Computer Organization Lab Course Manual

Vrije Universiteit Amsterdam

Edition 2020 - 2021

Contents

Note: references required for completing Computer Organization have been marked with a *

Acknowledgements

The 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 the manual received a refresh and expansion by Otto Visser. Finally in 2014 the decision was made to switch from the x86 to the x86-64 architecture and the manual was edited by Elvan Kula.

Chapter 1

Introduction

1.1 Getting started

Given that this course is given in the second period of the first year, it is quite likely that you have little 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 before starting on the first assignment in Section 2.1. Take special notice of Section 1.3, which summarises and refreshes the prior knowledge that we assume on your part. The assignment texts will frequently refer to the reference (found in Chapter 3), which contains essential information for completing the lab course.

1.2 Lab course rules and etiquette

Teaching assistants will be present during lab course hours to offer you advice and the possibility to have your work reviewed. Please keep in mind that they are *not* there to deal with faulty lab course equipment, problems with your laptop and software, problems with your login account, disputes about deadlines and lab course rules, possible special exceptions (e.g., due to illness on your part) or problems with lab course enrolment. We kindly ask you to bypass the teaching assistants in such matters and to go straight to the lab course coordinator (see table below).

Lab course equipment:	Virtual Machine provided in Lab 0.
Deadlines, rules, exceptions:	Computer Organization lab course coordinator (Tim Hegeman)
Enrolment:	Director of education

Keep in mind that the work you hand in during the lab is subject to certain rules and quality guidelines. The rules and guidelines are listed explicitly for this lab course in Appendix A. Your are expected to check your work prior to handing it in for review. The lab course assistants will only consider work that is in full compliance with the stated rules and guidelines.

1.2.1 Verifying your work

In order for you to get the credit that you passed a lab exercise, you need to have your work checked by a teaching assistant. For this, you will have to submit your work to canvas and then enqueue to the digital queue. One of the assistants will then come and visit you and give you the oppurtunity to explain your work.

1.3 Assumed prior knowledge

This lab course is by no means an introductory course in computer programming. In this subsection we will briefly describe the level of experience that we 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 have followed and understood the lectures and that you have studied 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 have all been 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 six 64-bit general purpose registers (`RAX`, `RBX`, `RCX`, `RDX`, `RDI` and `RSI`) which you can use freely when writing programs. You should know that the x86-64 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. There is a second set of eight general purpose 64-bit registers named R8-R15.

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 assembly languages, machine language and high-level programming languages by carefully restating their 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:

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

```
 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, 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

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

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.

Writing large programs in assembly language still has its drawbacks unfortunately. 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.

1.4 Schedule

During the lab course you will learn the basics of writing programs in assembly language. You will learn how to call functions and what recursion looks like.

- Assignment 0 + study of paragraph 3.1 (at home)
- Learn programming environment and do assignment 1 (2 hours)

[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.

- Assignment 2 (2 hours)
- Assignment 3 (4 hours)
- Assignment 4 (8 hours)
- Assignment 5 (optional) (6 hours)
- Assignment 6 (optional) (8 hours)
- Assignment 7 (optional) (8 hours)
- Assignment 8 (optional) (at least 8 hours)
- Assignment 9 (optional) (at least 8 hours)

You will *need* your time for this lab course, it is not easy. Many students underestimate the lab, do not start when they should and find out that they cannot complete it anymore too late. Do not let this happen to you and make sure you visit every session, so you can talk about your questions to the assistants.

1.5 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 assembler. 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 on-line 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.

You should now be mentally prepared to start the assignments. Good luck!

Chapter 2

Assignments

Please note: all the assignments marked "extra" will only provide points iff[1] you have assignment 1–4 checked (but you do not need to do Assignment 7 before you can do 8; no worries)! It is also worth noting that you can not get points for doing more than one of the Assignments 5 a–c, you have to choose one of them.

2.1 Assignment 0: a detailed example

In this assignment you will study the development process and the implementation of a simple, non-trivial example program. In the subsequent exercises you can borrow ideas from this example for your own programs, but for now, the most important thing is that you will learn:

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

Since this is an introductory exercise, we will start with a very simple example problem. We are going to develop a program that prints all prime numbers below 1000. The algorithm that we will use to solve this problem was developed by an old scholar who went by the name of Eratosthenes of Cyrene. Eratosthenes lived in northern Africa from 276 B.C. to 194 B.C. and he devised what is probably the oldest known algorithm for finding prime numbers: the *Sieve of Eratosthenes*. The algorithm is not terribly efficient but it is very easy to understand, which is exactly why we use it here.

We will start by describing the Sieve algorithm 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 of the algorithm

The Sieve algorithm is quite simple. We start by constructing a list of all numbers between 2 and some upper limit, which in our case is the number 1000:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 ... 997 998 999 1000
```

We then take the first element of the list (the number 2, which is a prime!) and remove all multiples of this number from the list, except for 2 itself:

[1] no, this is not a spelling error; see `http://en.wikipedia.org/wiki/If_and_only_if`

```
2 3 . 5 . 7 . 9 . 11 . 13 . 15 . 17 . 19 . 21 ... 997 . 999 .
```

We then continue with the next element after 2 which is still in the list. This number is 3, again a prime number, and we remove all multiples of 3 except 3 itself:

```
2 3 . 5 . 7 ... 11 . 13 ... 17 . 19 ... 997 ...
```

We continue this process with the next number in the list (5, again a prime), etc. until we reach the end of the list. You should be able to convince yourself that the remaining numbers are all the prime numbers below 1000.

Step 2: specification of the algorithm

Now that we are familiar with the basics of Eratosthenes's Sieve, we will transform the algorithm into a formal specification. Why do we have to do this? Well, because there are many small, practical details that we need to consider before we can actually implement the algorithm. One such problem is the problem of representation: how will we represent the list of numbers in our program? Will we use a *linked list*[2] structure containing the numbers or is it better to store the numbers in a list of fixed size? 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 sieve program below.

Instead of maintaining a list of remaining numbers, we will maintain an array of Boolean values to denote which numbers are still present and which have been crossed out. All entries in this array are initialised to `true`, since all numbers are initially present. We remove a number from the list by setting its corresponding table entry to `false`. Here is the pseudocode that describes the algorithm:

```
main() {
    bool numbertable[1000];

    // Initialise the number table
    for(int i = 0; i < 1000; i++)
        numbertable[i] = true;

    // The sieve algorithm
    for(int number = 2; number < 1000; number++) {
        // If the number is still in the list it must be prime
        if(numbertable[number]) {
            // Print the prime number
            print(number);

            // Cross out all multiples of the number
            int multiple = 2 * number;
            while(multiple < 1000) {
                numbertable[multiple] = false;
                multiple += number;
            }
        }
    }
}
```

The pseudocode above uses only simple operations on simple data types (integers, booleans) and basic programmatic constructs such as for-loops and if statements. These constructs can easily be translated to assembler programs, as we shall see in the next step.

