CSE1400 Lab Assignments Manual

Delft University of Technology

Edition 2022-2023[1]

[1]Revision 20220903T1019

Acknowledgements

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

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

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

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

Contents

Acknowledgements **i**

1 Rules, Regulations and Etiquette **2**
1.1 Teaching Assistants: what they can and cannot do . . . 2
1.2 Verifying Your Work . . . 2
1.3 Rules & Regulations . . . 2
1.3.1 Correct Specifications . . . 3
1.3.2 Functionally Correct Code . . . 3
1.3.3 Algorithmically Correct Code . . . 3
1.3.4 Properly Commented Source Code . . . 4
1.3.5 Stack frames . . . 4
1.3.6 Official calling conventions . . . 4
1.4 Deadlines and Schedule . . . 4
1.5 Anti-Fraud Policy . . . 5

2 Getting Started **6**
2.1 Setting Up Your Environment . . . 6
2.1.1 Direct GCC Support . . . 6
2.1.2 WSL: Windows Subsystem for Linux . . . 7
2.1.3 Virtual Machine . . . 8
2.1.4 Docker . . . 9
2.2 Editors & Syntax Highlighting . . . 10
2.3 Building, Running and Debugging Programs . . . 11
2.3.1 Building & Running . . . 12
2.3.2 Debugging with GDB . . . 12

3 Mandatory Assignments **13**
3.1 ASSIGNMENT 1: Powers . . . 13
3.1.1 Assignment description . . . 13
3.1.2 Required Reading . . . 14
3.2 ASSIGNMENT 2: Recursion . . . 15
3.2.1 Assignment description . . . 15
3.2.2 Required Reading . . . 15
3.3 ASSIGNMENT 3: Memory . . . 15
3.3.1 Assignment description . . . 15
3.3.2 Required Reading . . . 17

4 Bonus Content **18**
4.1 ASSIGNMENT 4: A Colourful Discovery (max 500 points) . . . 18
4.1.1 Assignment Description . . . 18
4.1.2 Required Reading . . . 19
4.2 ASSIGNMENT 5 (500 points) . . . 19

4.2.1 Option A: Implement "diff" in Assembly 20
4.2.2 Option B: Implement a Simplified `printf` Function 20
4.2.3 Option C: Implement a Hashing Function 22
4.3 ASSIGNMENT 6: Hide Data in a Bitmap (750 points) 23
4.3.1 Assignment Description 23
4.3.2 Required Reading 24
4.4 ASSIGNMENT 7: Implement an Interpreter for Brainfuck (500–800 points) 25
4.4.1 Assignment description 25
4.4.2 Required Reading 26
4.5 ASSIGNMENT 8: Implement a Game (1,000 points) 26
4.5.1 Assignment description 26
4.5.2 Hints about Solving this Assignment 26
4.5.3 Last But Not Least 27

Chapter 1

Rules, Regulations and Etiquette

This chapter elaborates on aspects that you should consider during the labs. The role of Teaching Assistants (TAs) will be detailed in Section 1. Section 2 will elaborate on the submission process. Section 3 will briefly explain the necessary in order to get a solution approved. Lastly, sections 4 and 5 highlight the importance of deadlines and of not committing fraud.

1.1 Teaching Assistants: what they can and cannot do

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 proper authorities (see table below).

Login accounts:	Service point
Deadlines, rules, exceptions:	Course Coordinators (co-cs-ewi@tudelft.nl)
Enrolment:	Academic Counsellors (ac-bsc-ewi@tudelft.nl)

1.2 Verifying Your Work

In order for you to get the credit that you passed a lab exercise, you need to have your work approved by a teaching assistant. For this, you will have to submit your work to Submit[1] and then enqueue to the digital Queue[2]. Please enable notifications on this page. Once one of our assistants picks up your request, you will receive a notification. Once that happens, kindly wait for the TA to arrive at your spot. If your request has been picked up by an assistant but you are not found in the described spot, your request will be denied and you will have to enqueue again.

