Specification for Lab assignment 3

Programming in the UNIX environment

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

2017-09-27

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 basic instructions for a stack machine. A stack machine is defined by not using any registers, i.e. all operands for instructions are popped off the stack and the result is returned to the stack. Thus almost all of the instructions themselves do not have to reference any registers or memory locations as both the input and output vectors are the stack per default. This simplifies matters as the compiler doesn't have to do any register allocation.
The compiler is described in the Lex and Yacc tutorial[1] on the lab page (there are a few additions to the lab that aren't in the tutorial though). However, you do not actually have to understand how Lex and Yacc works, follow the instructions in the file *'build'* to build the different versions of the compiler/interpreter. The description of the calculator proper begins on page 19. There is no documentation of the stack language of the compiler other than the C- code in *calc3b.c* but that should be enough as there really are not that many instructions, you will get away with changing each of 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 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.

[1] "A COMPACT GUIDE TO LEX & YACC" by Tom Niemann (`http://www.epaperpress.com/lexandyacc/`)

2.2 The Languages

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]. 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 Euclides 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

[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).

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

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'*. 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.

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 folder from it's learning 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 Stack Machine

Your task is to change *calc3b.c* into *calc3i.c* that instead of emitting *'push a; push b; sub; pop a'* emits reasonable x86-64 assembler instructions (that uses the x86-64 stack). You will need to allocate space for the symbol table and emit some header for the assembly then the instructions that make up the code and then a few lines to clean up, so you should write a shell script driver that emits a prologue then runs the compiler and appends the assembly output, then appends the epilogue to an assembly file. After that your driver should call *'gcc'* (or *'as'* and *'ld'* separately) to assemble and link the assembly file to produce runnable code. The shell script should be called *x86-64-driver.sh* and take as input a file named with the *'.calc'* ending and produce a runnable object file with the same name but without the *'.calc'* ending, i.e. *'x86-64-driver.sh bcd.calc'* should produce the runnable file *'bcd'* it should also produce the assembly file but with the *'.s'* ending, e.g. *'bcd.s'*.
Your compiler should emit code that handles 64 bit signed integers.

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 comes across one of these new instructions. 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, 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 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

The book on Linux assembly programming[4] contains all that you need to know, but its instruction tables are not complete, look in the Intel documentation[5] for that, there is at least one I can think of that makes your work a little easier. 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.

[4] "Programming from the Ground Up Book" by Jonathan Bartlett (`http://savannah.nongnu.org/projects/pgubook`)

[5] Basic Architecture/Instruction Set Reference (`http://www.intel.com/products/processor/manuals/index.htm`)