[2] `http://en.wikipedia.org/wiki/Linked_list`

Step 3: implementation

The final step in our development process is to translate the specification into working assembler code. We present the complete implementation of the sieve program in working x86-64 assembler on the next page, followed by a detailed explanation of the code. In later exercises, you can use this program as a template for your own work. Try to read along and try to 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 first programs that you will be asked to write will be much simpler than this one.

```
# ****************************************************************************
# * Program name : sieve                                                     *
# * Description  : this program prints all the prime numbers below 1000      *
# ****************************************************************************
.bss
NUMBERS:    .skip 1000          # memory space for the number table

.text
formatstr:  .asciz "%d\n"       # format string for number printing

.global main
# ****************************************************************************
# * Subroutine  : main                                                       *
# * Description : application entry point                                    *
# ****************************************************************************
main:
        pushq   %rbp                # store the caller's base pointer
        movq    %rsp, %rbp          # initialize the base pointer

# Initialize the number table:
        movq    $0, %rbx            # initialize 'i' to 0.
loop1:
        movb    $1, NUMBERS(%rbx)   # set number table entry 'i' to 'true'
        incq    %rbx                # increment 'i'
        cmpq    $1000, %rbx         # while 'i' < 1000
        jl      loop1               # go to start of loop1

# The sieve algorithm:
        pushq   $2                  # initialize 'number' to 2 on stack
loop2:
        movq    -8(%rbp), %rbx      # load 'number' into a register
        cmpb    $1, NUMBERS(%rbx)   # compare NUMBERS[number] to '1'
        jne     lp2end              # if not equal, jump to end of loop 2
        movq    $formatstr, %rdi    # first argument: formatstr
        movq    %rbx, %rsi          # second argument: the number
        movq    $0, %rax            # no vector arguments
        call    printf              # print the number
        movq    -8(%rbp), %rbx      # 'multiple' := 'number'
        shlq    $1, %rbx            # multiply 'multiple' by 2

loop3:
        cmpq     $1000, %rbx        # compare 'multiple' to 1000
        jge     lp2end              # goto end of loop2 if greater/equal
        movb    $0, NUMBERS(%rbx)   # set number table entry to 'false'
        addq    -8(%rbp), %rbx      # add another 'number' to 'multiple'
        jmp     loop3               # jump to the beginning of loop 3
lp2end:
        movq    -8(%rbp), %rbx      # load 'number' into a register
        incq    %rbx                # increment 'number' by one
        movq    %rbx, -8(%rbp)      # store 'number' on the stack
        cmpq    $1000, %rbx         # compare 'number' to 1000
        jl      loop2               # if smaller, repeat loop2

end:
        mov     $0, %rdi            # load program exit code
        call    exit                # exit the program
```

Program explanation

We will now discuss the implementation in some detail.

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 `.bss` and `.text` directives on lines 5 and 8 tell the assembler to put all the subsequent code in a specific *section*. Other assembler directives, like `.global` on line 11, make certain labels visible to the outside world. Study the description of these assembler directives in paragraph 3.5.3.

Instructions

The example program uses a number of commonly used instructions, such as `mov`, `push`, `cmp` and `jmp`. You can look up the exact meaning and function of these and other useful instructions in paragraph 3.5.2.

The beginning of the program

At the beginning of the program, which actually starts at `main` on line 19, we see that the base pointer is being initialised. This opaque ritual has to be performed at the start of each subroutine, including the `main` subroutine. You will learn all its secrets when we discuss subroutines and the stack in Assignment 1.

Input and Output

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

```
movq $formatstr, %rdi    # first argument: formatstr
movq %rax, %rsi          # second argument: the number
call printf              # print the number
```

This deserves a thorough explanation. First of all, doing input and output in a computer program is not really a trivial job. It involves a call to the kernel and the gory details on how to do this differ from operating system to operating system. Luckily, there is an easy way to circumvent all these difficulties and that is by using the IO functions in the operating system's standard C library[3]. Calling functions in the standard C library is no different from calling subroutines in your own programs. We will discuss this mechanism in the next assignment.

Registers, variables and the stack

The example program uses a number of variables to store data. On lines 20 through 24 it uses the loop counter `i` and on line 28 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 `i`, 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 store their values on the stack, as we see with the `number` example: it is created by pushing the number 2 onto the stack on line 28, but later we load it into a register (line 30) when we need to check its value.

[3]Some of you will probably be curious about how to do it the hard way. We appreciate and encourage your curiosity, so we provided the details in the appendix, Section 3.6

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. We will delve into stack details in one of the exercises.

Programming constructs (if/else, for, while, etc.)

Our pseudocode program is defined in terms of familiar program constructs, such as `for` loops and `if` statements. In assembly, we do not have such high level constructs, but we do have a number of standard tricks to achieve similar results. These tricks are all listed and explained in Section 3.2. You can use this information to translate your pseudocode into assembly language systematically.

The end of the program

At the end of the 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.

2.2 Assignment 1: your first simple program

In this assignment you will be asked to write your first assembly program. You will have to use the knowledge you acquired from assignment 0 in order to complete this task, so make sure you have a thorough understanding of the example program. Remember that you can always ask the lab course assistants for help. For this program, you will not have to write any specifications, since there is no significant algorithmic complexity involved. However, you are of course required to write proper comments.

In order to complete this assignment you will need to call the `printf` subroutine. Paragraph 3.3.1 of the reference section explains the details of calling subroutines and paragraph 3.4.1 explains how to use the `printf` subroutine. Section 3.1 explains the commands that you will need to enter on your shell in order to build and run your program.

Exercise:
Create a new text file, called "hello.s". Implement a simple "main" routine that exits the program immediately with the proper exit code and without crashing. Build your program and run it. Alter your main routine in such a way that it prints a message containing your names, study numbers and the name of the assignment on the terminal. You should not need more than one call to `printf` to display your message. After completing the rest of the exercises in part 1, you will need to have the source code of this program checked by the teaching assistants, so make sure you keep all your files in order.

2.3 Assignment 2: subroutines and I/O

In this assignment you will write your first simple subroutine. You will also learn how to obtain input from the user. The `scanf` subroutine from the C standard library can be used to obtain input from users. It is discussed in paragraph 3.4.2 of the reference section. Paragraph 3.3.2 explains how to define your own subroutines.