1.3 Rules & Regulations

The work you hand in during the lab is subject to certain rules and quality guidelines, which will be explained in this chapter. Your work will only be considered considered if it is in full complies with those. These rules effectively apply to *all* lab courses, but to avoid any confusion, they are stated here explicitly for this course.

[1] `https://submit.tudelft.nl`
[2] `https://queue.tudelft.nl`

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*.
5. Your code uses stack frames for all subroutines.
6. Your code uses the *official calling conventions*.

We stress that your work will not be considered fit for approval until you meet all of these criteria. In other words, if your assignment submission fails to meet any of the aforementioned points, you will **not** pass the assignment. The details on what we consider to be *correct* in this context will be defined in the subsequent paragraphs.

1.3.1 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 section 2.1 of the content manual. 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.

1.3.2 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.

1.3.3 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.

1.3.4 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 British English. 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.

You are allowed to develop your own style of commenting and layout, but we demand that you be consistent and precise. Sloppy comments or disturbing layouts will not be accepted.

1.3.5 Stack frames

Assembly is not a high-level programming language. Consequently, more thought has to be put into the behaviour of the stack. For instance, your code must always clean the stack after every function call.

For an explanation of everything that you must consider, kindly refer to section 2.5 of the content manual.

1.3.6 Official calling conventions

This relates to writing subroutines (which you will be required to for all assignments). You are expected to respect the official calling conventions at all times. That is to say, registers should be used in a specific order and for specific scenarios. When you can no longer use just registers alone for your program, you should start using the stack.

For additional information, consult section 2.6 of the content manual.

1.4 Deadlines and Schedule

In addition to the rules listed in this chapter, there may be hard deadlines. Please check the Brightspace pages for the most up-to-date information regarding those. Please note that teaching assistants are not allowed to consider any work after the expiration of a deadline. Any complaints should thus be addressed to the course coordinators.

The following table gives an overview of the **recommended** schedule for the lab. For a more detailed dealines overview, kindly check the Brightspace calendar: `https://brightspace.tudelft.nl/d2l/le/calendar/499403`.

Week 1:	Find a lab partner, read the lab manual and set up the environment
Week 2:	Assignment 1
Week 3:	Assignment 2 (Actual deadline A1)
Week 4:	Assignment 3 (Actual deadline A2)
Week 5:	Midterm Exam
Week 6:	Bonus assignments (Actual deadline A3)
Week 7-9:	Bonus assignments
Week 10:	Endterm Exam

Note: We recommend to sign off assignments as early as you are able to. There are limited sign-off slots available, and waiting until the deadline can mean that there are no longer enough slots for you to get (all) your assignment(s) signed off.

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. Do not let this happen to you and make sure you visit every session, so you can talk about your questions to the assistants. Outside lab hours, we encourage you to use `https://answers.ewi.tudelft.nl`.

1.5 Anti-Fraud Policy

Our anti-fraud policy is very simple: zero-tolerance, within the limits set by TU Delft. We will pursue each case of potential fraud, and will use the means provided by TU Delft 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.

Chapter 2

Getting Started

This chapter contains all the information you will need to get started on this lab including how to set up your environment, tips on which applications to use, and instructions on how to compile, run, and debug your code.

2.1 Setting Up Your Environment

There are multiple ways in which you can set up your environment, depending on your operating system. The option we recommend for your system is highlighted in bold.

	Windows			
	Windows 10	**Other**	**Mac OS**	**Linux**
Direct GCC Support	No	No	No	**Yes**
Windows Subsystem for Linux	**Yes**	No	No	No
Virtual Machine	Yes	**Yes**	**Yes (Intel CPUs only)**	Yes
Docker	(Yes)	(Yes)	**Yes (Mac with M1)**	(Yes)

2.1.1 Direct GCC Support

Supported systems: Linux

If your system is listed under direct GCC support, you can simply install the required software on your system as explained below.

