CSE1400 Lab Content Manual

Delft University of Technology

Edition 2022-2023[1]

[1]Revision 20220901T2315

Acknowledgements

The CSE1400 lab course assignments were originally developed by Sidney Cadot for the PowerPC architecture. The accompanying documents, two versions of the assignments and a tutorial on PowerPC assembler, were also written by Sidney. Jonne Zutt maintained these documents over the years after Sidney left the university. In 2004, the decision was made to change the target architecture of the lab course from the somewhat obscure PowerPC to the ubiquitous Intel x86 platform. The lab course environment had to change accordingly: the carefully tweaked commercial IDE/PowerPC emulator running on Microsoft Windows was abandoned in favour of the GNU assembler and debugger, which are available on nearly every Linux distribution. Because of these changes, the assignments and the accompanying reference material were rewritten by Denis de Leeuw Duarte.

In 2005 a new curriculum has been started, which transformed the old lab into a more elaborate project for Software Technology students, while Media and Knowledge Technology students only had to do a trimmed down version. The manual was modified to match the new requirements by Bas van der Doorn and Sander Koning.

In 2011 and 2012 the manual was further updated by Mihai Capotă and Alexandru Iosup and in 2013 and 2014 the manual received a refresh and expansion by Otto Visser. In 2014, the decision was made to switch from the x86 to the x86-64 architecture and the manual was edited by Elvan Kula. Maarten Sijm restructured the manual in 2019 and during the Corona (no, not the beer) summer of 2020 further restructuring and editing was done by Sára Juhošová, Taico Aerts and Otto Visser.

During the summer of 2022, this manual was split into two parts: the content manual, dealing with the theoretical concepts, and the assignments manual, providing information about the lab and its assignments. This restructuring was done by Alexandru Postu, with additional help from: Alexandra Marcu, Ana Băltăreţu, Ana Cristiana Marcu, Daniel Peter, Rareș Toader and Yiğit Çolakoğlu.

Contents

Chapter 1

Introduction

Given that this course is given in the first quarter of the first year, it is quite likely that you have zero programming experience. There is however also a reasonably large group amongst you that seems to have some programming experience. For those, a computer was usually "sketched" as a machine that executes program statements in a line-by-line fashion, manipulating variables and performing calculations as it goes along. This simple model allowed you to write your first computer programs in Java, Python, or some other language. Of course, the underlying mechanics of executing programs are a bit more complicated. It is quite likely that you have already encountered the limitations of this naive model in the form of incomprehensible error messages and seemingly inexplicable program failures. Apparently, it is necessary to acquire a more thoroughly detailed mental picture of the machine in order to solve many common programming problems. The goal of this lab course is to fill in these details. Much of this knowledge is to be acquired through practical assignments, but careful reading is also an important part of the educational process.

You should read the remainder of this introductory section as well as all of Chapter 2 before starting on the first lab assignment. Take special notice of Section 1.2, which summarises and refreshes the prior knowledge that we assume on your part.

1.1 Why Assembly (Still) Matters

To round up this introductory chapter, it is time to answer an important and often asked question. Why is it necessary that you learn how to program in assembly? Is Java or C not much more convenient? There are two answers to this question. First and foremost, not all programs can be written in high level languages like C or Java. Contrary to popular lore, new compilers, Virtual Machines and operating system kernels are not passed down to us from the heavens. Instead, and with an equal sense of drama, they have to be forged by the hands of engineers of flesh and blood through long toil and serious hardship. Engineers like *you*. If you, the computer scientists of the future, do not know how to program in assembly language, who will? And who will port our kernels to the latest 128-bit CPU or develop the next generation of embedded cellphone software or the driver for your new video card? In a few years, people will be looking at you to perform such feats, and you better be prepared.

There is a second, perhaps even more important reason for you to study assembly. In the words of Donald E. Knuth, one of the most respected minds in our field:

> Expressing basic methods like algorithms for sorting and searching in machine language makes it possible to carry out meaningful studies of the effects of cache and RAM size and other hardware characteristics (memory speed, pipelining, multiple issue, look aside buffers, the size of cache blocks, etc.) when comparing different schemes.

The point Knuth makes here is that you cannot ever expect to develop proper computer programs if you do not have a basic understanding of how computers work on the lowest level and of how programs are represented there. A point that Knuth even fails to mention is that we live in an online world today and that malicious attackers use *their* knowledge of assembly language to exploit the programs that you may one day write. Thus, learning something about assembly language is a lesson that will be of essential value to you whether you aspire to be a kernel hacker, a systems analyst, a game programmer, a web developer, or a theoretical computer scientist. In fact, here is another priceless quote from Knuth that says it all:

> People who are more than casually interested in computers should have at least some idea of what the underlying hardware is like. Otherwise the programs they write will be pretty weird.

1.2 Assumed Prior Knowledge

Since this lab course is a first year, first quarter course, we expect no prior programming knowledge from you. However, this section briefly describes the experience that we will assume on your part. More importantly, we will refresh and summarise the knowledge that you must have before you can start the assignments.

Background Knowledge

We assume that you will follow and understand the lectures and study the accompanying book and lecture notes in the necessary detail. In particular, we assume that you know and understand what *opcodes*, *instructions*, *subroutines*, *stacks*, *registers* and *program counters* are and that you have a general idea of what occurs during the compilation and linking stage of an executable program. We further assume that you know the difference between *bits*, *nibbles* and *bytes* and that you can convert numbers between different number representation systems (hexadecimal, binary, etc.) and we assume that you understand the concept of *endianness*. These topics will all be treated during the lectures and instructions.

In this lab course you will learn how to program in the x86-64 assembly language. You should know that "the x86-64 architecture" is the generic name for the architecture of CPUs found in garden-variety personal computers[1]. We assume that you have studied the x86-64 architecture in your lecture notes. You should know that x86-64 has fourteen 64-bit general purpose registers (`RAX`, `RBX`, `RCX`, `RDX`, `RDI`, `RSI`, and `R8`-`R15`) which you can use freely when writing programs. It also has a 64-bit stack pointer register (`RSP`) which contains the memory address of the top of the current program stack and a base pointer register (`RBP`), which is used during subroutine execution. The purpose of these registers will be explained to you during the assignments.

Finally, you should know what we mean by the Von Neumann architecture, i.e., you should know that a computer is roughly comprised of three subsystems: a *CPU*, a *random access main memory* and an *IO subsystem* which is in turn comprised of IO devices such as mice, keyboards, sound cards, hard disks, etc. You should know that the CPU is capable of executing *instructions* which reside in the main memory and that instructions are simply binary codes of varying lengths. In the next paragraph we refresh your knowledge of some important definitions regarding computer languages.

Essential Concepts

It is likely that you do not yet have a very clear image in your mind as to what goes on inside the bowels of your computer when it is executing a program. The main goal of this lab course is to remedy this situation. To start off, we will eliminate any romantic preconceptions on your part regarding machine, assembly, and high-level programming languages by carefully restating their

[1] In the literature you may encounter the terms AMD64 and x64, which are synonyms for x86-64.

definitions and their respective purposes. Much of this should already be known to you but it is essential that you read the following carefully.

Computer memory stores programs as sequences of instructions and data in binary format. To a computer, there is no essential difference between instructions and data. Here is a real example of part of a computer program, as it would look in the main memory of a computer:

```
 0          01001000
 1          11000111
 2          11000000
 3          00000001
 4          00000000
 5          00000000
 6          00000000
 7          01001000
 8          11000111
 9          11000001
10          00000001
11          00000000
12          00000000
13          00000000
14          01001000
15          00000001
16          11000001
```

The line numbers are not part of the program but you could think of them as memory addresses, as each byte has its own memory address. This type of "zeros and ones" program representation is called the *machine language* representation of a program. The machine language representation is, of course, the only representation that a computer can understand. Any higher level language representation of a program (such as a Java or C program) needs to be translated into machine language at some point, before it can be executed by the hardware. Machine languages are specific to the CPU architecture, e.g., there is a PowerPC machine language, an x86-64 machine language, etc.