Exercises:

1. Copy your "hello.s" program into a new file, called "inout.s". Alter the message in such a way that it prints the correct assignment name ("Assignment 2: inout"). Create a subroutine called "inout" and call it from within your main routine right after your message is displayed. At first, your subroutine should simply return immediately. Build and test your program.

2. Alter your "inout" subroutine in such a way that it asks the user for a number, increments the number by one and prints the result on the terminal. Build and test your program and save it for later approval.

2.4 Assignment 3: parameter passing and local variables

Now that you have played around with some simple subroutines it is time to gain a more complete understanding of this important programming construct. In this assignment you will write a subroutine that takes several input parameters and returns a computed value. If you have not already done so, study the remainder of paragraph 3.3.2, which discusses the techniques and conventions surrounding subroutine parameter passing and return values.

Exercises:

1. The following partial specification of the `pow` subroutine is given:

```
/**
* The pow subroutine calculates powers of natural bases
* and exponents.
*
* Arguments:
*
*    base - the exponential base
*    exp  - the exponent
*
* Return value: 'base' raised to the power of 'exp'.
*/
int pow(int base, int exp) {
        int total = 1;

        //...

        return total;
}
```

   Complete the specification of the `pow` subroutine. You should only use looping constructs and simple arithmetic operations to compute the total.

2. You may have your specification checked by one of the lab course assistants. The lab course assistants can prevent you from implementing the wrong algorithm. If you are confident your specification is correct, you don't have to ask an assistant to check and you may immediately proceed with the next step.

3. Write a program called "power" which contains an implementation of your `pow` subroutine. The `main` routine should ask the user for a positive base and exponent. The program should then calculate the resulting power using the `pow` subroutine and print the return value.

2.5 Assignment 4: recursion

By now, you should have a fairly thorough understanding of the stack mechanism and of its uses (e.g., storing local subroutine variables). In this assignment you are going to write a recursive

subroutine that calculates the factorial of a number ("n!"). A recursive subroutine means that the subroutine will call itself, usually with a different parameter to calculate a part of the final solution. This subroutine will be about 14 instructions in length when it is finished, but writing it will be fairly difficult. Do not be discouraged if it takes you a few hours to get it working.

Exercise

1. Copy your "inout" program to a new file called "factorial.s". Create a new subroutine called "factorial". This new subroutine should take one parameter, n, and for now it should simply do nothing and return n in the RAX register. Alter your inout subroutine in such a way that it calls factorial with the number it reads, instead of incrementing it by one. It should print the result of factorial on the terminal.

2. Write a pseudocode specification of your factorial subroutine. The subroutine accepts a parameter n and it should return n!. Make sure your algorithm is recursive. It should not need to be more than a few lines of pseudocode.

3. You may have your specification checked by one of the lab course assistants. The lab course assistants can prevent you from implementing the wrong algorithm. If you are confident your specification is correct, you don't have to ask an assistant to check and you may immediately proceed with the next step.

4. Implement your factorial routine. Test your program thoroughly.

This exercise wraps up the basic assembler programming assignments. You should go and have your code checked by a lab course assistant. Well done!

2.6 Assignment 5a: (extra) implementing "diff" in assembly (500 points)

For this exercise you will implement the Unix "diff" program in assembly. The purpose of the diff program is to compare two files line by line and show all the differences between them. As a sample output, consider the following two files:

```
Hi, this is a testfile.
Testfile 1 to be precise.
```

```
Hi, this is a testfile.
Testfile 2 to be precise.
```

The resulting output of diff is then:

```
$ diff testfile1 testfile2
2c2
< Testfile 1 to be precise.
---
> Testfile 2 to be precise.
```

As you can see it tells us that the second line is different (2c2 means that line 2 in the original has been **c**hanged to become line 2 in the new file). For more information on how the diff command works, please check the diff manuals (type `man diff` in a terminal) and check Wikipedia at `http://en.wikipedia.org/wiki/Diff`.

For this assignment your program will have to be able to do the following:

- Implement a line-by-line comparison version of diff. This means it is not required to have the `a` and `d` outputs that the real diff offers. Only the changes will suffice, though we encourage you to also try the detection of addition and deletion of lines.

- Implement at least the -i and -B options that diff offers (see the diff manual to learn what these options do).

Note : It is not required that you read text from a file or the standard input. It is allowed for you to hardcode the texts you are comparing in your source. If you do this however, the student assistants will change this hardcoded text to confirm that your program works for different texts as well.

2.7 Assignment 5b: (extra) implement a simplified printf function (500 points)

As mentioned before, the `printf` subroutine is just a subroutine like any other. To prove this, you will write your own simplified version of `printf` in this assignment.

Exercise:

Write a simplified `printf` subroutine that takes a variable amount of arguments. The first argument for your subroutine is the format string. The rest of the arguments are printed instead of the placeholders (also called format specifiers) in the format string. How those arguments are printed depends on the corresponding format specifiers. Your `printf` function has to support any number of format specifiers in the format string. Any format specifier after that may be printed without modification.

Unlike the real `printf`, your version only has to understand the format specifiers listed below. If a format specifier is not recognized, it should be printed without modification. Give your `printf` function a different name (e.g. `my_printf`) to avoid confusion with the real `printf` function in the C library. Please note that for this exercise you are not allowed to use the `printf` function or any other C library function. This means you will have to use system calls for the actual printing. Your function must follow the proper x86_64 calling conventions.

Supported format specifiers:

%d Print a signed integer in decimal. The corresponding parameter is a 64 bit signed integer.

%u Print an unsigned integer in decimal. The corresponding parameter is a 64 bit unsigned integer.

%s Print a null terminated string. No format specifiers should be parsed in this string. The corresponding parameter is the address of first character of the string.

%% Print a percent sign. This format specifier takes no argument.

Example:

Suppose you have the following format string:

> My name is %s. I think I'll get a %u for my exam. What does %r do? And %%?

Also suppose you have the additional arguments "Piet" and 10. Then your subroutine should output:

> My name is Piet. I think I'll get a 10 for my exam. What does %r do? And %?

Hints

To get started you may divide the work in a number of steps. Note that these are just hints, you do not have to follow these steps to finish this assignment.

1. Write a subroutine that prints a string character by character, for example by using `putchar`.
2. Modify the subroutine to recognize format specifiers in the format string. Initially, you can discard the format specifiers rather than process them. Characters that are not part of a format specifier can be printed as before.
3. Implement the various format specifiers. It may help to implement %u before %d.
4. It may help to store all input argument registers on the stack at the start of your function, even if you don't end up using them.