Linux

Depending on your Linux distribution the installation will be different. Use the package manager of your Linux distribution to install the `gcc` and `gdb` packages. We support GCC v4 or above but we will use v8 when checking your work. For example, for Debian based distributions like Ubuntu, simply write the following command in your terminal:

```
sudo apt install build-essential gdb -y
```

If you are running another Linux distribution we will assume you know what you are doing and will be able to install `gcc` and `gdb` yourself (we recommend the package build-essential).

2.1.2 WSL: Windows Subsystem for Linux

Supported systems: Windows 10 and 11

Windows includes the option to install and run Linux inside of Windows. It is simple to install and set up. *Please note however, if you want a graphical interface for the "Create a Game" assembly bonus assignment, you have to use a virtual machine for it to display on Windows.*

Installation

1. Right click on the Windows button or press `Windows` + `X`.
2. Select "Windows PowerShell (Admin)".
3. Paste the following command into PowerShell and press enter:

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
```

4. Restart your computer.
5. Go to the Microsoft Store (open the start menu and type store).
6. Search for "Ubuntu" or go directly to `https://www.microsoft.com/store/apps/9n6svws3rx71`.
7. Click "Get" to install.
8. Wait for the installation to complete. If nothing happens for a while, try pressing enter.

You can now launch Ubuntu from your start menu. The first time you will be asked to set a username and password for your Linux account (be sure to remember these or write them down).

Finally, run the following command to install the necessary build tools:

```
sudo apt update && sudo apt upgrade -y && sudo apt install build-essential gdb -y
```

For more instructions and troubleshooting, see `https://docs.microsoft.com/en-us/windows/wsl/install-win10`.

Terminal

The default terminal that Windows uses for WSL is suboptimal and lacks support for features like blinking (which you need for bonus assignment 4). Instead, we recommend using the **Windows Terminal** (no, not the terminal that Windows comes with, but a program Microsoft made and named like this. We didn't invent the name either). You can install it from the Microsoft Store: `https://www.microsoft.com/store/productId/9N0DX20HK701`

Once installed and launched, you can change the settings to use Ubuntu by default, by clicking on the down arrow in the menu bar.

Finding files

When using WSL, it is recommended to put your files on a location in Windows, e.g. somewhere in your Documents folder. You can access this location in WSL by using the `cd` command to navigate to the correct folder, though you will have to translate your Windows path to a Linux path. For example, if your files are at `C:\Users\Student\Documents\CO\Lab`, you should use the following command to navigate to it in WSL (case sensitive!):

