Specification for Lab assignment 3

Programming in the UNIX environment

Created by Stefan Axelsson
Modified by Florian Westphal
Department of Computer Science
Blekinge Institute of Technology

2019-10-14

1 Your task

One of the only remaining legitimate reasons for learning to program in assembly is to write a compiler that translates instructions in some higher level language to machine code for the target architecture. In this lab you will do just that, but in order for you to not have to spend time learning compiler construction, the compiler proper will be given to you and you only have to change the current backend that emits code for a hypothetical stack machine to one that emits assembly code for the x86-64 architecture on Ubuntu Linux. This lab requires very little code from you, but a great deal of understanding.

2 Background

2.1 The Provided Compiler

The compiler compiles code in a very simple calculator language to assembly pseudo code. In order to build the compiler, you should follow the instructions in the file *'build'*. For this assignment, only *'calc3b.exe'* is important[1].
The basic idea of this assignment is that you rename the file *'calc3b.c'* into *'calc3i.c'* and change the print statements to one or more print statements that emit the x86-64 instructions that you want instead. Note that if you use a register in one of the case blocks you have to make sure that you either do not need it at the end, or that it is saved (and restored) before you exit (or make a recursive call to *'ex'*). You probably do not have to save and restore registers however.

2.2 The Language

The calculator language is equally simple; it doesn't contain any instructions for input and only one for output. There are variables that can be assigned to and read from, but they are restricted to single letter identifiers that are the lower letters from a-z, thus one cannot have more than 26 different variables[2].

[1] If you are interested in what the other compiler/interpreter versions do, try them out or refer to the Lex and Yacc tutorial ("A COMPACT GUIDE TO LEX & YACC" by Tom Niemann (`http://www.epaperpress.com/lexandyacc/`))

[2] This makes handling the symbol table very easy as one can just allocate 26 memory positions and index into that array given the single letter identifier (i.e. 'a' is the first position, 'b' the second, and so on), so when the compiler sees the variable 'b' it can just store or fetch the data at the second position in the array (doesn't have to be an actual array, labels are fine).

An example of a program in the calculator language:

```
a=732;
b=2684;
while(a != b) {
        if(a > b) {
                a=a-b;
        } else {
                b=b-a;
        }
}
print a;
```

This program calculates the greatest common divisor between 732 and 2684 prints the result (244 in this case), by using Euclid's original algorithm[3].
The given stack based compiler translates this to:

```
        push        732
        pop         a
        push        2684
        pop         b
L000:
        push        a
        push        b
        compNE
        jz          L001
        push        a
        push        b
        compGT
        jz          L002
        push        a
        push        b
        sub
        pop         a
        jmp         L003
L002:
        push        b
        push        a
        sub
        pop         b
L003:
        jmp         L000
L001:
        push        a
        print
```

As you can see the stack language is equally simple. The output consists of *'push/pop'* instructions that either push the value in variable a onto the stack or pop the value off the top of the stack and stores it in the variable.
Operators are equally simple; arithmetic operators like *'sub'* above pop two values off the stack, subtract them and push the result back on the stack. Thus the statement *'a=a-b'* in the calculator language is converted to; *'push a; push b; sub; pop a'*, which puts the values in variables *'a'* and *'b'* on the stack in the correct order, subtracts them, and stores the result back into variable *'a'*. Since the x86-64 *'sub'* instruction does not take its arguments from the stack and does not put its result onto the stack, you will have to add extra instructions to do so, when handling the substraction case.

[3] `http://en.wikipedia.org/wiki/Euclidean_algorithm`

The control structure consists of comparison instructions *'compGT'* for *'compare greater than'* which sets an invisible true or false flag somewhere in the computer that subsequent *'jz'* - *'jump if zero'* instruction can act on, incidentally the only conditional jump instruction. The compiler proper (in file *calc3b.c* in the included files) can thus do its work by a simple recursive strategy that does not need to take more than the current input token into account, i.e. it only needs to check if the previous comparison instruction returned true or false.

3 Requirements

3.1 Project Structure

The basic folder structure for your project has to look as follows (you may add folders, if necessary):

```
project
    |----- bin/
    |----- lexyacc-code/
    |----- lib/
    |----- src/
    |----- c-driver.sh
    |----- x86-64-driver.sh
    +----- Makefile
```

The `lexyacc-code` folder shall contain all source files for your compiler. Basically, you can just use the provided folder and add your *calc3i.c* (and *calc3c.c* - cf. Section 3.5) file. You can either create build artefacts in this folder as well, which is not so nice, but permitted, or you could create a dedicated folder for the build artefacts.
The `src` folder should contain the source files for your external library (cf. Section 3.3). As with the compiler code, you can decide where the build artefacts will be created.
Last but not least, it must be possible to build your compiler as well as the external library by issuing the `make` command within the `project` folder, which will place the compiler executable(s) in the `bin` folder, while the library should be placed into the `lib` folder.