2.8 Assignment 5c: (extra) implement a hashing function (500 points)

For this assignment, implement a program to calculate a hash such as SHA-1, SHA-256, MD4, MD5, or any other hash you like, of the input given to the program. If you do not know what a hash function is, Google a bit first.

Exercise:
Choose a well known hashing algorithm, and write a program that calculates and prints the hash of the input given to the program. (From standard input or from a file.)

Optional help from our side for SHA-1:
Read the Wikipedia page about SHA-1[4] (especially the pseudo code). As you will see, the algorithm has to split up the data up in 512-bit chunks, and then process each chunk separately. In this simplified exercise, you will only implement the function to process a chunk. The rest of the code is provided by us.

Our part of the code can be downloaded from Canvas. You can combine this with your own code by adding "`./sha1_test64.so`"[5] as another parameter to `gcc`. Our code does everything until and including the line "break chunk into sixteen ..." in the pseudo code on Wikipedia. The command used to compile your code could like something like this:

```
gcc -o test my_sha1_chunk.s ./sha1_test64.so -no-pie
```

Your part of the code should not have a `main` function, but instead have a `sha1_chunk` function, which will be called by our part of the code. This `sha1_chunk` function takes two parameters: First, the address of `h0` (`h1`, `h2`, etc. are stored directly after `h0`). (See Wikipedia's pseudo code for SHA-1 for these names.) Second, the address of the first 32-bit word of an array of 80 32-bit words. The first sixteen of this array are set to the sixteen 32-bit words the chunk contains (which are called `w[0]` till `w[15]` on Wikipedia). Your function should modify `h0` till `h4` as described in the pseudo code.

When you execute the combined program, our part of the code prints a lot of information on what is happening, and when your function is called. It displays the result of your function, and whether that is correct or not. You can of course print more debugging information from your own function using `printf`.

[4] `http://en.wikipedia.org/wiki/Sha1`
[5] there is also a sha1_test32.so for 32 bit compilation available

2.9 Assignment 6: (extra) using assembly to hide data in a bitmap (750 points)

On April 2, 1990, Dilbert was sent to the tiny country of Elbonia on a secret mission[6]: to infiltrate the formerly communist, newly capitalist country and find targets who can become Western engineers. Secretly, Dilbert was also charged with becoming a liaison to the resistance movement from within Elbonia's totalitarian neighbour, North Elbonia. It has been over three decades, and the mission is still on ... with a twist. The governments of Elbonia and North Elbonia have joined forces and together tightened their grip on information. Facebook, Twitter, YouTube have been banned. Searches on `google.com` are often filtered out or redirected to `NorthElbonia-SearchAndTortureDept.com`.

You are the new tech assistant of Dilbert. Your mission: using assembly, encrypt messages into the photos of Elbonian white noise that Dilbert provides for you. In detail:

1. Make sure you understand the message.
2. Compress the message using the Run-Length Encoding (RLE) technique.
3. Prepare the white noise image.
4. Use XOR to encrypt the message into the white noise image.
5. Save the results as image bitmaps, in BMP format.
6. Test that you can decrypt the message, using again the XOR encryption technique.

More details follow. This message will not self-destruct in 5 seconds.

Data: The Message

Complexity: Easy
To demonstrate that your assembly code works, encrypt and decrypt the message `Reading Dilbert strips or encoding Elbonian messages are not good excuses for failing the TI1406 final exam.`, including the final dot ('.'). Save the message as an assembly data chunk before you continue.

Each message, before encryption, must be preceded and followed by the pattern $8 \times T, 4 \times I, 2 \times 1, 2 \times 4, 4 \times 0, 8 \times 5$, where `<number>` × `<character>` means that *character* is repeated *number* times, e.g., $8 \times T$ means `T T T T T T T T`. The added parts are called the "lead" and the "trail" of the message.

Data Compression: Run-Length Encoding (RLE)

Complexity: Moderate
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 this assignment, 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 `8T` means that the sequence length is 8 and the item is `T`, for the fully decrypted text `T T T T T T T T`.

You have to devise your own algorithms for RLE-8-encoding and RLE-8-decoding the message described in the previous paragraph.

[6] See `http://www.dilbert.com/fast/1990-04-02/` and `http://www.dilbert.com/fast/1990-04-03/`.

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

Table 2.1: XOR truth table.

Data: Repeating White Noise Patterns

Complexity: Easy
Since Dilbert's photo camera has been confiscated, we cannot provide you with pictures of anything Elbonian. However, Elbonians are very proud of their national art, which is based on white noise rectangles, that is, rectangles filled with a seemingly random collection of white and black pixels. You are to implement a generator of Elbonian white noise.

Be warned, to show that in Elbonia even white noise is controlled, the Elbonian Ministry of Mud approves every year a single pattern as that year's white noise. The pattern can be seen in the numerous works of poorly printed art on the streets. Dilbert believes this year's pattern is:

`W W W W W W W W B B B B B B B B W W W W B B B B W W B B B W W`

(31 pixels long), where `W` means white and `B` means black.

Create a sequence by repeating the Elbonian white noise pattern, followed by a red pixel, 32 times. This will form a 32×32 image, where each pixel is either white, black, or red.

Data Encryption: XOR

Complexity: Moderate
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 2.1 summarises the truth table of XOR.
Two properties of XOR are of interest:

$$x \oplus 0 = x \tag{2.1a}$$
$$x \oplus x = 0 \tag{2.1b}$$

From Equations 2.1a and 2.1b (non-idempotency), it is trivial to observe that:

$$\begin{aligned}(x \oplus y) \oplus y &= x \oplus (y \oplus y) \\ &= x \oplus 0 \\ &= x\end{aligned} \tag{2.2}$$

Equation 2.2 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:

$$(M \oplus K) \oplus K = M \tag{2.3}$$

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:

$$\begin{aligned}M \oplus K &= \texttt{01010100 01000101 01010011 01010100} \\ &\oplus \texttt{01010100 01010010 01011001 00100001} \\ &= \texttt{00000000 00010111 00001010 01110101}\end{aligned} \tag{2.4}$$

The decrypted text is:

$$\begin{aligned}(M \oplus K) \oplus K &= \texttt{00000000 00010111 00001010 01110101} \\ &\oplus \texttt{01010100 01010010 01011001 00100001} \\ &= \texttt{01010100 01000101 01010011 01010100} \\ &= M\end{aligned} \tag{2.5}$$

If the size of the message is m bits and the size of the key is $k << n$ bits, the key can be repeated. Effectively, using the full k bits of the key `K`, first the first k bits of the message `M` are encrypted, then the next k bits, etc.

