Module 2
Code Generation and Runtime Environments

Compilers and Execution Environments (ID2202)
7.5 hp

David Broman
KTH Royal Institute of Technology
Monday 22nd November, 2021

Abstract

Welcome to Module 2 of the course! In this module you will learn how to code x86 assembly, how linkers and loaders work, and how to perform instruction selection. The goal of Module 2 is that you have a complete (unoptimized) compiler for the Cigrid language: from source code, all the way down to x86 machine code. Note that all examples in this module assumes that they are compiled and executed under Linux, using the Vagrant file supplied in the course material.

This document outlines the different exercises for Module 2. The exercises are divided into three levels: Satisfactory (S), Good (G), and Very Good (VG). Depending on your goal of the final grade in the course, you may choose which exercises that you perform. Please note that the minimum for passing the course is to complete all exercises at level Satisfactory (S). To reach level (G) for this module, all exercises at level (S) and (G) have to be completed. Finally, to achieve level (VG), you need to finish all exercises at all three levels. For details about the grading, see the course memo on Canvas.

Contents

1 Level: Satisfactory (S):

The goals of the tasks at level Satisfactory (S) are to (i) learn the basics of x86 assembly, and (ii) to compile a very small subset of Cigrid down to x86 assembly.

1.1 x86 assembly programming

In this first exercise, you will learn the basics of x86 assembly programming. This is the only time that you will write assembly programs explicitly. In the rest of the course, your compiler will generate assembly code for you!

Recommended Reading

Before you start, we recommend that you

- go through lecture 7 in depth,
- check out the links about assembly programming on the Canvas website (end of the page "Literature and Resources"), and
- Check out the example files `lecture-asm-gcc.zip`, available on the Canvas page "Assignment Tasks and Workshop".

Getting started

This exercise assumes that you have installed Vagrant and that you are compiling all the programs using the Vagrant setup outlined on the page "Vagrant" on the Canvas course website. The reason for using Vagrant in the course is that it is important to use the same operating system and the same assembler (there are slight differences between Windows, Mac, and Linux). [1]

Under the folder `/template-projects` there is a folder `/asm` that contains template files for this task. Copy the files under `/template-projects/asm/` to `/solutions/module2/asm/` (you need to create this new folder). Add these files to your Git repository and push the changes.

Within folder `/solutions/module2/asm/`, run `make`. Two executables are now created: `main-c` and `main-asm`. File `main-c` is based on the files `factorial.c` and `main-c.c`. This C implementation shows the correct behavior of the program. Right now, the file `main-asm.asm` contains a hello world x86-64 assembly program. The overall task is to implement your solution in file `main-asm.asm` so that it gets the same behavior as when executing `main-c`.

Sub tasks

We strongly recommend that you implement your solution in steps, and check the solution in each step. Hence, we have divided the main task into several sub tasks:

1. Implement the menu function, so that the program prints out the menu if less than 3 arguments are given to the program. See the C file for correct behavior. Note that you should not call `puts` more than once. Recall that you can use command `echo $?` to make sure that you return the correct error code.

[1] NOTE: If you are using a new Mac with an Apple M1 chip, you need to contact the course team. The M1 chip is using the ARM instruction set infrastructure, whereas the exercises in this course require an x86 compatible processor. We can provide a solution to this problem, but you need to contact us in such a case.

2. Implement the echo command `e`. Note that pointers in x86-64 assembly are 64 bits (8 bytes). The return code when executing this command should be 0.

3. Implement the magic number command that prints out the magic number `-126`. This means that you need to implement function `print_int` in assembly code. The function should follow the semantics, as implemented in file `main-c.c`. Note that it is not allowed to call any C libraries for printing the decimal number. The only function that is allowed to call is `putchar` (see the C standard library documentation). Advice: Try to implement positive numbers first, and then add negative numbers afterwards. Also, try to implement small parts of the program and test every step. Try out cases such as 0, -1, -1231, 7, 22, 921 by changing the magic number.