3.2 Basic Compiler

As described above, your task is to change *calc3b.c* into *calc3i.c* that instead of emitting the pseudo assembly code emits actual x86-64 assembler instructions (that uses the x86-64 stack). Your compiler should emit code that handles 64 bit signed integers.
Since the provided compiler only produces the instructions required to translate the calc language code, it does not by itself create an assembly program, which can be compiled into an executable. This is the case, since it does not define data and text segments as well as the symbol table, which you need in order to handle variables. Furthermore, the compiler also does not call the exit function/system call to terminate the produced program.
In order to produce an assembly program, which can be compiled into an executable, you will have to write a shell script, called *'x86-64-driver.sh'*, which takes as input a file with the *'.calc'* ending and then:

1. writes the required prologue (e.g. define data and text segment, define the symbol table) into a new file with the ending *'.s'*
2. appends this file with the output of your compiler (i.e., *calc3i.exe*) for the given *'.calc'* file
3. appends an epilogue (e.g. for calling the exit function/system call)

After that your driver should call *'gcc'* (or *'as'* and *'ld'* separately) to assemble and link the assembly file to produce an executable.
For example, when I run your shell script as follows: *'x86-64-driver.sh bcd.calc'*, I expect as output a file called *'bcd.s'*, which contains the produced x86-64 assembler code for the file *'bcd.calc'*, as well as a file called *'bcd'*, which is an executable program, which does what was written in *'bcd.calc'*.

3.3 Additional Functions

The lab requires you to implement support for three new "instructions" that are not in the stack machine described in the original Lex and Yacc tutorial. These must be implemented as function calls, that is your compiler should emit a call to a library function whenever it handles one of these new instruction cases. You will of course also have to supply the implementation (in assembly language) of the routines that calculate the results of these new instructions.
If you want consideration for higher marks you have to implement these as a proper external library (compiled into a *'.so'* or *'.a'* file), emit proper calls and link your resulting binary with the library containing the implementation (cf. Section 5). Otherwise, it is OK to just include an implementation for each of the functions in the prologue that you emit anyway as part of your program. The three new instructions are:

1. fact - Take one argument and return the factorial of that argument. For example: 0! = 1.
2. lntwo - Take one argument and return the binary logarithm of that argument. For example: lntwo 32 = 5.
3. gcd - Take two arguments and return the greatest common divisor between the two arguments. For example: 36 gcd 24 = 12.

You are free to chose whichever sensible implementations you wish for these (and other) algorithms.

3.4 Output Handling

The assembler code produced by your compiler has to be able to print values to standard out. While you can use the C-libraries to handle output, e.g., by calling *'printf'*, you should use system calls instead, e.g., *'write'*, if you want consideration for a higher grade (cf. Section 5).

3.5 C Stack Machine

If you want consideration for the highest mark for this assignment, you should also write a backend that produces 'C' code, instead of x86-64 assembly code, and handles that much the same it would the assembly code (cf. Section 5). Write a driver called *'c-driver.sh'* and call the compiler *'calc3.c.c'*. Note that your compiler should emit code much as the previous one for a low level 'C' stack machine that you should provide an implementation for in your prologue (i.e. just "translating" the calc code to 'C' is not acceptable, though one can actually do that.) For the implementation of the 'C' stack machine, it should be sufficient to provide a simple stack implementation. Then, the 'C' code generated by your compiler can push and pop values to this stack. You do not have to provide an assembly language implementation for the extra three functions in this case but can just call an implementation in 'C' or an already existing 'C' library function.

4 Presentation

The assignment should be presented in person with both group members in attendance on a lab slot. You will be required to show the code, let me run it and be prepared to discuss your work. You can use the provided test programs to make sure that your compiler works before you submit it.

5 Grading

You should demonstrate a clear understanding of the problem and detail your strategy of how to solve it, including drawbacks and advantages of the solution chosen. Your code should be clear, concise and to the point.

You are allowed to discuss problems and solutions with other groups, but the code that you write must be your own and cannot be copied, gleaned, downloaded from the internet etc. (You may of course copy idioms and snippets. However, you must be able to clearly describe each part of your implementation.

The following list details, which of the requirements, stated in Section 3, have to be implemented to get a certain grade.

Grade - D

- Implement the compiler for the basic stack machine (cf. Section 3.2)
- Implement one of the additional functions (cf. Section 3.3)

Grade - C

- All requirements for Grade D
- Implement the remaining two additional functions (cf. Section 3.3)
- Implement all three additional functions as proper external library (cf. Section 3.3)

Grade - B

- All requirements for Grade C
- Use system calls instead of library functions to handle output (cf. Section 3.4)

Grade - A

- All requirements for Grade B
- Implement the compiler for the C stack machine (cf. Section 3.5)

Note: Even if your implementation fulfils the requirements for the grade you were targeting, you might get a lower grade (lower by one), if your code is of bad quality. For this reason, there are no requirements for grade E. You can only get an E, if your implementation fulfils the requirements for grade D and your code is of bad quality. In this way, it is impossible to fail the assignment just because of poor code quality.

6 Additional Information

Useful resources for this assignment are:

- "Programming from the Ground Up Book" by Jonathan Bartlett (`http://savannah.nongnu.org/projects/pgubook`) - The examples are in AT&T syntax, but only for x86 assembly

- "x86-64 Assembly Language Programming with Ubuntu" by Ed Jorgensen (`http://www.egr.unlv.edu/~ed/x86.html`) - The examples are for x86-64 assembly, but only in Intel syntax
- "Intel® 64 and IA-32 Architectures Software Developer Manuals" (`https://software.intel.com/en-us/articles/intel-sdm`)

Apart from the bundle of source files for the compiler that you are given there are also a small number of testfiles with the required results in the README.txt file that is included.
You are encouraged to download all the files and start playing around with them on the supplied test programs so that you get a feel for what the supplied compiler does, maybe make a few changes to the printf statements to understand what goes where.

Good luck, and may the source be with you.

Typeset by LaTeX.