Last, a good example of implementing the XOR encryption technique in C is the "C Tutorial - XOR Encryption" by Shaden Smith, June 2009[7].

Data Representation: the BMP Format (Simplified)

Complexity: Difficult

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 (look-up table, see also the paragraph on white noise). 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[8].

Luckily for you, of the many flavors of encodings, Elbonian authorities only accept one type. Thus, you must use the following BMP format for this 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[9]: 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 white noise); height of image in pixels (integer, four bytes, set to 32–see paragraph on white noise); 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

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

[8] 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.

[9] 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 Elbonian engineers.

(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 white noise); 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 white noise); 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).

Last, but not least

You should go and have your code checked by a lab course assistant. You have now officially proved mastery in the basics of assembly programming. Not bad!

2.10 Assignment 7: (extra) implement an esoteric language interpreter (700–1000 points)

For this assignment, implement an interpreter for a very simple "programming language". We recommend you to chose a very simple esoteric programming language such as Brainfuck or Whitespace. In Brainfuck, for example, every character of the source code is a command, and there are only eight very simple commands. The interesting thing about this language is that even though it might not seem like it, it is still Turing complete. (This basically means that you can write every possible program in it, but not necessarily in an efficient way.)

Exercise:
Choose some basic programming language (we recommend Brainfuck; `http://esolangs.org/wiki/Main_Page` is a great place to look for similar simple and/or fun languages) and write a program in assembly that can execute a program in that language. Your program should read the source code of the interpreted program from from a file, given as a command line argument to your interpreter.

You can find a tarball containing assembly code and instructions for reading a file specified as a command line argument on Canvas.

Example:

Suppose you have written a Brainfuck interpreter. You have a file called `hello.b` containing the following code:

```
>+++++++++[<++++++++>-]<.>+++++++[<++++>-]<+.+++++++..+++.>>>++++++++[<++++>-]
<.>>>++++++++++[<+++++++++>-]<---.<<<<.+++.------.--------.>>+.
```

Now, executing your program (assuming it is called `brainfuck`) as follows

```
./brainfuck hello.b
```

should make it give

```
Hello World!
```

as output.

More complex programs to test your interpreter with can be found all over the internet. (For Brainfuck, see e.g. `http://esoteric.sange.fi/brainfuck/utils/mandelbrot/mandelbrot.b`.)

Bonus points: (up to 3 × 100 points)

Find another interpreter for the same language, and compare the speed of your interpreter against it. You will probably notice a big difference in speed. What is causing that?

Find ways to significantly improve the speed or memory usage of your interpreter. Depending on the optimisations you make, you can get at most 300 points extra, making a grand total of 1,000 for this assignment.

2.11 Assignment 8: (extra) using assembly to implement a game (1,000 points)

For this exercise you will use your basic assembler programming skills to develop a complete and useful program: a game. For the purpose of this assignment, a game is an interactive computer application in which at least one user (player) influences the outcome via keyboard or mouse input.

You are free to choose any game, except for simple text-based games such as "Guess a number" or Trivia. The game you choose does not have to be overly complex either; we suggest implementing a (single-player) Pong game, where the player controls a paddle and tries to prevent the ball from crossing the player's goal line, or something of comparable difficulty. Before you start programming, we encourage you to ask a lab assistant whether they think your idea is reasonable and sufficient.

Your task is to implement a game, subject to the following requirements:

Requirements

1. Your implementation should implement correctly the rules of the game.
2. Your implementation should display the state of the game. If the rules of the game make it possible, your implementation should display the current progress of the player toward achieving the goal of the game.
3. Your implementation should permanently record the top scores, if the rules of the game allow it. Your implementation should also have an option to display them. A bootable game only needs to store top scores until the next reboot (i.e. they do not need to be written to disk).
4. Your implementation should be able to receive input from at least one player. The input has to come either from the keyboard or from the mouse.

Notes and Hints about solving this assignment

Important: For this extra assignment you should NOT expect content-related help from the lab assistants, that is, you should not expect the lab assistants to debug your code or even to suggest how you should solve the assignment. Instead, this assignment allows you to demonstrate your ability to go beyond the manual (and lab assistant).

Hint #1: Although we are sure you know how to use a search engine and understand the keywords you need to search for this assignment, we would like to give you a hint for writing games in assembly. For the past 15 years, the demo scene and several generations of students around the world have learned much from `Denthor's Asphyxia Tutorials`, `http://archive.gamedev.net/archive/reference/listed82.html?categoryid=130`. In particular, you may be interested in graphics (tutorial `Mode 13h`, `http://archive.gamedev.net/archive/reference/articles/article347.html`).

Hint #2: Bootable game Another option is to make your game bootable, so that it can run without any operating system. This means that you will have access to the graphical (VGA) memory directly, implement interrupt handlers to work with timing and input, and basically be

able to (but also have to) do everything yourself. There are no standard libraries either. (So no printf, no exit, no nothing.) These advantages may also be disadvantages.

You do not need to start from scratch in implementing a bootable game. You can use a simple library that we[10] made and that will set up most of the basic things for you. This library switches the processor to 32 bit mode, makes sure you have a stack, and lets you set interrupt handlers. After that, you are basically on your own. Again, many advantages, but these can turn easily into disadvantages.

The "bootlib" library can be found online, at `https://m-ou.se/p/bootlib`. It contains a README (which you should read), and an example which you should try. We recommend you work on some Linux distribution, as it is not quite easy to get the compiler/linker working right on Windows or Mac.

Last but not least

You should go and have your code checked by a lab course assistant. You have now officially proved excellence in mastering the basics of assembly programming. Congratulations, you have completed the most difficult part of the lab!

2.12 Assignment 9: (extra) Intro to HPC: implement a memory bandwidth benchmark (500–750 points)

For this assignment, you are required to implement using x86-64 assembly, a simplified (i.e., single-core) version of the STREAM (`http://www.cs.virginia.edu/stream/ref.html`) benchmark. This benchmark measures the memory bandwidth of the computer using simple array operations (kernels).

Considering A, B, C are arrays of length N, and q is a scalar, the four STREAM kernels are the following:

- **COPY:** $A[i] = B[i]$, for $i \leq N$
- **SCALE:** $A[i] = q \cdot B[i]$, for $i \leq N$
- **ADD:** $A[i] = B[i] + C[i]$, for $i \leq N$
- **TRIAD:** $A[i] = B[i] + q \cdot C[i]$, for $i \leq N$

Exercise: For this assignment, your goal is to implement the four STREAM array operations in x86-64 assembly and compare its performance with the default (single-core) STREAM implementation. The input arrays should consist of 64 bit unsigned integers. Is the performance better/worse? Why?