4. Implement the `+` command. You should call the standard C function `atoi` when you convert a string into a signed integer value.

5. Implement command `!` which should call the external function `factorial_message`, which is already implemented in file `factorial.c`. Note that you need to update the `Makefile` so that (i) the factorial file is separately compiled into an individual object file, and (ii) that you also link with this file. Note that you need to declare the function as external in the ASM file. Note also that you can use the UNIX command `man` to learn more about different commands.

To pass all the automatic grading tests, all the above must work. However, we encourage you to implement it in steps and that you run the autograder on each of the subtasks.

Grading tag

Use the following tag when submitting this exercise: `#module2-asm-s`

1.2 Instruction selection and compiling basic Cigrid to x86-64

In this exercise, you will learn how to do basic instruction selection of straight line code, which does not contain any branches.

Recommended Reading

Before you start, we recommend that you

- listen through Lecture 8 in detail, and
- read Chapter 11 of the course book, with focus on Sections 11.4 and 11.5.

Task

The overall task is to perform limited instruction selection on one basic block. This means that your compiler should be able to generate x86-64 assembly code for a Cigrid program that only contains a `main` function, with the following limitations:

- The main function only needs to handle straight line code, where there are no branches (e.g., `if` or `while` statements).
- The `return` statement must at least support integers constant (e.g., `return 1;`) and return using a variable (e.g., `return x;`).

- Definitions of local integer variables (e.g., `int x = 8`) and local assignments (e.g., `x = 8`).
- The following expressions must be supported: binary addition (e.g., `x + 3`), integer decimal constants (e.g., `123`), and variables (e.g., `x`).

For instance, the following program should be possible to compile:

```
int main(){
  int x = 1;
  int y = 5;
  x = x + 4;
  x = x + y;
  return x;
}
```

If parameter `--asm` is supplied, your Cigrid compiler should pretty print the generated assembly code to standard output. For instance, if the Cigrid program above is compiled, the following assembly program may be sent to the standard output:

```
        global  main

        section .text
                                ; function 'main'
main:
        sub     rsp, 16
        mov     qword [rsp], 1
        mov     qword [rsp + 8], 5
        add     qword [rsp], 4
        mov     r10, qword [rsp + 8]
        add     qword [rsp], r10
        mov     rax, qword [rsp]
        add     rsp, 16
        ret
```

Note that your compiler does not have to produce exactly the same assembly code. However, if the produced assembly code is assembled, linked, and executed, it must give the same result.

A few observations and requirements regarding the code generation:

- For level S, test output data is generated via the program's return code (the `return` statement in a `main` function).
- If the generated assembly code is redirected and written into a file, e.g., using command `cigrid --asm test.cpp > a.asm` then, this file should be possible to assemble `nasm -felf64 a.asm`, linked `gcc -no-pie a.o`, and executed `./a.out`.
- You may add an optional flag `--compile` which is performing all these steps for you (good for testing). However, the automatic grading system will only use flag `--asm` to output and then assemble and link this assembly code.
- For S level, you can assume that there will always be a `main` function, so you may always add the `global main` directive.

- It is allowed to always use 64-bit registers for integer variables.

Some advice:

- Start by creating the pretty printer of the assembly code. Test it by just constructing some data objects that you pretty print separately from the rest of the code. Use escape character `\t` to get nice indentation.
- Construct data types for IR code as described in Lecture 8, even if you will not create any control-flow graph at level S. This will simplify your extensions, given that you decide to continue with level G.
- Do not try to implement for S, G and VG at the same time. Try to complete S first, before you continue with G.
- Add a pretty printer for the IR. You can add a flag `--ir` for pretty printing the IR. This is very helpful when you test the program. However, the autograder will not use this command line option.
- Create the instruction selection, going from IR to the assembly data types. Do not try to allocate real registers directly, but use a counter to represent all virtual registers. Check that the output assembly code looks reasonable.
- Add code for generating spilling of all virtual registers. This phase is easiest to do after instruction selection, by transforming the abstract syntax of the assembly code.

