TDT4205 Problem Set 5

Answers are to be submitted via Blackboard by April 3^{rd}.

1 Assembly programming

The lyrics of the counting song *"10 Green Bottles"* follow the pattern

10 green bottles hanging on the wall,
10 green bottles hanging on the wall,
and if 1 green bottle should accidentally fall,
there'd be 9 green bottles hanging on the wall.

It has a total of ten similar verses which count down the total, and ends when the last line reaches 0. Write an x86_64 assembly program which prints the full lyrics to this song.

2 Code Generation Part I

Having created a symbol table, you are now ready to start the fun part where we get an executable result, the code generation. For the first part, you will implement most functionality except for control structures, which is enough to compile and run the example programs in the ps5 folder. This cheatsheet should help you on the way.

2.1 String table

In the previous assignment, you extracted the strings into `string_list`. Now, finish the `generate_string_table` function by writing out symbol names and static data declaration. Make sure to select easily retrievable symbol names, e.g. `.STR`*idx*.

```
.data
.intout:    .asciz  "%ld "
.strout:    .asciz  "%s "
.errout:    .asciz  "Wrong number of arguments"
.STR0:      .asciz  "First string"
.STR1:      .asciz  "Hello, World!"
```

2.2 Global variables

Next, write out declarations of all global variables. This is done in almost the same way as the string table, except that the variables are uninitialized and thus should be placed in a `.bss` (block starting symbol) section. Since all variables are 64 bit, we align them by adding the keyword `.align 8`. After you have implemented the function `generate_global_variables`, the complete table should look something like this:

```
.bss
.align 8
.global_var0:
.global_var1:
```

2.3 Functions

The last task in global scope is to add functions. Entering a function has some common tasks that you will implement in the function `generate_function`. As with above, we use the function's label to assign the entry point. Program code is placed in the `.text` section, and also want to make the names global using the line '`.global func_name`'.

Next, save the caller's base pointer and stack pointer (which will be the callee's base pointer).

Last thing before starting the function code is pushing the arguments to the stack. Remember that the stack needs to be 16 bytes aligned. We can solve this by subtracting 8 from the stack pointer after pushing arguments if there is an odd number of arguments.

A function entry should look something like this:

```
// func my_func (a, b, c)
.text
.global _my_func
my_func:
    pushq   %rbp        // Save SP and BP
    movq    %rsp, %rbp
    pushq   %rdi        // Push arguments to stack
    pushq   %rsi
    pushq   %rdx
    subq    $8, %rsp    // Align, there were 3 args
```

2.4 Function body

With all global names set, we are ready to generate our functions.

2.4.1 Expressions

To keep it simple and avoid the problem of register allocation, we are going to treat our machine as a stack machine. When computing expressions, compute them one at the time using the `%rax` register, and push the intermediate result to the stack. Then, when it is needed you pop it back off the stack.

```
a := 3 * (2 + b)
```

looks something like this:

```
movq    $3, %rax        // Load 3
pushq   %rax            // push l.h. of multiplication
movq    $2, %rax        // Load 2
pushq   %rax            // push l.h. of addition
movq    ._b, %rax       // Given b is a global variable
popq    %r10            // pop l.h. (%r10 arbitrarily chosen)
addq    %r10, %rax      // Intermediate result 2 + b
popq    %r10            // pop l.h.
addq    %r10, %rax      // Intermediate result 3*(2+b)
movq    %rax, -0(%rbp)  // Given a is the first parameter
```

2.4.2 Function calls

Implement function calls. When calling a function, the first six arguments are passed by registers. The skeleton code provides an array `record`, containing these registers in correct order. Any subsequent arguments are pushed to the stack. Before that though, you need to push any caller-saved registers

Print statements can be translated into a sequence of `printf` system calls, with one call per item in the PRINT statement's list. Use the provided format strings, `strout` and `intout` to properly format your print item. Pass the format string to `%rdi` and the value to print to `%rsi`.

You should now be able to compile `helloworld.vsl` from PS 2.

2.4.3 Loading and storing

When an expression is resolved, its final value will be stored in the `%rax` register. Now save it to the location of the symbol that is being assigned to. For global variables we simply reference them by name. For parameters and local variables, we need to calculate their address using the sequence number. Parameters start at the base pointer, any parameters beyond 6 are store *above* the base pointer, and local variables follow after the up to 6 first parameters. The table below shows what the stack would look like for a function with 8 parameters.

seq	name
6	parameter 7
7	parameter 8
-	base pointer
0	parameter 1
1	parameter 2
.	...
5	parameter 6
8	local 1
9	local 2

2.4.4 Return statements

By convention, the return value is stored in the `%rax` register, which is where we already have put the result that we want to return after resolving the return expression. What's left is to restore the stack and base pointers and call return.

```
movq    %rbp, %rsp
popq    %rbp
ret
```

Note: VSL requires all functions to return a value, but most of the example programs handed out earlier did not comply with this (they do now). If we don't want to require a return statement, e.g. in our main function, the `generate_function` function can be extended after generating the function body to always move a zero to `%rax` and run the return code from above. Now, if a function doesn't return anything we make sure to properly exit it.