needed:

```
movq $_gp,%rbx
addq $4, %rbx
movq %rbx, %rsi
movl $.int_rformat, %edi
movl $0, %eax
call scanf
```

Integer Arithmetic Expressions

In x86-64 assembly, operations can be done on registers. The best way to generate code is to store all intermediate values in x86-64 registers. Using the registers `%eax, %ebx, %ecx, %edx, %esi, %edi, %r8d, %r9d, %r10d, %r11d, %r12d, %r13d, %r14d` and `%r15d` is sufficient. You should not need any other temporary memory locations. Note that you will need to expand registers to their 64-bit correspondences for address calculation. For an operation, the operands should all be put into registers, the left operand should be the result register, the operations should be performed and then the right operand register 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)

```
movq $_gp,%rbx
addq $0, %rbx
movl (%rbx), %ecx
movq $_gp,%rbx
addq $4, %rbx
movl (%rbx), %edx
addl %edx, %ecx
movl %ecx, %esi
movl $.int_wformat, %edi
movl $0, %eax
call printf
```

Note that integer division is not as straightforward. See the notes from class on x86-64 assembly to figure out how to do integer division.

Logical Expressions

Logic expressions are similar to arithmetic expressions. For the x86-64, the value for `false` is 0 and the value for `true` is 1. You can use the bit-wise x86-64 logical instructions, `and`, `or`, `xor`, and `not` to implement logical expressions.

There are multiple ways to implement relational expressions. 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)

```
movq $_gp,%rbx
addq $0, %rbx
movl (%rbx), %ecx
movq $_gp,%rbx
addq $4, %rbx
movl (%rbx), %edx
cmpl %edx, %ecx
movl $0, %ecx
movl $1, %ebx
cmovg %ebx, %ecx
movl %ecx, %esi
movl $.int_wformat, %edi
movl $0, %eax
call printf
```

Note that instruction `cmpl %edx, %ecx` compares the values of the two registers and update the flags. Later, `cmovg %ebx, %ecx` will copy `%ebx` to `%ecx` if the flag is "greater than". Otherwise `%ecx` remains as 0. The other conditional move instructions you might need are `cmove, cmovne, cmovge, cmovl, cmovle` where `e` means equal, `g` greater than, and `l` less than.

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

```
movq $_gp,%rbx
addq $0, %rbx
movl $5, %ecx
movl %ecx, (%rbx)
```

Requirements

Write all of your code in C or C++. It will be tested on CS lab machines (Rekhi 112, 112a, or 117) and MUST work there. You will receive no special consideration for programs which "work" elsewhere.

Input. I have provided my solution to Project 1 as a Makefile project in `CminusProject2.tgz`. Sample input is provided in the directory `CminusProject2/input`. I will go over my code at a scheduled time for those who wish to use it. To run your compiler, use the command

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

which will output to `<file>.s`
To create an executable, run the following command on the assembly file

```
gcc -o <file> <file>.s
```

You may then directly run the executable.

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

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

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

An Example

Given the following Cminus program (3.add.cm):

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

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

it may be implemented with the following x86-64 assembly.

```
        .section        .rodata
.int_wformat: .string "%d\n"
.str_wformat: .string "%s\n"
.int_rformat: .string "%d"
        .comm _gp, 16, 4
        .text
        .globl main
        .type main,@function
main:        nop
        pushq %rbp
        movq %rsp, %rbp
        movl $10, %ebx
        movl $20, %ecx
        addl %ecx, %ebx
        movl %ebx, %esi
        movl $.int_wformat, %edi
        movl $0, %eax
        call printf
        movq $_gp,%rbx
        addq $0, %rbx
        movl $1, %ecx
        movl %ecx, (%rbx)
        movq $_gp,%rbx
        addq $8, %rbx
        movl $3, %ecx
        movl %ecx, (%rbx)
        movq $_gp,%rbx
        addq $12, %rbx
        movl $4, %ecx
        movl %ecx, (%rbx)
```

```
movq $_gp,%rbx
addq $4, %rbx
movq $_gp,%rcx
addq $0, %rcx
movl (%rcx), %r8d
movq $_gp,%rcx
addq $8, %rcx
movl (%rcx), %r9d
addl %r9d, %r8d
movq $_gp,%rcx
addq $12, %rcx
movl (%rcx), %r9d
addl %r9d, %r8d
movl %r8d, (%rbx)
movq $_gp,%rbx
addq $4, %rbx
movl (%rbx), %ecx
movl %ecx, %esi
movl $.int_wformat, %edi
movl $0, %eax
call printf
leave
ret
```

We will go through this example in class.