The little code snippet above is actual x86-64 machine code written in binary notation. As you can see, programs take up a large amount of space when written in binary, so we will commonly write such code in hexadecimal format. As you know, each nibble is represented by one hex digit so we can rewrite these 17 bytes of code in 34 hex digits:

```
48 c7 c0 01 00 00 00  # move the number 1 into the RAX register
48 c7 c1 01 00 00 00  # move the number 1 into the RCX register
48 01 c1              # add the contents of RAX to RCX
```

We have regrouped the bytes in such a way that one instruction fits on one line and we have added some explanatory comments, which are denoted by the '`#`' character. Do not be afraid at this point if you do not understand the code fragment completely, but please do take a close look. In this piece of code you can see three x86-64 instructions with some operand data. The first instruction is the so called `mov` instruction. It moves a value to a register. In this case, the value is the number 1 and the register is the x86-64's RAX register. On the second line you see another `mov` instruction, again with operand 1, but this time it moves the value to the RCX register. The third line shows us the `add` instruction which sums the contents of RAX and RCX and stores the result in RCX. As you can see, not all of the instructions in the x86-64 architecture are of the same length. Usually the actual instruction is one to two bytes long. The `mov` instruction is only two bytes long (`48 c7`) and the `add` instruction is three bytes long (`48 01 c1`). The operand data of the `mov` instructions is four bytes long. Since x86-64 is a *little-endian* machine (see 2.2), the four byte integer 1 is encoded as `01 00 00 00` as you can see. One final thing to note is that the operand registers are encoded as `c0` and `c1`.

In ancient times[2], programmers had to enter programs into the computer's memory by means of *punch cards* - small pieces of cardboard which contained zeros and ones encoded in the presence or absence of holes in the cardboard. Obviously, writing computer programs in zeros and ones or hexadecimal numbers like this was a very cumbersome, error-prone task and in modern times it would be next to impossible. Modern computer programs easily contain around 10 MB of machine code, which would amount to 83 886 080 zeros and ones or 20 971 520 hex digits. You could imagine the horror of having to type them by hand. Worse, you could imagine the horror of finding bugs in such programs! For these reasons, people switched to *assemblers* in the 1950s. Assemblers are special computer programs that translate text from a more humanly readable symbolic assembly language ("assembly" for short) into machine code. In the assembly language, each instruction code has a short *mnemonic* or nickname associated with it and each number can be represented in decimal or hex, instead of in bits. Just as each architecture has its own machine code, each machine code has its own assembly language. Below, we see the same program snippet as above, but this time it is written in x86-64 assembly language:

```
movq    $1, %rax    # Move the number 1 into the RAX register
movq    $1, %rcx    # Move the number 1 into the RCX register
addq    %rax, %rcx # Add the contents of RAX to RCX
```

As you can see, our cryptic piece of binary machine code is as simple as 1 + 1! Well, almost. The most significant property of assembly language is that it closely resembles the raw machine code in structure. If you have a table of opcodes and some knowledge of the fields in an x86-64 instruction you could easily translate this assembly code directly into machine code - even by hand.

Unfortunately, Writing large programs in assembly language still has its drawbacks. As programmers, we still have to deal with millions upon millions of instructions and we still have to concern ourselves with registers, stacks and memory locations, while we would rather like to focus on solving our problems. If we want to add two numbers, we would like to tell the computer something like `y = a + b` rather than having to explain the operation in register-level detail. In short, we would like to program computers in a language that translates easily to mathematics or English rather than machine language. This desire prompted the development of so-called 3GLs[3]. You have probably heard of a lot of these 3GLs and in due time you will learn many of them: C, C++, Java, Pascal, Prolog, Haskell, etc. In order to run a program that is written in a 3GL, we need to translate it to assembly language first[4]. The tools that do this are called *compilers*. Unlike assemblers, compilers are amongst the most complex of computer programs in existence. Compiler technology continues to evolve as it has done for over sixty years since admiral Grace Hopper wrote the first compiler in assembly language[5]. It is often difficult to predict exactly what instructions a compiler will generate when given a particular snippet of 3GL code. Nonetheless, below is a line of C/Java code that might cause a compiler to spit out something that looks like our example fragment:

```
int x = 1 + 1;
```

And there it is! As an aside, it is, in fact, highly unlikely that a compiler will ever generate our little snippet of assembly code due to optimisations like *constant folding*. Luckily for you, that is entirely outside the scope of this lab course.
You should now be mentally prepared to start the assignments. Good luck!

[2]In computing terms, "ancient" is roughly between 1920 and 1950.

[3]This stands for "third generation language", with machine language being the first and assembly being the second generation.

[4]We could also use an interpreter, but we ignore that option here.

[5]Hopper is also renowned for the discovery of the first "bug": a dead moth in one of the relay switches of the Mark II calculator.

Chapter 2

Mandatory Content

In this section you will study the mandatory content of this lab which includes the development process, implementation of a simple, non-trivial example program as well as some basic programming constructs you will need within this lab. This chapter also contains the mandatory assignments. The manual is shaped in such a way that all the knowledge you need for an assignment is located before it so make sure to read everything in order so as to not be confused about what to do.

In the assignments you can borrow ideas from the example for your own programs but for now, the most important thing is that you will learn:

- what an assembly program looks like and how it is structured
- how to transform an idea into a good specification
- how to transform a specification into an assembly program
- how the basic programming constructs work in assembly (if/else, while, for, etc.)

2.1 Designing a Program

We will start by describing the program in plain English. We will then write the algorithm down more formally in *pseudocode*. Finally, we will translate the pseudocode into working assembly code and we will discuss that code in line by line detail.

Step 1: Description

The program we are going to write is quite simple: it will ask the user for an input number and increment it by 1 if it is even, otherwise return the same number. For simplicity, we will assume that the input will always be greater than or equal to 0. The following table illustrates some possible inputs and their outputs:

input	output
0	1
1	1
2	3
3	3
10	11
21	21
42	43
1041	1041

Step 2: Specification

Now that we are familiar with what our program is supposed to do, we will transform the description into a formal specification. Why do we have to do this? Well, because in the real world, there are many small, practical details that need to be considered before we can actually implement an algorithm. One such problem is the problem of representation: how will we represent the information in our program? Will we use a *linked list*[1] structure, an array, a map? Is a structure even needed? What implications will it have for the complexity of our program if we choose one representation over another? Resolving such questions is part of the creative challenge of programming, so usually you will have to decide on these matters for yourself. However, we will always expect you to formalise these decisions in the form of a good specification, before you start programming. As an example of what we consider to be a good specification, we present the specification of our program below.

Since this program is very simple, it does not actually require any structure to hold our data but simply a single variable on which we will perform the operations. Here is the pseudocode[2] that describes it:

```
main() {
        // print the welcome string
        print("Welcome to our program!")

        // call the inout subroutine
        inout()
}

inout() {
        // ask for the input
        int NUMBER = read(keyboard_input)

        // check whether the number is even
        if (NUMBER % 2 == 0) { // use modulo to determine divisibility by 2
                NUMBER = NUMBER + 1 // increment by 1
        }

        // print the outcome
        print(NUMBER)
}
```

The pseudocode above uses only simple operations on a simple data type (an integer) and an if statement which is a basic programming construct. These constructs can easily be translated to assembler programs, as we shall see in the next step.

Step 3: Implementation

The final step in our development process is to translate the specification into working assembler code. Here, we present the complete implementation of the program in working x86-64 assembler. In later exercises, you can use this program as a template for your own work. Try to read along and understand what happens, using the comments in the code and the subsequent explanations as a guide. Do not be intimidated if you do not understand all the details just yet, the following sections contain a detailed explanation of the language.

```
# ************************************************************************
# * Program: Oddifier                                                    *
# * Description: This program prints the closest >= odd number to the input *
# ************************************************************************

.text
welcome:        .asciz "\nWelcome to our program!\n"
prompt:         .asciz "\nPlease enter a positive number:\n"
```

[1] http://en.wikipedia.org/wiki/Linked_list
[2] https://en.wikipedia.org/wiki/Pseudocode

```
input:          .asciz "%ld"
output:         .asciz "The␣result␣is:␣%ld.\n\n"

.global main

main:
        # prologue
        pushq   %rbp                    # push the base pointer
        movq    %rsp, %rbp              # copy stack pointer value to base pointer

        movq    $0, %rax                # no vector registers in use for printf
        movq    $welcome, %rdi          # first parameter: welcome string
        call    printf                  # call printf to print welcome
        call    inout                   # call the subroutine inout

        # epilogue
        movq    %rbp, %rsp              # clear local variables from stack
        popq    %rbp                    # restore base pointer location

end:    # this loads the program exit code and exits the program

        movq    $0, %rdi
        call    exit

# ************************************************************************
# * Subroutine: inout                                                    *
# * Description: this subroutine takes an integer as input from a user, *
# * increments it by 1 if it is even, and prints it out                  *
# * Parameters: there are no parameters and no return value              *
# ************************************************************************
inout:
        # prologue
        pushq   %rbp                    # push the base pointer
        movq    %rsp , %rbp             # copy stack pointer value to base pointer

        movq    $0, %rax                # no vector registers in use for printf
        movq    $prompt, %rdi           # param1: prompt string
        call    printf                  # call printf to print prompt

        subq    $16, %rsp               # reserve space in stack for the input
        movq    $0, %rax                # no vector registers in use for scanf
        movq    $input, %rdi            # param1: input format string
        leaq    -16(%rbp), %rsi         # parameter2: address of the reserved space
        call    scanf                   # call scanf to scan the users input

        movq    -16(%rbp), %rsi         # load the input value into RSI
                                        # (RSI is the second parameter register)

        movq    %rsi, %rax              # copy the input to RAX
        movq    $2, %rcx                # move the value 2 to RCX
        movq    $0, %rdx                # clear the contents of RDX
        divq    %rcx                    # divide the content of RDX:RAX by the
                                        # content of RCX (result stored
                                        # in RAX and remainder in RDX)
        cmpq    $0, %rdx                # compare RDX to 0
        jne     odd                     # if they are not equal (input is odd),
                                        # don't increment
even:
        incq    %rsi                    # increment the input value

odd:
        movq    $0, %rax                # no vector registers in use for printf
        movq    $output, %rdi           # param1: output string
                                        # param2: number (in RSI)
        call    printf                  # call printf to print the output

        # epilogue
        movq    %rbp , %rsp            # clear local variables from stack
        popq    %rbp                   # restore base pointer location

        ret                            # return from subroutine
```