Basic requirements (500p):

- Implement the four array operations.
- Find N large enough such that the data does not fit into the CPU cache (i.e., if the array is too small, the bandwidth you measure may be the cache bandwidth).
- Measure the running time of each operation (for this you may use a system call).
- Run each operation 20 times and report average running time.
- Based on the runtime, report the **best** achieved memory bandwidth.

Extra requirements (250p):

[10] Lab assistants Maurice Bos and Maarten de Vries are the authors of the bootable game library.

- Vectorize the previously designed code using Intel AVX, or your SIMD instruction set of choice.
- Compare its achieved performance with the basic version. Does it perform better? Why?

Chapter 3

Reference documentation

This section contains all the reference material and background information that is needed to complete the lab course. Throughout the assignments parts 1, 2 and 3 you will frequently be referred to parts of this additional documentation. This chapter is by no means a complete reference, but should contain all the information needed to complete at least the compulsory part of the assignments.

3.1 Building and running programs*

The short story: open a terminal window, navigate to the directory that contains your sources and run the following commands:

```
gcc -o nameofyourprogram nameofyoursource.s -no-pie
./nameofyourprogram
```

Please note that the last parameter is only valid if you are executing this command on the Virtual Machine provided by us.

The longer story: in order to create an executable program out of your assembly source code you need to *assemble* it using a tool called `gas`, the GNU assembler. This creates a so called *object file*[1]. Since your actual program may consist of subroutines that are defined in different object files and libraries, the resulting object file(s) needs to be linked into an executable using the tool `ld`, the linker. Unfortunately, it is also necessary to include a host of other files in the linking process, such as the standard C library and the C runtime environment, to produce a proper Linux executable. The exact files to link may differ from one Linux distribution to another and the list may become rather long, which is why we do what every sane person does: we swallow our pride and cheat by simply using `gcc`, the GNU compiler collection, to call `gas` and `ld` on our behalf. If your are curious as to how bad the actual calls look, try using `gcc` in verbose mode with the `-v` flag. You will see that we did not succumb without battle:

```
gcc -v -o nameofyourprogram nameofyoursource.s -no-pie
```

3.2 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 3.2.1. In paragraphs 3.2.2 through 3.2.4 we provide examples of actual programming constructs.

[1] This has nothing to do with objects in the Object Oriented Programming sense.

3.2.1 Conditional branching*

There are many so-called "jump" or "branch" instructions on the x86 which load a new value into the program counter. These instructions come in two flavors. First, there are the normal 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 first operand of the `cmp` instruction was greater than the second. For an overview of the various conditional branch instructions, see the instruction set reference in paragraph 3.5.2. The subsequent paragraphs will demonstrate other programming constructs based on the conditional branch instructions. Paragraph 3.2.2 will give a more compact implementation of the `if`-statement.

3.2.2 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:

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

    elsecode:
        ...           # ELSE-code
        jmp end

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

3.2.3 While/do-while/for loops*

The `do-while`-loop Pseudocode:

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

In this example we jump back to the beginning of the loop as long as the condition holds. Implementation-wise, this is the simplest type of loop:

```
    loop:
        ...           # loop code

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

The `while`-loop Pseudocode:

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

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:

```
    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 Pseudocode:

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

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

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

        RAX++;
    }
```

You should be able to implement this one yourself.

3.2.4 Switch-case statements

Pseudocode:

```
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:
        shl $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.

3.2.5 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 Subroutines*

In this subsection we will introduce the concept of *subroutines*. A subroutine is the generic name for a sequence of instructions with a well defined start and end and a well defined purpose. Java methods, C functions and Pascal procedures are all examples of subroutines in higher level languages. In the subsequent paragraphs we will explain how to call subroutines and how to define them.

3.3.1 Calling subroutines*

A subroutine is simply a block of instructions which starts at some memory address. If we want to execute or *call* a subroutine, we simply need to jump[2] 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:

[2] 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.

```
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. After executing all instructions in the `somesub`-subroutine, execution will simply return to- and continue at the first instruction after the `call` instruction.

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[3] we will use registers for this purpose. More specifically, we will place the arguments in the following registers:

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

If you have more arguments, than the remaining arguments need to be pushed to the stack in reverse order (first argument pushed last).

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
```

Note that `foo` might use these registers as well, so the value 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); this is called: caller saved. 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`, `%R12`, `%R13`, `%R14` and `%R15`, all other registers are caller saved and may be modified by subroutines. Note that if you needed to use the stack, that `foo` will *not* remove the arguments from the stack, so you should pop them off yourself after the call returns.

Furthermore: 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. If you are not using the stack for function arguments, 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.

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.

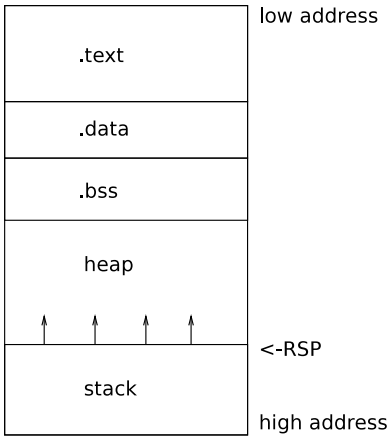
[3]This convention is the so called "C calling convention" and if we adhere to it our subroutines and calls will be fully compatible with the system's standard C library. You can find the full documentation of the calling conventions here: `https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf`

3.3.2 Writing your own subroutines*

As we have seen in the previous paragraph, 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. One important part of the calling conventions is the x86-64 stack and how you can use it. This will be explained first. Afterwards, we will implement the `foo` subroutine from the previous paragraph.

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.

