CS244, STELLENBOSCH UNIVERSITY

TUT 3: INTRODUCTION TO 32-BIT INTEL ASSEMBLY LANGUAGE PROGRAMMING

8 OCTOBER 2021

> There are 10 types of people: Those who understand binary, and those who do not.

OVERVIEW

The goal of this tutorial is to get acquainted with writing small functions in assembly language and integrating the resulting object code with existing C programs. You must also familiarise yourself with NASM assembler under Linux. After successfully completing the tutorial, you should be able to:

- understand basic assembly language instructions;
- translate C statements such as assignments and loops to assembly language by hand;
- understand how function results and call-by-value parameters work in C; and
- use the Intel assembly language instructions `cmp`, `dec`, `idiv`, `imul`, `j`*`cc`*, `jmp`, `mov`, `pop`, `push`, `ret`, `sar`, `test`, and `xor`.

A NOTE ON THE ARCHITECTURE

The assembly language tutorials were designed to work in NARGA. They require a Linux installation running on a host that provides the Intel IA-32 instruction set. Because you have access to NARGA, where everything required is already set up, and also, because of the sheer number of possible complications on binary level, ***no assistance will be given with setting up your own computer for assembly language programming.*** If you still want to work on your own computer, please note the following.

1. Even though modern Intel chips are 64-bit, we study 32-bit assembly language.[1] You cannot write 64-bit assembly language for these tutorials—in particular, because of the different call convention used for 32-bit assembly language on Linux.
2. For the same reason, these tutorials cannot be completed on either macOS or Windows. Also, both of these operating systems have binary executable file formats that are incompatible with Linux.
3. If you have a machine with a 64-bit chip – which is the norm today – you have to install the 32-bit compatibility libraries. The makefiles invoke NASM with the `-f elf32` setting, and the C compiler with the `-m32` switch.
4. Any Intel or AMD chip should work, because they binary-compatible for the instructions we use. However, RISC chips – like the ARM chips found in the Raspberry Pi – have a different assembly language, and cannot be used.

[1] You're not being cheated: Because they can be manufactured cheaply, Intel still sells and releases new 32-bit processors for embedded systems, which constitute one of the likely use cases for assembly language today.

INSTRUCTIONS

1. Clone your repository for Tutorial 3 at

   `rw244-2021@gitolite.cs.sun.ac.za:`⟨*US number*⟩`/tut3`

2. The repository contains a number of files, including a makefile and a C program that must be used for testing. The C header file `tut3.h` contains the following three prototypes:

   - `int max(int x, int y);`
   - `int power(int x, int y);`
   - `int gcd(int m, int n);`

   The `max` function is already implemented and should be used as a guideline to complete the two remaining functions. C implementations of all the functions are given in the file `test3.c`; they are called `max_c`, `power_c`, and `gcd_c`, respectively. Complete `power` and `gcd` in assembly language. *Note that you do not need to define any local variables for this tutorial.*

3. You have to complete both `power` and `gcd` for submission via Git. The last commit will be assessed as your attendance assignment for next week.

4. You have to complete your plagiarism declaration in the repository.

5. You MUST code your solutions by hand, and using any tools for disassembly or decompilation is considered cheating. Handwritten assembly looks very different from compiler-generated assembly, and the difference is quite easy to spot. If I have any suspicion as to the provenance of your submission, I reserve the right to call you up for a viva voce in the form of a screenshare on Teams.

6. There are no marks for style, but marks may be subtracted for using local variables and for particularly baroque attempts. Efficiency is key: Think like an assembly language programmer. For example, if you must divide by a power of two, using the `idiv` instruction instead of a shift is not only daft and inefficient, but will also complicate your register assignment.

7. You can obtain information on NASM by executing the command `info nasm` at the command prompt. Links to additional resources are on SunLearn.