2.2 Assembler Directives

The commands that start with a period (e.g. `.bss`, `.text`, `.global`, `.skip`, `.asciz`, etc.) are *assembler directives*. Assembler directives have special functions in an assembler program. For instance, the `.text` directive at the beginning of the file tells the assembler to put all the subsequent code in a specific *section*. Other assembler directives, like `.global`, make certain labels visible to the outside world. The following is a description of the most commonly used assembler directives or *pseudo-instructions* as they are sometimes called. The directives are grouped by functionality. For a full reference, see the official documentation of the GNU assembler[3].

Section directives: `.text`, `.data`, `.bss`

```
.text
.data
.bss
```

The memory space of a program is divided into three different *sections*. These directives tell the assembler in which section it should put the subsequent code.

The `.text` segment is intended to hold all instructions. The `.text` segment is read-only. It is perfectly fine to include constants and ASCII strings in this segment.

The `.data` segment is used for initialised variables (variables that receive an initial value at the time you write your program, such as those created with the `.word` directive).

The `.bss` segment is intended to hold uninitialised variables (variables that receive a value only at runtime). Therefore, this section is not part of the executable file after compilation, unlike the other two sections.

Defining constants: `.equ`

```
.equ NAME, EXPRESSION
```

The `.equ` directive can be used to define symbolic names for expressions, such as numeric constants. An example of usage is given below:

```
.equ FOO, 1024

pushq $FOO  # push 1024
```

Declaring variables: `.byte`, `.word`, `.long`, `.quad`

```
.byte VALUE
.word VALUE
.long VALUE
.quad VALUE
```

The `.byte`, `.word`, `.long` and `.quad` directives can be used to reserve and initialise memory for variables and/or constants. Just as the assembler translates instructions into bits of memory contents directly (as explained in subsection 1.2), these directives will be transformed into memory contents as well, i.e. there is no special magic involved here. `.byte` reserves one byte of memory,

[3] `https://sourceware.org/binutils/docs-2.25/as/Pseudo-Ops.html#Pseudo-Ops`

`.word` reserves two bytes of memory `.long` reserves four bytes and `.quad` reserves 8 bytes. Whether these bytes will actually be writable depends on the section in which you define them (see above for a description of the sections). Each directive allows you to define more than one value in a comma separated list.

A few examples:

```
FOO: .byte 0xAA, 0xBB, 0xCC    # three bytes starting at address FOO
BAR: .word 2718, 2818          # a couple of words
BAZ: .long 0xDEADBEEF          # a single long
BAK: .quad 0xDEADBEEFBAADF00D  # a single quadword
```

Note that the x86-64 is a *little-endian* machine, which means that a value like `.long 0x01234567` will actually end up in memory as `67 45 23 01`. Of course, you normally do not notice this since the `movl`-instruction will automatically reverse the byte order while it loads the long back into memory. Taking endianness into account, it should be clear that the following three statements are completely equivalent:

```
FOO: .byte 0x0D, 0xF0, 0xAD, 0xBA, 0xEF, 0xBE, 0xAD, 0xDE
FOO: .word 0xF00D, 0xBAAD, 0xBEEF, 0xDEAD
FOO: .long 0xBADF00D, 0xDEADBEEF
FOO: .quad 0xDEADBEEFBAADF00D
```

Reserving memory: `.skip`

```
.skip AMOUNT
```

Sometimes it is necessary to reserve memory in bigger chunks than bytes, words, longs or quads. The `.skip` directive can be used to reserve blocks of memory of arbitrary size:

```
BUFFER: .skip 1024 # reserve 1024 bytes of memory
```

Placing this directive in the `.data` section will initialize all bytes with zero, while placing it in `.bss` will leave the data uninitialized (and will thus contain "random" data from previous programs).

Strings variables: `.ascii`, `.asciz`

```
STRING:  .ascii string
STRINGZ: .asciz string
```

These directives can be used to reserve and initialise blocks of ASCII encoded characters. In many higher-level programming languages, including C, strings are simply blocks of ASCII codes terminated by a zero byte (`0x00`). The `.asciz` directive adds such a zero byte automatically. The following two examples are thus equivalent:

```
WELCOME: .ascii "Hello!!"  # A string..
         .byte 0x00        # ..followed by a 0-byte.

WELCOME: .asciz "Hello!!"  # A string followed by a 0-byte.
```

Global symbols: `.global`

```
.global label
```

This directive enters a label into the symbol table. The symbol table is a table of contents of sorts which is contained in the binary assembled file. Publishing labels in the symbol table is useful if you want other programs to have access to your labels, e.g. if you want the labels to be visible in the debugger or if you want other programs to use your subroutines[4]. One very important use of the symbol table is to export the `main` label. This label *must* be exported because the operating system needs to know where to start running your program.

```
  .global main
```

2.3 x86-64 Assembly Language

This subsection contains a short language reference for the x86-64 assembly language. Apart from a list of commonly used instructions, there is a short rundown of the differences between the so called "AT&T syntax" and the "Intel syntax". This course uses the GNU *assembler*[5]. Because this assembler only supports the AT&T syntax, we use this syntax throughout the course. If you are also interested in the Intel syntax, you can find it in the official Intel x86-64 platform manual.

2.3.1 About the AT&T Syntax

If you have examined some of the x86-64 assembler examples in the book of Hamacher et al., you will have noticed that there is a difference between the x86 assembler they use and the one we use during the lab course. An explanation is in order here. First of all, there is of course no such thing as an "official" x86-64 assembly language. In theory, you could write your own x86-64 assembler and come up with a new syntax of your own. In practice however, there are only two flavors of x86-64 assembly language which are in actual use. There are good arguments for using either, but we have chosen to use the AT&T syntax for the lab course, while the book has chosen to use the Intel syntax. While this is really all you need to know regarding the subject, we provide a short background on the issue for the sake of completeness:

The Intel syntax, as used in the book, is the preferred syntax that was used and developed by the Intel Corporation - the designers of the x86-64 architecture. The Intel syntax is what you will see in the official x86-64 reference manual and platform definition. If anything, this could be considered the "official" x86-64 syntax. Long before the Intel Corporation introduced the x86-64 however, there was the UNIX operating system and the programming language C, both of which were developed at Bell Labs, the R&D department of the American phone company AT&T. Bell Labs and others had ported the UNIX operating system to a variety of different architectures before the x86-64 even existed, and thus the AT&T-style of assembly languages was widespread long before the x86-64 came along. While Intel may favour its own syntax, it is much more beneficial for the rest of us to learn AT&T syntax, as there are many other AT&T-style assemblers available for other hardware platforms. In addition, most compilers generate AT&T-style output and it is, arguably, more elegant than Intel syntax.

As a final note, it is important to stress that the Intel syntax uses a different order for source and destination, e.g. if you read the Intel manual, `mov A B` will copy a value from B to A, whereas in AT&T syntax the equivalent `movl A B` will copy the value from A to B.

2.3.2 Instructions and Operands

The lines of code in an assembly program using AT&T syntax consist of an *instruction* (what should happen) and possibly some *operands* (the data to act with). This paragraph explains how to use the different kinds of operands.

Operand Prefixes: Registers and Literal Values

The AT&T syntax uses a number of prefixes for operands. You have probably seen them already:

[4]Sharing subroutines is not part of this lab course, but if you are interested you can have a look at subsection 3.4

[5]An assembler is a program that translates an assembly code file into machine language. See: `https://sourceware.org/binutils/docs-2.25/as/index.html`

- Register names are prefixed by the `%` character (e.g. `%rax`, `%rsp`).
- Literal values are prefixed with the `$` character (e.g. `$3`).