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



3.3.3 Cleaning up the stack*

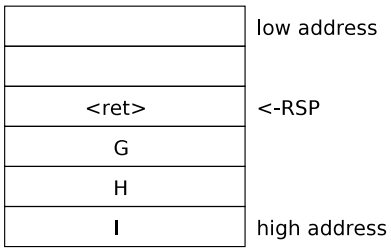
Of course, if the stack grows too large, it will eventually overwrite your program's code and data. This is called a *stack overflow* and it is usually indicative of a serious design flaw in your program. To avoid this problem the caller must clean up the stack after every function call by adding the parameter-block size to the stack pointer directly. 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
movq ..., %...         # The other 5 arguments to their registers
movq $formatstr, %rdi  # The format string
movq $0, %rax          # no vector arguments for printf
call printf            # Print the numbers
addq $8, %rsp          # Clean the stack (magic number)
```

3.3.4 The foo subroutine

If we want to implement a version of the `foo` subroutine that accepts more than 6 arguments, we will need to know where to find the extra arguments from inside the subroutine. We will also need to know where the return address resides on the stack, so that we can safely return control to the caller after our subroutine finishes. To understand how this is done, we will take a look at the stack right after someone calls our `foo` routine. Remember that the caller pushes the extra arguments

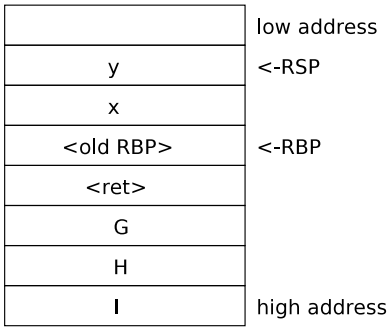
in reverse order before executing the `call` instruction? Remember also that the `call` instruction automatically pushes the return address? Well, if you do, it should not be too surprising that the stack will look like this by the time control reaches the first instruction in our subroutine:



With this knowledge under our belts we can start to envision our first simple subroutine. First and foremost, a subroutine is a block of instructions which has a *label*. Second, it should pop the return address off the stack and load it into the program counter (i.e. "jump") once it finishes. Being ever so considerate of our lazy nature, the x86 designers provided the `ret` instruction to do exactly that: it pops an eight byte value off the stack and loads it into the program counter. Thus, our first implementation of `foo` simply looks like this:

```
foo: ret # return from subroutine
```

This implementation of `foo` does very little of course. It ignores the arguments and it returns immediately without setting a proper return value in `RAX`. A more interesting routine would do some calculations and return a value. It might in turn call other subroutines. It is very likely that the subroutine will also push some values onto the stack during its lifetime. 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 the 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. The process of storing the `RBP` and copying the stack pointer is called the *subroutine prologue* and it is the same for all subroutines you write. Below 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):



In addition to some parameters `G`, `H` and `I`, we can see two local variables `x` and `y`, which were created on the stack during the execution of the subroutine body. We can see the "moving" stack pointer `RSP` which keeps pointing at the current top of the stack. We can also see the "static" base pointer `RBP` which remains fixed during the execution of the entire subroutine. Finally, we present a more meaningful implementation of `foo`, one that does some actual calculations and returns a value in `RAX`. The following example shows how the `foo` subroutine can access its actual parameters and its local variables relative to the base pointer `RBP`.

```
# ************************************************************
# * Subroutine   : foo                                       *
# * Arguments    : int a, int b, int c                       *
# * Return value : int                                       *
# * Description  : a simple demonstration subroutine         *
# ************************************************************
foo:
     pushq %rbp              # Prologue: push the base pointer.
     movq  %rsp, %rbp        # and copy stack pointer to RBP.

     movq %rdi, %rax         # load 'a' into RAX.
     addq %rsi, %rax         # add 'b' to 'a'.
     addq %rdx, %rax         # add 'c' to the total.
     pushq $1234             # create a local variable 'x'.
     pushq $0                # create another local variable 'y'.
     addq  $3, -8(%rbp)      # add 3 to 'x'.
     incq  -16(%rbp)         # increment 'y'.
     ...                     # etc, etc.

     movq  -16(%rbp), %rax   # Move a return value into RAX.
     movq  %rbp, %rsp        # Epilogue: clear local variables from stack.
     popq  %rbp              # Restore caller's base pointer.
     ret                     # return from subroutine.
```

3.3.5 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.

3.4 Input and output*

Doing IO in an assembler program can be quite tricky. First of all, normal processes do not have permission to access the hardware IO 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[4]. Instead, we will use the operating system's standard C library to do IO for us. 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

[4]If you're curious anyway, check out appendix 3.6.

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`.

3.4.1 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. In its simplest form it takes only one argument: an ASCII string. 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 [5]. 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.

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 %d years old\n", 25);
```

Assembly:

```
mystring: .asciz "I am %d 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 '%d' sequence is encountered. The '%d' sequence simply tells `printf` that it may expect an extra argument and that the argument must be interpreted as a decimal number for printing. We will also use other special sequences, such as '%X', during the lab course, but we will present them during the relevant exercises.

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

3.4.2 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`:

```
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, the x86 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:

```
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
```

3.5 x86 assembly language reference*

This subsection contains a short language reference for the x86 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", which is used by the GNU assembler [6] used during the course, and the "Intel syntax" of x86 assembly, which is used in the official Intel x86 platform manual.

3.5.1 About the AT&T syntax*

If you have examined some of the x86 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 assembly language. In theory, you could write your own x86 assembler and come up with a new syntax of your own. In practice however, there are only two flavors of x86 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 architecture. The Intel syntax is what you will see in the official x86 reference manual and platform definition. If anything, this could be considered the "official" x86 syntax. Long before the Intel Corporation introduced the x86 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 even existed, and thus the AT&T-style of assembly languages was widespread long before the x86 came

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

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.

We will now discuss the most important properties of the AT&T syntax, both in relation to the Intel syntax and on its own. You need to understand these properties before you can use the instruction set reference in the next paragraph or the official Intel x86 manual. First and foremost, 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 four different uses of the `push` instruction:

```
    pushb $3 # Push one byte onto the stack (0x03)
    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)
```

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.

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 acces 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 byte 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
```

In our instruction set reference we do not list these suffixes explicitly. The Intel manual does not list them either.

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

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

In AT&T syntax, a special expression exists to denote *offsets*. In many situations, you will want to use the value in a register as an offset relative to some fixed point. We denote this by putting braces around the register name and a base value before the braces:

```
foo: .byte 12
bar: .quad 34

    movq $foo, %rax    # Load the address of foo into RAX
    movq 1(%rax), %rbx # Load the _value_ of bar into RBX
```

In the example, we reserved space for the variables `foo` and `bar`. We then load the address of `foo` into `RAX`. In the second `mov` instruction we use the offset notation to calculate the address of `bar`, i.e. you should read `1(%rax)` as "one plus the contents of the `RAX` register". This syntax is explained in more detail in the GNU assembler documentation [7].

As a final note, it is important 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.

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

3.5.2 Instruction set reference

The next page contains a list of commonly used x86 instructions. It should be sufficient to get you through the lab course, but you may always study the official Intel manual [8] to obtain more instructions. The instructions are all case insensitive. In the table we denote the necessity of a `b`, `w`, `l` or `q` postfix with a period ('`.`'). See the previous paragraph for a description of AT&T syntax postfixes. As a further note, most bi-operand instructions require at least one of their operands to be a register. The other operand may be either a register or a memory location. Note that the multiplication- and division instructions require their operands to be in special registers[9] and that they store their results in more than one register (RAX and RDX).

[8] `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`
[9] Other forms of these instructions also exist, but they are not listed here.

Mnemonic	Operands	Action	Description
			Data transfer
`mov.`	SRC, DST	DST = SRC	Copy.
`push.`	SRC	`%RSP` -= 8, (`%RSP`) = SRC	Push a value onto the stack.
`pop.`	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	RDX = RAX%SRC; RAX = RAX/SRC	Unsigned division.
`idiv.`	SRC	RDX = RAX%SRC; RAX = 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`			`decl %rcx`, jump if not zero.
			Logic and shift
`cmp.`	A, B	`sub A B` (Only set flags)	Compare and set condition flags.
`xor.`	SRC, DST	DST = SRC ^ 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.
`shr.`	A, DST	DST = DST >> A	Shift right.
			Other
lea.	A, DST	DST = &A	Load effective address.
int	INT_NR		Software interrupt.

3.5.3 Assembler directive reference*

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 [10]

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 1.3), 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, `.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 3.5.3). 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 is a *little endian* machine, which means that a value like `.word 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
```

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

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

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

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).

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[11]. 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
```

3.6 Advanced topics

This section contains some background information on a number of advanced topics. The material is provided for the interested reader, i.e. is *not* required knowledge for the TI1406 lab course and it will mostly be interesting to those that have already finished the lab work. Most of the subjects require basic knowledge of the C programming language.

[11] Sharing subroutines is not part of this lab course, but if you are interested you can have a look at subsection 3.6.1

3.6.1 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 -no-pie
```

... 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.

3.6.2 Doing IO 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 IO 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. 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 use the `syscall` instruction.

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

A complete list of available Linux system calls can be found in the kernel source code, or at the following address: `http://man7.org/linux/man-pages/man2/syscalls.2.html`. 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 3.6.1). 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!

Appendix A

Rules and Regulations

This appendix states the rules and regulations to which you need to adhere in order to have your assignments approved. These rules effectively apply to *all* lab courses, but to avoid any confusion, they are stated here explicitly for this course.

A.1 Necessary conditions for approval

To have your assignments approved, your work needs to meet all of the following conditions:

1. You need to deliver *correct specifications*.
2. Your finished code needs to be *functionally correct*.
3. Your code needs to be *algorithmically correct*.
4. Your code needs to be *properly commented*.

We stress that your work will not be considered fit for approval until you meet all four of these criteria. The details on what we consider to be *correct* in this context will be defined in the subsequent paragraphs.

A.1.1 Correct specifications

If an assignment so asks, you will need to hand in a *correct specification* for the specified part of the exercise. Specifications must be written in *pseudocode*. Pseudocode is code that resembles actual high level (non-assembler) program code closely. It is called pseudocode because it does not necessarily have to be real, compilable code. A good, clear example of pseudocode is provided in Assignment 0 (2.1). We expect your pseudocode to be similar in clarity, quality and level of detail. Specifically, this means that your pseudocode must have:

- proper comments
- a good, clean layout
- a simple, clearly understandable structure

We expect that you have your specifications checked *before* you start on your implementation work. It is highly likely that the teaching assistants will ask you to make changes to your specifications, so we strongly advise you not to start programming before having your specs approved. If you choose to ignore this advice, we do not accept responsibility for any wasted effort on your part.

A.1.2 Functionally correct code

The source code that you hand in should be functionally correct. This is implied by the following:

- The source code must compile without errors or warnings.
- The program must function as specified in the assignment.
- The program must run to completion without runtime errors.

A.1.3 Algorithmically correct code

The programs that you hand in should be *algorithmically correct*. The correct algorithm is, by definition, the algorithm that was conveyed to you by the teaching assistants during the approval of your specifications. This implies that a program cannot be algorithmically correct if you did not have your specifications approved by a teaching assistant. We explicitly state here that you are *not* free to implement the desired functionality at your personal leisure and that solutions which are merely functionally correct are not sufficient for approval.

A.1.4 Compliance with subroutine conventions

As explained in Section 3.3, a calling convention allows subroutines to interact through a common set of rules. Furthermore, other conventions, such as the use of a prologue and epilogue in each subroutine, make code easier to understand and reuse by others. We expect that all uses of subroutines in your code, both calling external subroutines and writing your own, adhere to the conventions laid out in Section 3.3.

A.1.5 Properly commented source code

All source code, including pseudocode, should be properly commented. Commenting source code is not an exact science, but at the least we expect that you adhere to the following guidelines:

Natural language comments should be written in a natural human language, which is preferably in English or Dutch if you insist. We expect that you use the same language consistently. Pseudocode and overly complicated mathematical notations are not accepted as proper comments.

Continuity the comments in the body of your code should "tell the story" of the code in a concise, clear manner. If you strip away the accompanying source code, your comments should make up a compact description of the algorithm in acceptable prose. Do place your comments with care and avoid overly lengthy comments.

Subroutines Each subroutine should be accompanied by a clear description of its function. A description should also be given of the meaning and type of the arguments and the return value.

Layout Part of the readability of source code comes from good layout. We expect you to be precise and, most importantly, consistent in your style.

The "Sieve of Eratosthenes" example of assignment 1 contains good examples of properly commented assembler- and pseudocode. You should strive for the same level of quality. You are allowed to develop your own style of commenting and layout, but we demand that you to be consistent and precise. Sloppy comments or disturbing layouts will not be accepted.

A.2 Deadlines

The lab runs from the 3rd through the 7th week of the course. All groups of students must submit and present their four mandatory assignments and any bonus assignments they wish to complete **before** or during the lab session in the following weeks:

- week 5 for assignments 1 and 2,
- week 6 for assignments 3 and 4, and
- week 7 for bonus assignments.

For concrete dates, please refer to the Canvas page on the lab. Please start early and submit your assignments in time, such that we avoid the situation in which all students submit just in time before the deadline.

A.3 Anti-fraud policy

Our anti-fraud policy is very simple: zero-tolerance, within the limits set by the Vrije Universiteit. We will pursue each case of potential fraud, and will use the means provided by the Vrije Universiteit to punish (attempts to) fraud.

The following are some of the cases that are considered fraud:

- Sending your code to other groups. The motivation of "I sent it for them to find some inspiration" does not work.
- Copying somebody else's code. Changing the names of variables in someone else's code and submitting the results is still considered fraud.
- Receiving help from someone, when the help amounts to letting that someone write your code.
- Renting the services of a programmer, for example from `Rent-a-Coder.ro`, to solve the assignments for you.