Grading tag

Note that you should extend your current Cigrid compiler in folder `solutions/cigrid/`. Use the following tag when submitting this exercise: `#module2-cigrid-s`

A note on the examination of Module 2 and Module 3

The topics of Linkers, Loaders, Garbage Collection, and JIT are presented as part of Module 2 (lectures 7 and 9), but the content will not be examined until in the theory exercises in Module 3. The reason for this is to load balance the workload between the different modules. In Module 3, the theory exercises at level S will include topics from all three modules. However, level S at Module 3 will not contain any more development of the Cigrid compiler. Hence, if you aim at level S in Module 3, your final work on the Cigrid compiler will be here in Module 2. However, levels G and VG in Module 3 contain further improvements of the compiler, to optimize it and to make it faster!

2 Level: Good (G):

The goals of the tasks at level Good (G) are to understand how to compile programs using control-flow graphs (CFGs).

2.1 Compiling with Control-Flow Graphs

Your task is to extend your compiler to also support the following language constructs:

- Binary expressions `+`, `-`, `*`, `/`, `%`, `<`, `>`, `<=`, `>=`, `==`, `!=`.
- Statements `if` (including `if` followed by `else`) and `while` loops.
- Possibility to call and return from external functions, including parameter passing (up to 6 arguments).
- Support for character constants.

Constructs that are explicitly *not* needed for level G are:

- Definition of user defined functions.
- Unary operators.
- Logical and bit-manipulating binary operators.
- Arrays and pointers.
- Strings.
- The `break` statement.
- Heap memory allocation, including `new` and `delete`.
- The `struct` construct.
- Global variables.

Note also that your program must follow the AMD64 ABI calling convention.

Some advice:

- You might need to use the assembly instruction `movsx` when sign extending a byte (for the `char` data type) to a 64-bit registers.
- When using labels in NASM, you can run into the situation where you get a name conflict with another NASM keyword, such as a predefined instruction name. In such a case, you can prepend the label name with a `$` character. For instance, if you have an external function called `foo`, you may call the function using syntax `call $foo`.

Grading tag

Use the following tag when submitting this exercise: `#module2-cigrid-s-g`. It will check that the Cigrid implementation still works for level (S).

3 Level: Very Good (VG):

The goal of this task is to get complete understanding of the toolchain, including memory management.

3.1 Finalize the complete Cigrid compiler

In this task, you should finalize and create a complete Cigrid compiler, including all the parts that were missing from the previous exercises. That is, any correct Cigrid program should be possible to compile, link, and execute. Note, however, that some parts are optional, and will not be tested in the automatic grading system (see below). The program should be correct, but not optimized. Key challenges in this task include the support of user-defined functions, memory management, pointers, variables defined within scopes, and global variables. Note also that program arguments using the C style `main` function signature must work. That is, using the following signature: `int main (int argc, char **argv);`

Note that the following constructs are optional, but you may implement them if you want to:

- the `struct` construct, and accessing elements in a struct,
- the `break` statement,
- the support for declaration of variables in local scopes,
- global variables with other types than integers (integer types need to be supported), and
- support for more than six function arguments.

Some advice:

- To be able to support `varargs` functions, that is, functions that can take variable number of arguments, you need to follow the AMD64 ABI. However, this specification is not really clear regarding this matter. For `printf`, it seems to be OK to just clear `rax` before you call `printf`. In this task, you only have to care about `printf` as a `varags` function. Hence, you may have an explicit check in your compiler and clear `rax` in case a user makes a call to `printf`.
- Note that you can use the standard clib functions `malloc` and `free` when you implement the `new` and `delete` operators.

Grading tag

Use the following tag when submitting this exercise: `#module2-cigrid-s-g-vg`