Instruction Postfixes (Specifying Operand Size)

Many instructions in AT&T syntax need to be postfixed with a `b`, `w`, `l` or `q` modifier. This postfix specifies the size of the operands, where `b` stands for "byte", `w` stands for "word" (2 bytes), `l` stands for "long" (4 bytes) and `q` stand for "quadword" (8 bytes). As an example, take a look at this concept if we apply it to the `push` instruction:

```
pushb $3 # Push one byte onto the stack (0x03) (NOTE: See below)
pushw $3 # Push two bytes onto the stack (0x0003)
pushl $3 # Push four bytes onto the stack (0x00000003)
pushq $3 # Push eight bytes onto the stack (0x0000000000000003)
```

Note: the `push` *instruction does not actually support pushing only one byte, we just show the idea here such that you can apply it to more complicated instructions which do support it, like* `mov`*.*

All four instructions in the example push the literal value '3' onto the stack, but the actual size of the operand is different in each situation. We will do assembly in 64-bit, so mostly you will have to use the `q` postfix.

In our instruction set reference table in section 2.3.3, we do not list these suffixes explicitly. The Intel manual does not list them either.

Partial Registers

The size suffix is especially important when you use *partial registers*. The x86-64 allows you to address smaller parts of the 64-bits registers through special names. By replacing the initial `R` with an `E` on the first eight registers, it is possible to access the lower 32 bits. If we use the `RAX` register as an example, you can address the least significant 32 bits of this register by using the name `EAX`. To access the lower 16 bits the initial R should be removed (`AX` for `RAX`). In a similar fashion, the highest and lowest order bytes in this `AX` register can be addressed by the names `AH` and `AL` respectively. Again, we present a few examples using the `mov` instruction:

```
movl %eax, %ebx # Copy 32 bits values between registers
movq %rax, %rbx # Copy 64 bits values between registers
movw %ax, %bx   # Copy only the lowest order 16 bits
movb %al, %bl   # Copy only the lowest order 8 bits
movb %ah, %al   # Copy 8 bits within a single register
```

The following table shows how you can access the lower X bytes of each register:

8-byte	4-byte	2-byte	1-byte
%rax	%eax	%ax	%al
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

Addressing Memory

There are several ways of addressing the memory in AT&T syntax:

- **Immediate addressing:**
  - Directly using a label will yield the value at the address of the label. (e.g. `label`)
  - Prefixing a label with a `$` will yield the address of the label. (e.g. `$label`)
- **Indirect addressing:**
  - Surrounding a *register* with parentheses will yield the value at the memory address stored in the register. (e.g. `(%RAX)`)
  - Prefixing the left parenthesis with a *displacement* will yield the value at the memory address stored in the register plus the displacement (e.g. `-8(%RBP)`).
  - *Advanced:* Accessing memory with *displacement*`(`*base*`,` *index*`,` *scale*`)` will yield the value at memory address "*displacement* + *base* + *index* × *scale*" Here, *displacement* is a constant expression (may include labels), *base* and *index* are registers, and *scale* is either 1, 2, 4, or 8. (e.g. `table(%RDI, %RCX, 8)`)

  Indirect addressing is explained in more detail in the GNU assembler documentation [6].

2.3.3 Instruction Set

The example program uses a number of commonly used instructions, such as `mov`, `push`, `cmp` and `jmp`. The next page contains a list of commonly used x86-64 instructions. It should be sufficient to get you through the lab course, but you may always study the official Intel manual [7] to obtain more instructions. Some important notes:

- The instructions are all case insensitive.
- In the table we denote the necessity of a `b`, `w`, `l` or `q` postfix with a period ('`.`').
- Most of the instructions with two operands require at least one of their operands to be a register. The other operand may be either a register or a memory location. This may differ per instruction and is specified in detail in the official Intel manual [7].

[6]`https://sourceware.org/binutils/docs-2.25/as/i386_002dMemory.html#i386_002dMemory`

[7]`http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, specifically volume 2, the instruction set reference

- The multiplication and division instructions require their operands to be in special registers[8]. The multiplication instruction will store the result in both `%RDX` and `%RAX` (denoted in the table by `%RDX:%RAX`), while the division instructions will require the dividend to be in these two registers. The higher-order bits should be in `%RDX` while the lower-order bits should be in `%RAX`. Note that the division operator also stores the remainder of the division in `%RDX`.

Mnemonic	Operands	Action	Description
			Data Transfer
`mov.`	SRC, DST	DST = SRC	Copy.
`pushq`	SRC	`%RSP` -= 8, (`%RSP`) = SRC	Push a value onto the stack.
`popq`	DST	DST = (`%RSP`) , `%RSP` += 8	Pop a value from the stack.
`xchg.`	A, B	TMP = A, A = B, B = TMP	Exchange two values.
`movzb.`	SRC, DST	DST = SRC (one byte only)	Move byte, zero extended.
`movzw.`	SRC, DST	DST = SRC (one word only)	Move word, zero extended.
			Arithmetic
`add.`	SRC, DST	DST = DST + SRC	Addition.
`sub.`	SRC, DST	DST = DST - SRC	Subtraction.
`inc.`	DST	DST = DST + 1	Increment by one.
`dec.`	DST	DST = DST - 1	Decrement by one.
`mul.`	SRC	`%RDX:%RAX` = `%RAX` * SRC	Unsigned multiplication.
`imul.`	SRC	`%RDX:%RAX` = `%RAX` * SRC	Signed multiplication.
`div.`	SRC	`%RAX` = `%RDX:%RAX` / SRC `%RDX` = `%RDX:%RAX` % SRC	Unsigned division.
`idiv.`	SRC	`%RAX` = `%RDX:%RAX` / SRC `%RDX` = `%RDX:%RAX` % SRC	Signed division.
			Branching
`jmp`	ADDRESS		Jump to address (or label).
`je`	ADDRESS		Jump if equal.
`jne`	ADDRESS		Jump if not equal.
`jg`	ADDRESS		Jump if greater than.
`jge`	ADDRESS		Jump if greater or equal.
`jl`	ADDRESS		Jump if less than.
`jle`	ADDRESS		Jump if less or equal.
`call`	ADDRESS		Jump and push return address.
`ret`			Pop address and jump to it.
`loop`	ADDRESS		`decq %RCX`, jump if not zero.
			Logic and Shifting
`cmp.`	A, B	`sub` A B (Only set flags)	Compare and set condition flags.
`xor.`	SRC, DST	DST = SRC ˆ DST	Bitwise exclusive or.
`or.`	SRC, DST	DST = SRC | DST	Bitwise inclusive or.
`and.`	SRC, DST	DST = SRC & DST	Bitwise and.
`shl.`	A, DST	DST = DST << A	Shift left by one bit.
`shr.`	A, DST	DST = DST >> A	Shift right by one bit
			Other
lea.	A, DST	DST = &A	Load effective address.
int	INT_NR		Software interrupt.

[8]Other forms of these instructions also exist, but they are not listed here.

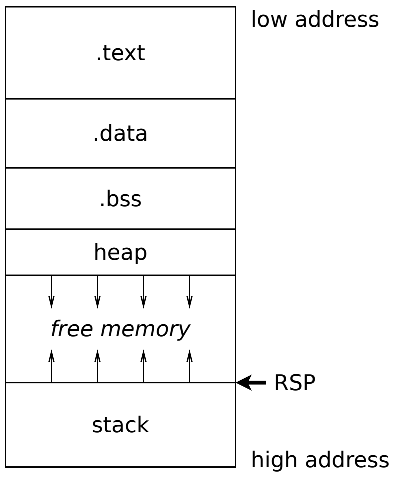
2.4 Registers & Variables

The example program from section 2.1 uses a variable to store data. We see in the comments that we are using a variable called "`number`" which corresponds to the one used in the pseudocode. But where do these variables live? Do they exist in the registers, on the stack or somewhere in main memory? The answer is: all of the above. Sometimes, like in the case of `number`, we can simply keep our variables in the registers. The registers are fast and easy to access, so if possible we like to keep our variables there. The number of registers is limited however, so sometimes we may have to temporarily push their values on the stack and later pop it into a register when we need to check the value.

Aside from register shortage, there is one very important reason to store variables on the stack in some cases: on the x86-64 platform, registers are *caller saved* by convention. This means that if you call a subroutine from your program, like `printf` or one of your own subroutines, the subroutine may and will likely overwrite some of your registers. In other words, if you need the data in your registers to be consistent after you call a subroutine, you will need to save it on the stack.

2.5 The Stack

Here is a visual impression of the memory layout of a running process:



The parts of the memory labeled `.text`, `.data`, and `.bss` contain all program instructions and other data originating from assembler directives. More information on these memory sections can be found in the assembler directive reference (paragraph 2.2). The *heap* is used to store data allocated using the C functions `malloc` and `calloc`. [9]

2.5.1 The Stack Pointer

We will now explain the stack mechanism. The x86-64 has a special stack pointer register: `RSP`. This register is initialised by the operating system once your program starts. At that point, it contains the address of the first byte *after* your program's memory space. Essentially, this means that the stack is empty at this point in time. The stack can "grow" downward into your program's memory space. When a `push` instruction is executed, the value in the stack pointer register gets decremented by some amount and the pushed value is stored at the new location at which the stack pointer then points.

[9]We will not use the heap in the compulsory part of this lab, but the interested can find more information on how the heap works in C: `https://www.gribblelab.org/teaching/CBootCamp/7_Memory_Stack_vs_Heap.html`

2.5.2 Cleaning up the Stack

Of course, if the stack grows too large, it will eventually overwrite your program's code and data. This is called a *stack overflow* and it is usually indicative of a serious design flaw in your program. To avoid this problem, the caller must clean up the stack after every function call by adding the parameter-block size to the stack pointer directly. This is because (as you will read in section 2.6) the stack is at times used to pass parameters to subroutines. Look at the simple example of the print function below. Cleaning the stack should not be more difficult than this:

```
    pushq $42                # Push a magic number, the seventh argument
    movq  ... , %...         # Move arguments 2 through 6 to their registers
    movq  $formatstr , %rdi  # First argument: the format string
    movq  $0, %rax           # no vector arguments for printf
    call  printf             # Print the numbers
    addq  $8, %rsp           # Clean the stack (pop the magic number)
```

2.5.3 The Base Pointer

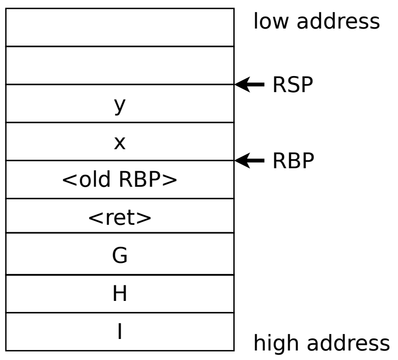
Usually, subroutines push and pop values on and off the stack. With the stack pointer being ever in motion, it may become hard to keep track of things that reside on the stack. To make stack navigation in subroutines easier, the x86-64 offers a special base pointer register: `RBP`. It works like this: upon entry of our subroutine, we immediately push the current value of `RBP` onto the stack. We then copy the current stack pointer value to `RBP`. During the lifetime of our subroutine, `RBP` will not change. That is, it will always point at the "base" of our subroutine's stack area. This way, we can always find our local variables and subroutine arguments relative to `RBP`. At the end of our subroutine we pop the old `RBP` value off the stack again and return from the subroutine.

2.5.4 Subroutine Prologue and Epilogue

At the beginning of the example program, we see that the base pointer is being initialised. The opaque ritual has to be performed at the start of each subroutine, including the `main` subroutine and is explained in the following section.

The process of storing the old base pointer and copying the stack pointer to be a new base pointer is called the *subroutine prologue*. Similarly, restoring the old base and stack pointers is called the *subroutine epilogue*. These two parts of code should be the same for all subroutines you write and can therefore also be seen as part of the calling conventions (paragraph 2.6.1).

To the right is a graphical representation of the stack as it would look during the execution of a typical subroutine. Each "block" in the image represents eight bytes. The calling subroutine has left the values `I`, `H`, and `G` on the stack. On top of that, we find the return address (pushed by the `call` instruction), the base pointer of the calling subroutine (pushed in the *prologue* of the current subroutine), and two values `x` and `y` (which were created on the stack during the current subroutine).



In this same image, you can see the base pointer register pointing to the first memory location that the current subroutine uses. Similarly, the stack pointer points to the first free memory location on the stack. The space between the base pointer and the stack pointer is also called the *stack frame* of the current subroutine.

Please note that "pointing at" can be misinterpreted. The stack pointer register points at the location between the last pushed value and the free space. When we read a value at a pointer, we read from low to high addresses (downwards in the image). This means that if we read the value at `(%RSP)` we read the last pushed value. Similarly, to get the first pushed value in our stack frame, we need to read at `-8(%RBP)`.

2.5.5 Accessing arguments passed via the stack

Take another look at the memory layout in the previous image. Now imagine that the current subroutine is one that takes nine arguments. Knowing how the memory is laid out on the "border" between two subroutines, we know that we can find the stack arguments two memory spaces (or, 16 bytes) below the current base pointer. You can use indirect memory addressing (see paragraph 2.3.2) to retrieve these values.

2.6 Subroutines

A subroutine is simply a block of instructions which starts at some memory address (indicated with a *label*). If we want to execute or *call* a subroutine, we simply need to jump[10] to its first instruction. After executing the subroutine we expect control to return to us, i.e. we expect the program to return to the first instruction after our subroutine call. To make this possible, the called subroutine should somehow be aware of the address of the next instruction after the call. By convention, we simply push that address onto the stack right before making the jump. To our ease and comfort, the kind people at Intel provided a single instruction that performs both these steps in one fell swoop: the `call` instruction. Calling a simple subroutine is thus no more difficult than this:

```
call somesub   # call the somesub subroutine
```

The label `somesub` in this example is associated with the starting address of the subroutine that we want to call. The `somesub`-subroutine will now execute all of its instructions and finish off with the `ret` instruction. This instruction will pop the return address (which the `call` instruction had pushed) from the stack and make execution will simply return to the first instruction after the `call` instruction.

2.6.1 Calling Conventions

The calling of subroutines hinges heavily on a number of conventions. Thanks to these conventions, if you know how to call one subroutine you know how to call them all. Imagine if this was not the case, you would have to check the exact register and stack usage of each subroutine you would want to use. On the flip side, when writing our own subroutines we will have to honour these conventions as well.

Passing Parameters

Usually, we will want to pass some *parameters* to a subroutine. To do this, we need to put them somewhere where the subroutine can find them when it executes. We are more or less free to choose between using the registers, the stack or some part of memory other than the stack to store these parameters, as long as both the writer of the subroutine and the user agree on the location. By convention[11] we will use registers for this purpose.

[10] A "jump" is nothing more than loading a new memory address into the program counter, or `RIP`, as this register is called on the x86-64.

[11] This convention is the so called "C calling convention" and if we adhere to it our subroutines and calls will be fully compatible with the system's standard C library. You can find the full documentation of the calling conventions here: `https://web.archive.org/web/20160801075139/http://www.x86-64.org/documentation/abi.pdf`

More specifically, we will place the arguments in the following registers:

1. `%RDI`
2. `%RSI`
3. `%RDX`
4. `%RCX`
5. `%R8`
6. `%R9`

We will clarify this by an example. Let us assume that we have a subroutine called `foo`, that takes three integer arguments, i.e. the signature of the subroutine is `foo(int a, int b, int c)`. Imagine that we want to call `foo` with the parameters 1, 5 and 2, i.e. `foo(1, 5, 2)` in pseudocode. In assembler, we copy the arguments in the registers and execute the `call` instruction to call this subroutine:

```
    movq $2, %rdx # third argument
    movq $5, %rsi # second argument
    movq $1, %rdi # first argument
    call foo      # Call the subroutine
```

If the subroutine that you are calling needs more than six arguments, then the remaining arguments need to be pushed to the stack in reverse order (first argument pushed last). Note that the called subroutine will *not* remove the arguments from the stack, so you should pop them off yourself after the call returns. If you are interested in writing your own subroutine that needs more than six arguments, see paragraph 2.5.5.

Callee-saved vs. Caller-saved

Note that every subroutine is limited to use the same general-purpose registers, so the values in these registers might no longer be the same when the function returns. If you want to preserve these values, you will need to save them somewhere (e.g. the stack).

Some registers are caller-saved: these registers may be modified by subroutines and should thus be saved by the caller of a subroutine if the value needs to be used after the call returns. The list of caller-saved registers is: `%RAX`, `%RCX`, `%RDX`, `%RDI`, `%RSI`, and `%R8` through `%R11`.

Registers can also be callee-saved; these registers may be used by a subroutine, but when the subroutine returns they must have the same value they had when the subroutine started execution. The list of callee saved registers is: `%RBX`, `%RSP`, `%RBP`, and `%R12` through `%R15`.

Stack Alignment

If you are going to use the stack in your code, you need to make sure that the stack remains 16-byte aligned. This means that the `%RSP` register should always be a multiple of 16 when you do a call.[12]

If you are not using the stack inside a subroutine, then this is easy: any `call` instruction pushes an 8 byte return address and in the prologue of your function you push the old `%RBP` value. These two pushes together are exactly 16 bytes, thus ensuring your stack remains aligned. For more information on why you need to push the `%RBP` register to the stack, see paragraphs 2.5.3 and 2.5.4.

Note that the `main` routine will also be called with an aligned stack, but the return address pushed by this call causes the stack to be unaligned again.

[12]For example, the `scanf` function will crash with a segmentation fault if this is not the case!

The Return Value

The final question that remains regarding the invocation of subroutines is that of the return value. Some subroutines (such as `sqrt()` or `sin()`) return a value after they execute. By convention, subroutines leave their return value in the `RAX` register. If for example our `foo` returned an integer, it would be in the `RAX` register after the `call` instruction returned.

2.6.2 Recursive Subroutines

A recursive subroutine is a subroutine that calls itself during its execution. This enables the subroutine to repeat itself for a number of times. Below is the pseudocode of a recursive example function. For a given x less than or equal to 42, the function calculates and returns the sum of all integers from x to 42.

Pseudocode:

```
function example(x) {
    if (x == 42)
        return 42;
    else
        return (x + example(x + 1))
}
```

With recursive subroutines there's still an issue: when does the routine need to stop from calling itself? To prevent infinite recursion, you need to determine a recursive case and a base case (or stop condition). With this example it would be logical to stop the recursion when the function receives an input value of 42. This is done by checking for the condition at every invocation of the function. If the condition holds we can return a known correct value. If the condition did not hold the function will call itself with different parameters and use the result of that invocation to compute the correct value.

Recursive functions are often used in computer science because they allow programmers to write a minimal amount of code. It often produces code that is very compact. However, recursion can cause infinite loops when the stop condition is not written properly.

2.7 I/O

If you examine the pseudocode and the resulting assembly code of the example carefully, you see that we have translated the `print(number);` statement into the following lines of assembler code:

```
movq    $formatstr, %rdi    # first argument: formatstr
movq    %rcx, %rsi          # second argument: the number
movq    $0, %rax            # no vector arguments
call    printf              # print the number
```

Doing I/O in an assembler program can be quite tricky. First of all, normal processes do not have permission to access the hardware I/O devices directly, so all input and output has to be handled by the operating system. Since different operating systems have different ways of doing things it isn't very useful to teach you the specifics of one system[13]. Instead, we will use the operating system's standard C library to do I/O for us. Calling functions in the C library is no different from calling subroutines in your own programs. This has many benefits. First of all, there is a standard C library available on most operating systems and second, it will do some nice tricks for us such as ASCII-to-integer conversions and vice versa. In this subsection we will discuss the `printf` and `scanf` subroutines from the standard C library. Both these functions are functions that take a non-fixed amount of arguments, also known as "varargs". These functions take an extra (hidden) argument in `RAX`, defining the number of vector registers used in the call. During this lab we will not be using these registers, so you always load a zero into `RAX`.

[13]If you are curious anyway, check out paragraph 3.3.

2.7.1 Printing to the Terminal

The standard C library contains a subroutine called `printf`. We will use this subroutine for output. The subroutines from the C library can be called directly from your programs, just like normal subroutines. The *linker* will make sure that the actual subroutine is found once your program is built. `printf` takes a variable number of arguments.

Basic Example

In its simplest form, `printf` takes only one argument: the memory address of a string of ASCII characters. We will now present a pseudocode example followed by an assembly example.

Pseudocode:

```
    printf("Hello world!\n");
```

Assembly:

```
mystring: .asciz "Hello world!\n"

    movq $0, %rax          # no vector registers in use for printf
    movq $mystring, %rdi   # load address of a string
    call printf            # Call the printf routine
```

As you can see there are two strange details in this example. First of all, we include the special '\n'-sequence inside the string. This is translated by the assembler to a single 'newline'-character. Second, we do not actually provide the entire string as an argument, but rather just the memory address of the first character of the string, which is denoted by the `mystring` label[14]. By convention, C functions know where a string ends by looking for a byte with the value `0x00`. That byte indicates the end of the string.

Printing Variables

In addition to simple printing, we can also use `printf` to print variables and other calculated output to the terminal. We do this by embedding special character sequences in our string and by passing extra values to `printf`. Note that these character sequences have no special meaning for the assembler like '\n', instead they are understood by the `printf` function (and related functions).

We give another example.

Pseudocode:

```
    printf("I am %ld years old\n", 25);
```

Assembly:

```
mystring: .asciz "I am %ld years old\n"

    movq $0, %rax          # no vector registers in use for printf
    movq $25, %rsi         # load the value
    movq $mystring, %rdi   # load the string address
    call printf            # Call the printf routine
```

The `printf` function will automatically convert the integer value 25 to a ASCII representation of the decimal number 25 and it will substitute the value into the string at the point where the '%ld' sequence is encountered. The '%ld' sequence simply tells `printf` that it may expect an extra argument and that the argument must be interpreted as a long decimal number (64 bits) for printing.

[14]Remember that a label is just a memory address?

Other Format Specifiers

There are many other format specifiers, but you will probably not need to use them for the compulsory assignments. For the interested, here are some of the most commonly used format specifiers[15]:

Specifier	Usage
`%d`	decimal number (32 bits)
`%ld`	long decimal number (64 bits)
`%lx`	hexadecimal number (64 bits, using lowercase `a-f`)
`%lX`	hexadecimal number (64 bits, using uppercase `A-F`)
`%lu`	unsigned decimal number (64 bits)
`%c`	character
`%s`	string of characters (passed as a memory address)
`%%`	the literal character '%'

2.7.2 Reading from the Terminal

To gather input from the user, we use another routine from the system C library called `scanf`. This routine also has powerful number conversion facilities which work in a similar fashion as the `printf` subroutine we saw in the last paragraph. We supply at least two arguments to `scanf`, the first one being a *format string* containing a number of special character sequences and the subsequent ones being memory addresses at which `scanf` may put the read values. In the following pseudocode we use the '&' operator to denote "address of", e.g. `&number` is the memory address of the variable `number`.

Pseudocode:

```
int number;
scanf("%ld", &number);
```

In assembly, the address of a variable depends on its location. If you want to store a value into a global address you can simply use its label as the address. If you want to put a value in a stack variable you could calculate the address using the base pointer. Fortunately, x86-64 offers a `lea` instruction ("load effective address") which makes this rather simple. We provide a complete example of a `scanf` call which reads a decimal number from the keyboard and stores it in some local stack variable:

Assembly:

```
formatstr: .asciz "%ld"
      ...
      subq $8, %rsp           # Reserve stack space for variable
      leaq -8(%rbp), %rsi     # Load address of stack var in rsi
      movq $formatstr, %rdi   # load first argument of scanf
      movq $0, %rax           # no vector registers for scanf
      call scanf              # Call scanf
```

2.8 The End of the Program

At the end of the example program, we see another call to a function in the C system library. In most operating systems, programs need to return an *error code* which tells the operating system whether your program encountered any internal errors while running. By convention, programs return zero if no errors were encountered. Furthermore, the operating system may want to do some cleaning up after a program runs. To facilitate this, we call the `exit` function with our error code (zero) in the same way we used the `printf` function earlier.

[15]Other format specifiers are neatly listed at `http://www.cplusplus.com/reference/cstdio/printf/`

2.9 Programming Constructs

In your pseudocode specifications you will often use common, high-level programming constructs such as `if`-statements, `while`-loops and `switch`-statements. In this subsection we show you how to transcribe these constructs into assembly language by means of a series of examples. Fundamental to all the conditional examples is the general concept of *Conditional branching*, which is discussed in paragraph 2.9.1. In paragraphs 2.9.2 through 2.9.3 we provide examples of actual programming constructs and Chapter 3 provides some higher-level ones which are not necessary for the mandatory lab.

2.9.1 Conditional Branching

There are many so-called "jump" or "branch" instructions in the x86-64 instruction set which load a new value into the program counter. These instructions come in two flavors. First, there are the regular branch instructions such as `jmp` or `call` which cause program execution to continue at a different memory address. Second are the *conditional* branch instructions, which will only jump to the new target address if some condition holds. We can use these conditional jump instructions to implement conditional constructs, such as `if`-statements and `while`-loops:

Pseudocode:

```
if(RAX > 1) {
     //IF–code
} else {
     //ELSE–code
}
```

Implementation:

```
    cmpq $1, %rax    # compare RAX to 1
    jg ifcode        # jump to IF–code if RAX > 1
    jmp elsecode     # jump to ELSE–code otherwise

ifcode:
    ...              # IF–code
    jmp end

elsecode:
    ...              # ELSE–code
end:
```

The `cmp` instruction on the first line compares the contents of `RAX` to the number 1. It stores the results of this comparison (e.g. whether the contents of `RAX` were greater than-, equal to or less than 1) in the special `RFLAGS` register. The `jg` instruction ("jump if greater-than") is a conditional branch instruction. It tests the contents of the `RFLAGS` register and jumps to the `ifcode` label *if* the flags indicate that the second operand of the `cmp` instruction was greater than the first. For an overview of the various conditional branch instructions, see the instruction set reference in paragraph 2.3.3. The subsequent paragraphs will demonstrate other programming constructs based on the conditional branch instructions. Paragraph 2.9.2 will give a more compact implementation of the `if`-statement.

2.9.2 `if-then-else` Statements

In the previous paragraph we have seen an example implementation of the familiar `if`-statement. In this paragraph we change the sequence of the if- and else-blocks to come to a shorter implementation.

Pseudocode:

```
if(RAX > 1) {
     //IF-code
} else {
     //ELSE-code
}
```

Implementation:

```
     cmpq $1, %rax # compare RAX to 1
     jg ifcode     # jump to IF-code if RAX > 1

elsecode:
     ...            # ELSE-code
     jmp end

ifcode:
     ...            # IF-code
end:
```

2.9.3 Loops

The `do-while`-loop In this example we jump back to the beginning of the loop as long as the condition holds. This is the simplest type of loop.

Pseudocode:

```
do {
     //loop code
} while(RAX > 1);
```

Implementation:

```
loop:
     ...           # loop code

     cmpq $1, %rax # repeat the loop
     jg  loop      # if RAX > 1
```

The `while`-loop In this example we will break the loop if the condition does *not* hold, i.e. we jump to the end if `RAX` is lesser or equal to 1.

Pseudocode:

```
while(RAX > 1) {
     //loop code
}
```

Implementation:

```
loop:
     cmpq $1, %rax # if RAX <= 1 jump to
     jle  end      # the end of the loop

     ...           # loop code

     jmp loop      # repeat the loop
end:
```

The `for`-loop A `for` loop is really nothing more than a glorified `while`-loop.
Pseudocode:

```
for(RAX = 0; RAX < 100; RAX++) {
    //loop code
}
```

```
RAX = 0;
while(RAX < 100) {
    //loop code

    RAX++;
}
```

You should be able to implement this one yourself.

2.10 Bit Shifting

One of the powerful things about the assembly language is that you can very easily manage your values on the bitwise level. Bit shifting is a common advantage of this property.

The following is an example of a bit shift:

```
    movq    $0, %rax        # 00000000 ... 00000000 00000000
    movb    $142, %al       # 00000000 ... 00000000 10001110
    shr     $3, %rax        # 00000000 ... 00000000 00010001
```

It shifts the bits in memory by the number and in the direction you specify (`shl` is left-shift, `shr` is right-shift).

Chapter 3

Bonus Content

This chapter contains only bonus content. For every lab assignment (see the assignment manual), you will have a list of content that you must be familiar with in order to successfully complete it. You may get an introduction to said content here.

3.1 ANSI Escape Codes

ANSI escape codes (also known as ANSI sequences) were invented in the 70s to control the text terminals. This way they could do things like setting the cursor position, clear the screen, change the colours, etc in a standardised manner. For the terminal to understand that such a sequence needs to be treated differently from the normal text it starts with the escape character (ASCII character 27) followed by '['. Once the terminal detects these 2 characters the next few characters are interpreted as an ANSI code.

Take, for instance, the command below.

```
\x1B[4A
```

It moves the cursor 4 lines up. `1B` is the hex representation of the ASCII character 27. `\x` is generally used to signal that, in a string, the next 2 characters should be interpreted as a hex value (this is only necessary when you want to pass these codes to a program / function that supports this encoding, through a string).

If you want to try these escape codes on your terminal you can type them as parameters for the `echo` command. For example:

```
echo "\x1B[10B"
```

This will shift your cursor 10 rows down. If you have messed up your terminal, you can always reset it using the `reset` command.

A complete list of all codes can be found online [1]. Although the standard is already quite old, all terminal programs today support this standardised way of manipulating the text on the terminal.

[1] `https://en.wikipedia.org/wiki/ANSI_escape_code#8-bit`

3.2 Advanced Programming Constructs

This section deals with some more advanced programming constructs that you might need for the bonus assignments. In 3.2.1, you can read about switch statements. 3.2.2 contains information about lookup tables.

3.2.1 Switch-Case Statements

This subsection refers specifically to the switch-statements that you might have already encountered in some high-level programming languages. Below is a pseudocode example:

```
switch(RAX) {
    case 0:
        // case 0 code
        break;

    case 1:
        // case 1 code
        break;

    case 2:
        // case 2 code
        break;
}
```

To implement the `switch`-statement we have to create one small subroutine for each of the cases. We then create a table containing the starting addresses of these subroutines and we use the value of `RAX` to look up the proper subroutine address in the table. A table like this is called a *jump table*:

```
# The jumptable:
jumptable:
    .quad case0sub
    .quad case1sub
    .quad case2sub

# The case subroutines:
case0sub:
    ... # case 0 code
    ret

case1sub:
    ... # case 1 code
    ret

case2sub:
    ... # case 2 code
    ret

# The actual switch statement:
    shlq $3, %rax                # multiply RAX by 8
    movq jumptable(%rax), %rax   # load the address from the table
    call *%rax                   # call the subroutine
```

There is some trickery going on in the last three instructions that deserves some attention: first of all, we have to remember that the subroutine addresses in the jumptable are eight bytes long, so we will have to multiply our `RAX` register by eight before we can use it as a table index. We can of course accomplish this by shifting the operand left by three bits. Second, we have to use the '`*`' when calling a subroutine whose address is located in a register.

3.2.2 Lookup Tables

Very often, programs need to perform time consuming computations inside tight loops. If a small number of values are computed over and over again, we can simply precompute them at compile-time, put them in a table and replace the actual computation with a table lookup. Such a construct is called a *lookup table* and it can be used to simplify and speed up programs considerably. We demonstrate the lookup table through an example:

```
// Print various Fibonacci numbers
for(int i = 0; i < 100000; i++) {
      print(fibonacci(30 + (i % 10)));
}
```

Computing the n-th Fibonacci number is a very computationally intensive task and the `fibonacci` subroutine can be tricky to implement in assembler. By studying the example carefully, we observe that the only values which are actually calculated are `fibonacci(30)` through `fibonacci(39)`. Of course, we can simply precompute these values at compile time without having to implement the `fibonacci()` routine at all. The resulting program is both faster and easier to implement:

```
// A table containing the Fibonacci numbers from 30 to 39
int fibtable[] = {
               832040,
               1346269,
               2178309,
               3524578,
               5702887,
               9227465,
               14930352,
               24157817,
               39088169,
               63245986
    };

// Print various Fibonacci numbers
for(int i = 0; i < 100000; i++) {
      print(fibtable[i % 10]);
}
```

In assembly, we can use the `.byte`, `.word`, `.long` and `.quad` directives to construct the lookup table:

```
fibtable:
      .long 832040
      .long 1346269
      .long 2178309
      .long 3524578
      .long 5702887
      .long 9227465
      .long 14930352
      .long 24157817
      .long 39088169
      .long 63245986
```

3.3 Doing I/O without the C library

During the lab course we have used the system's standard C library to perform input and output operations. Of course, it is also possible to do I/O without the C library. The benefit of this approach is that our programs will be as short and small as possible, as we save ourselves the trouble of executing an extra subroutine call. The drawback, as explained earlier, is that the details of the procedure differ from one operating system to another. As an example, we demonstrate how to print a line of text on the terminal without using `printf`.

The procedure is fairly simple, but as with all topics in this advanced section, it requires some prior knowledge that is slightly outside of the scope of this lab course. During the Operating Systems course, you will learn that programs communicate with the kernel by performing a "system call". In a system call, a program transfers control to the kernel, much like a subroutine call transfers control to a subroutine. In fact, it is possible to pass parameters to a system call in much the same way.

The difference with an ordinary subroutine call is, as always, in the details. In 32 bit, the actual control transfer was achieved by causing a software interrupt, which is sometimes called a *trap*. This is done by executing an `int` instruction. On Linux, the interrupt number for a system call is always `0x80`. In 64 bit mode, things are not that different from what you are used to with C library functions. The arguments still go in the same registers as before, but now you will have to provide a magic number in `%RAX` defining what system call you want. For instance, doing an exit is done by setting `%RAX` to 60, putting the error code in `%RDI` (normally 0) and then using the `syscall` instruction.

```
# Perform the 'sys_exit' system call:
    movq    $60, %rax       # system call 60 is sys_exit
    movq    $0, %rdi        # normal exit: 0
    syscall
```

A complete list of available Linux system calls can be found in the kernel source code [2]. The complete call looks a bit less friendly than `printf`:

```
# define the string and its length:
hello:
    .asciz   "Hello!\n"
helloend:
    .equ     length, helloend - hello

# Perform the 'sys_write' system call:
    movq    $1, %rax        # system call 1 is sys_write
    movq    $1, %rdi        # first argument is where to write; stdout is 1
    movq    $hello, %rsi    # next argument: what to write
    movq    $length, %rdx   # last argument: how many bytes to write
    syscall
```

The code we see here is actually very similar to the code that we find inside the `printf` function itself. Many functions in the C library, including `printf`, use inline assembly code to perform their actual function (see 3.4). Since compilers are not operating system specific, the authors of `printf` have to resort to this technique.
Now that your code does not depend on the C library anymore, you can even assemble and link it yourself, without the `gcc` magic:

```
as -o hello.o hello.s
ld --entry main -o hello hello.o
```

Compare the size of your final program to the size of its C equivalent. Now that's efficiency!

[2]Or in lookup tables like this one: `https://filippo.io/linux-syscall-table/`

3.4 Mixing Assembly Language with C/C++

During the assignments we made frequent use of the subroutines in the system's standard C library. As the name implies, this library was largely written in the programming language C, i.e. we have seen that functions written in the C language are effectively ordinary subroutines that can be called directly from assembler programs. This isn't so strange if you consider that a C compiler actually produces assembler programs as an intermediate step, after which it uses the assembler to generate the actual object code. The obvious next question is: can we call assembler subroutines from C as well? The obvious answer is "yes".

The assembler subroutine is written in the normal manner. To allow the linker to "see" your subroutine, it is necessary to publish its name in the symbol table using the `.global` directive.

In your C source file, you will need to specify a *function prototype* for your assembler subroutine, e.g something like this:

```
/**
 * A function prototype for our assembler subroutine.
 */
int foo(int x);
```

Of course, you should use the same subroutine name in your C program as you do in your assembler file. After this prototype declaration, you can simply call the subroutine like you would call any other C function. The final step is to compile both the assembler file and the C source file and to link them together into a single binary. As usual, we offload all the hard work to `gcc`:

```
gcc -o myprogram myassemblersource.s mycsource.c
```

. . . and presto!

As an aside, it is also possible to include snippets of assembler code directly into your C source code. This can be very useful in cases where specific tight loops in your programs take a lot of time to execute. The details on how to inline assembler code, as this technique is called, are not standardised and they may differ from one C compiler to another. Consult the manual of your favourite compiler to find out how this mechanism works.

3.5 Data Compression: Run-Length Encoding (RLE)

Run-Length Encoding (RLE) is one of the simplest data encoding techniques. RLE is based on the notion of runs, which are sequences of specified length of the same item.

For the bitmap assignment (section 4.6 in the assignment manual), you will use RLE-8, which encodes, in turn, the size of the sequence and each item on 8 bits each. For example, the RLE-8-encoded sequence of two bytes with values `8` and `67` (the ASCII value of `C`) means that the sequence length is 8 and the item is `C`, for the fully decrypted text `CCCCCCCC`. Likewise, the text `ZZZZAAZZZA`, encoded using RLE-8, corresponds to the sequence of bytes `4`, `90`, `2`, `65`, `3`, `90`, `1`, `65`.

Input		Output
x	y	XOR(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.1: XOR truth table.

3.6 Data Encryption: XOR

One of the common logical operations is the eXclusive OR (XOR, $\oplus$), equivalent to the Boolean logic concept of "TRUE if only one of the two operands, but not both, is TRUE". Table 3.1 summarises the truth table of XOR.
Two properties of XOR are of interest:

$$x \oplus 0 = x \tag{3.1a}$$
$$x \oplus x = 0 \tag{3.1b}$$

From Equations 3.1a and 3.1b (non-idempotency), it is trivial to observe that:

$$\begin{aligned}(x \oplus y) \oplus y &= x \oplus (y \oplus y)\\ &= x \oplus 0\\ &= x\end{aligned} \tag{3.2}$$

Equation 3.2 means that we can use XOR to first encrypt ($x \oplus y$) and then decrypt ($(x \oplus y) \oplus y$) a one-bit message x with the encryption/decryption key y. It turns out that these two one-bit operations can be extended to n-bit operations, that is, for an n-bit message M and an n-bit key K:

$$(M \oplus K) \oplus K = M \tag{3.3}$$

For example, if the message is `TEST` in ASCII (`M = 01010100 01000101 01010011 01010100` in binary) and the key is `TRY!` in ASCII (`K = 01010100 01010010 01011001 00100001`), the encrypted text is:

$$\begin{aligned}M \oplus K &= \texttt{01010100 01000101 01010011 01010100}\\ &\oplus \texttt{01010100 01010010 01011001 00100001}\\ &= \texttt{00000000 00010111 00001010 01110101}\end{aligned} \tag{3.4}$$

The decrypted text is:

$$\begin{aligned}(M \oplus K) \oplus K &= \texttt{00000000 00010111 00001010 01110101}\\ &\oplus \texttt{01010100 01010010 01011001 00100001}\\ &= \texttt{01010100 01000101 01010011 01010100}\\ &= M\end{aligned} \tag{3.5}$$

Last, a good example of implementing the XOR encryption technique in C is the "C Tutorial - XOR Encryption" by Shaden Smith, June 2009[3].

[3][Online] Available: `http://forum.codecall.net/topic/48889-c-tutorial-xor-encryption/`.

3.7 Data Representation: the BMP Format (Simplified)

Storing data as images requires complex data formats. One of the simplest is the bitmap (BMP) format, which you must use. BMP files encode raster images, that is, images whose unit of information is a pixel; raster images can be directly displayed on computer screens, as their pixel information can be mapped one-to-one to the pixels displayed on the screen.

The BMP file format consists of a header, followed by a meta-description of the encoding used for pixel data, followed sometimes by more details about the colours used in the image. The BMP file format is versatile, that is, it can accommodate a large variety of colour encodings, image sizes, etc. It is beyond the purpose of this manual to provide a full description of the BMP format, which is provided elsewhere[4].

Luckily for you, of the many flavors of encodings, we opted to only accept one type. Thus, you must use the following BMP format for the bitmap assignment:

1. File Header, encoded as signature (two bytes, `BM` in ASCII); file size (integer, four bytes); reserved field (four bytes, `00 00 00 00` in hexadecimal encoding); offset of pixel data inside the image (integer, four bytes). The file size is the sum between the file header size, the size of the bitmap info header, and the size of the pixel data. The file header size is 14 (two bytes for signature and four bytes each for file size, reserved field, and offset of pixel data). The file size is the sum of 14 (the file header size), 40 (the size of the bitmap header), and the size of the pixel data.

2. Bitmap Header, encoded as[5]: header size (integer, four bytes, must have a value of 40); width of image in pixels (integer, four bytes, set to 32–see see paragraph on barcodes); height of image in pixels (integer, four bytes, set to 32–see paragraph on barcodes); reserved field (two bytes, integer, must be 1); the number of bits per pixel (two bytes, integer, set here to 24); the compression method (four bytes, integer, set here to 0–no compression); size of pixel data (four bytes, integer); horizontal resolution of the image, in pixels per meter (four bytes, integer, set to 2835); vertical resolution of the image, in pixels per meter (four bytes, integer, set to 2835); colour palette information (four bytes, integer, set to 0); number of important colours (four bytes, integer, set to 0).

3. Pixel Data, encoded as `B G R` triplets for each pixel, where `B`, `G`, and `R` are intensities of the blue, green, and red channels, respectively, with values stored as one-byte unsigned integers (0–255). It is important that the number of bytes per row must be a multiple of 4; use 0 to 3 bytes of padding, that is, having a value of zero (0) to achieve this for each row of pixels. The total size of the pixel data is $N_{rows} \times S_{row} \times 3$, where N_{rows} is the number of rows in the image (32–see paragraph on barcodes); S_{row} is the size of the row, equal to the smallest multiple of 4 that is larger than the number of pixels per row (here, 32–see paragraph on barcodes); and the constant 3 is the number of bytes per pixel (24 bits per pixel, as specified in the field "number of bits per pixel", see the Bitmap header description).

[4]BMP file format, Wikipedia article. [Online] Available: `http://en.wikipedia.org/wiki/BMP_file_format`. Note: Although Wikipedia is not a universally trustworthy source of information, many of its articles on technical aspects, such as the "BMP file format" have been checked by tens to hundreds of domain experts.

[5]This encoding is `BITMAPINFOHEADER`, which is a typical encoding for Windows and Linux machines. Older encodings, such as `BITMAPCOREHEADER` for OS/2, are obsolete. Newer versions, such as `BITMAPV5HEADER` exist, but they are too complex for our scientists.