```
cd /mnt/c/Users/Student/Documents/CO/Lab
```

Should you want to access/edit the files created in a WSL-only location from Windows (the other way around from what we recommend), you can do the following. In your Ubuntu terminal, enter `explorer.exe .` (with the dot!) This will open the current folder in Windows Explorer. Please note that this location of `\\wsl$\Ubuntu-20.04\...` is only accessible while WSL is running.

2.1.3 Virtual Machine

Supported systems: All but M1 Macs

With a virtual machine you can run a virtual computer as a program on your computer. We provide you with a virtual machine (VM) with Debian and the necessary tools already installed. We also pre-installed useful editors and their assembly syntax highlighting support (see Section 2.2). You can run this on your own machine using VirtualBox.

Installing and Starting

1. Download the virtual machine from Brightspace
   (Content → Course Resources → Lab → Virtual Machine).
2. Download VirtualBox from `https://www.virtualbox.org/wiki/Downloads` and install it.
3. Open Oracle VirtualBox.
4. Click the "Import" button.
5. Select the virtual machine you downloaded in step 1 and click Next.
6. Leave all settings at the default settings and click Import.

The virtual machine is now ready to be used. Click the "Start" button (green arrow) to start it. To log in use the following credentials:

username:	student
password:	pwd

Troubleshooting

The virtual machine is very slow.
You can increase the number of CPU cores available to the virtual machine which might alleviate this problem. Select the virtual machine, click Settings → System → Processor and increase the number of processors. Ensure that the amount selected falls within the green bar displayed by VirtualBox.

Not enough memory.
If you don't have enough memory to start the virtual machine, first try closing all unnecessary programs on your computer. You can adjust the amount of memory made available to the virtual machine as follows: Select the virtual machine, click Settings → System. Now set an appropriate amount of RAM (minimum 1024 MB). Ensure that the amount selected falls within the green bar displayed by VirtualBox.

(Windows) Hangs on black screen during boot.
If you are using Windows and your virtual machine keeps getting stuck on a black screen while starting up, make sure that Hyper-V is enabled on your machine. To enable it (or check if it is enabled), perform the following steps:

- Press start.
- Search for "Turn Windows features on or off".
- Find "Hyper-V" in the list.

- Ensure the checkbox of Hyper-V is fully checked (a check mark, not a square).
- Press "OK".
- Let the process finish and restart your PC when prompted.
- Try the virtual machine again.

VT-X/AMD-V is not available.
If the VM refuses to boot and gives an error like "VT-X/AMD-V is not available", please make sure that the virtualization is enabled in your BIOS. For example, the HP Workstations from the TU Delft Laptop Project do not have this setting turned on by default. Here's a list of steps on how to turn virtualization on:

- Go to your BIOS settings. This can usually be done by pressing `F12` while booting (although it varies based on machine - we encourage you to google what works for yours). To enter your BIOS settings from Windows, the following approach **might** work, based on your machine:
  - Press start.
  - Click the power icon.
  - Hold shift while clicking "Restart".
  - Wait for a bit until the Windows UEFI settings appear.
  - Click "Troubleshoot".
  - Click "Advanced options".
  - Click "UEFI Firmware settings". This will reboot your PC into the BIOS settings.
- Inside the BIOS Settings, go to the "Advanced" menu, then to "System", and enable "Virtualization options".
- Make sure to "Save and Exit", and not just "Exit" the BIOS Settings.
- Once your PC has rebooted, you should be able to run the virtual machine in VirtualBox. If not, please ask the lab assistants for help.

For other laptops, the BIOS setting might have a different name or be under a different category. Try googling for "enable virtualization on <model of your laptop>" to find instructions specific to your laptop.

2.1.4 Docker

Supported systems: All, including Macs with M1

On an M1-chip MacBook (Apple Silicon) you cannot use our virtual machine. Instead, you can use Docker.

Although this approach works for all systems, we suggest you try the previously mentioned ones. That is because this approach leads to limited functionality when it comes to debugging. This can lead to a harder time completing the assignments, especially when it comes to the more difficult ones.

1. Download Docker, from:
   - `https://docs.docker.com/desktop/install/windows-install/` (for Windows)
   - `https://docs.docker.com/desktop/mac/apple-silicon/` (for Apple Silicon)
   - `https://docs.docker.com/desktop/install/mac-install/` (for previous Mac versions)
2. Install it
3. Open a terminal

4. Use `docker pull wukl/co`

Now you can use the following command in a terminal to create a container:

```
docker run --platform linux/amd64 -it wukl/co
```

Inside the container, you can run gcc and gdb as explained later in the manual. You can use vim or nano as editor inside the container. We recommend to use your preferred editor in Mac OS, and just copy pasting your code into a file in the container (e.g. in nano: `nano myfile.s`). Please keep in mind that shutting off your computer or stopping the container means the files in it are gone.

2.2 Editors & Syntax Highlighting

You are free to write your code in a text editor of your choice. There are various text editors which support syntax highlighting of x86 assembly which can be helpful for the assignments. We recommend a text editor such as Sublime Text (`https://www.sublimetext.com/`), Visual Studio Code (`https://code.visualstudio.com/`), gedit, vim or emacs.We will explain how to add/enable syntax highlighting for the editors that we recommend.

Sublime Text

You can add syntax support for x86 assembly as follows:

1. Go to Tools → "Install Package Control..." and wait for the installation to complete
2. Go to Tools → "Command Palette..." and enter "Package Control"
3. Select "Package Control: Install Package" (See Figure 2.1)
4. Enter x86, click on GAS-x86 (See Figure 2.2)

To enable the highlighting for a file, select View → Syntax → GAS/AT&T x86/x64. You can also set it as the default for all files of the same extension by going to View → Syntax → Open all with current extension as... and then selecting GAS/AT&T x86/x64.

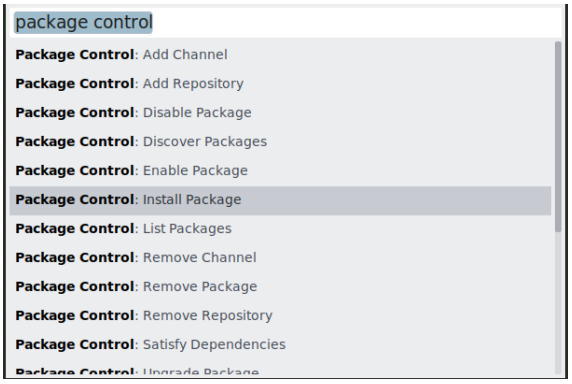


Figure 2.1: Installing a package in Sublime Text

Visual Studio Code

Install GNU Assembler Language Support as follows:

1. Launch VS Code Quick Open (`Ctrl + P`)

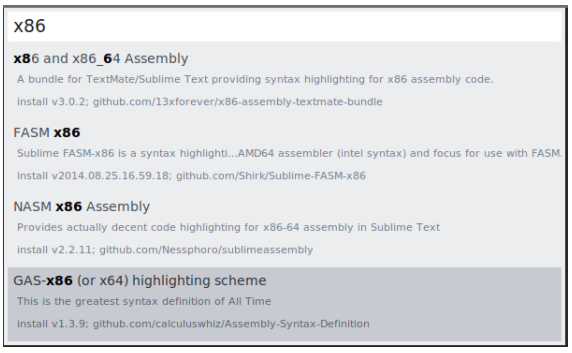


Figure 2.2: Installing x86 Syntax Highlighting for Sublime Text

2. Paste `ext install basdp.language-gas-x86`
3. Press enter

Save your files with the .s extension and VS Code will automatically enable the syntax highlighting.

gedit

To install x86 assembly syntax highlighting for gedit, open a terminal and paste the following command:

```
wget https://raw.githubusercontent.com/calculuswhiz/gedit-GAS-x86_64-highlighter/master/GAS_x86-64.lang
sudo cp GAS_x86-64.lang /usr/share/gtksourceview-3.0/language-specs/
```

Save your file with the .s extension and gedit will automatically enable the syntax highlighting. Otherwise, go to View, Highlight Mode..., and select "GAS (x86-64)".

VIM

Ensure that `git` is installed, then execute the following:

```
git clone https://github.com/calculuswhiz/vim-GAS-x86_64-highlighter.git
cd vim-GAS-x86_64-highlighter
sh install.sh
```

Save your file with the .s extension and vim will automatically enable the syntax highlighting.

Emacs

Emacs has an asm-mode which adds some support in writing assembly. There is a GAS mode for Emacs as well, which you can download at `https://github.com/Taeir/emacs-gas/blob/master/gas-mode.el`.

2.3 Building, Running and Debugging Programs

For your programs to be able to run, you need to compile them into an executable file. The following section contains instructions on how to do exactly this.

2.3.1 Building & Running

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

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

./nameofyourprogram
```

where `nameofyourprogram` is the name you want to give your executable file and `nameofyoursource.s` is the file in which you wrote your piece of code.

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 you 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 -no-pie -o nameofyourprogram nameofyoursource.s
```

The `-no-pie` flag has little to do with pie; PIE stands for Position Independent Executable and is an effort to make binaries less predictable and hacking harder. Unfortunately, it would make the assignments a bit harder as well, so you will have to provide -no-pie for now to disable it. More information can be found on the Internet [2].

2.3.2 Debugging with GDB

When your program gives unexpected output or a `Segmentation Fault`, one way to debug your program is to carefully read through your assembly code and try to find what's wrong. Another way is to use GDB.

In order to use GDB, pass the `-g` flag to the compiler as shown in the following command:

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

This will tell `gcc` to compile your program with debug flags enabled. In order to start the debugger, type in the following command:

```
gdb ./nameofyourprogram
```

You will enter a gdb shell, where you can use some basic commands to instruct the debugger. For example, `break myfile.s:10` will set a breakpoint in the file 'myfile.s' on line 10[3] and `run` will run your program. When execution is paused (either at a breakpoint or due to a segfault), you can use `info reg` to inspect the current register content or `next` to step to the next instruction. Many other commands can be found (for example) in this cheatsheet[4].

[1] This has nothing to do with objects in the Object-Oriented Programming sense.
[2] https://en.wikipedia.org/wiki/Position-independent_code#Position-independent_executables
[3] You can leave out the file name (e.g. `break 10`) when you just have one assembly file.
[4] Cheatsheet for GDB: `http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf`

Chapter 3

Mandatory Assignments

This chapter contains the description of all mandatory assignments. Before trying to get your solution approved by a Teaching Assistant, make sure to read 1.3.

3.1 ASSIGNMENT 1: Powers

This section will detail the first mandatory assignment that you need to do. For the assignment description, refer to 3.1.1. A list of the theoretical concepts that you require to do this assignment can be found under 3.1.2.

3.1.1 Assignment description

This first assignment consists of two parts. Part A will get you started with assembly and how to build and run your own assembly programs. In part B, you will write your first *subroutine* with I/O and some basic programming constructs. You will only need to hand in part B, but you can ask an assistant (not make a submission!) whether you are on the right track after finishing part A.

Part A: Getting Started

In this assignment you will be asked to write your first assembly program. You will have to use the knowledge you acquired from the example program (found under section 2.1 of the content manual) in order to complete this task, so make sure you have a thorough understanding of it. 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.

Exercises:

1. Create a new text file, called "power.s".
2. Implement a simple `main` routine that exits the program immediately with the proper exit code and without crashing.
3. Build your program and run it.
4. Alter your `main` routine in such a way that it prints a message containing your names, netIDs 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 B, 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.

Part B: Your First Assembly Algorithm

Now that you have successfully run your first assembly program, it is time to write a more complex program. In this assignment you will write a subroutine that takes several input parameters and returns a computed value.

Exercises:

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

```
/**
 * The pow subroutine calculates powers
 * of non-negative 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.
   It is not required to sign-off this specification, but you can ask a teaching assistant to check that it is correct. Also when you have questions about this assignment, the teaching assistant will ask for your specification.

2. Create a new subroutine called `pow`, which will be the implementation of your `pow` subroutine. Make sure the subroutine takes 2 inputs - first one should be the base and second one should be the exponent, and has 1 output - the base raised to the exponent.

3. Alter your `main` routine in such a way that it asks the user for a non-negative base and exponent. The first input should be the base and the second input should be the exponent.

4. Alter your `main` routine in such a way that it calls `pow` with the numbers it reads and prints the result of `pow` on the terminal.

3.1.2 Required Reading

1. You **must** respect the calling conventions. To read about them, check section 2.6.1 of the content manual.

2. To complete this assignment, you will need to use the **printf** subroutine. You may learn how to do so in section 2.7.1 of the content manual.

3. Section 2.3.1 of this manual explains the commands that you will need to enter on your shell in order to build and run your program.

4. To code the algorithm, you might find chapter 2.9 of the content manual useful. It describes programming constructs such as if statements and while loops.

3.2 ASSIGNMENT 2: Recursion

This section will detail the second mandatory assignment that you need to do. For the assignment description, refer to 3.2.1. A list of the theoretical concepts that you require to do this assignment can be found under 3.2.2.

3.2.1 Assignment description

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!"). 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.

Exercises:

1. Copy your "power" 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 `main` routine in such a way that it calls `factorial` with the number it reads, instead of calling power. `main` should print the result of `factorial` on the terminal.

2. Write a pseudocode specification of your `factorial` subroutine. The subroutine accepts a non-negative 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.
   It is not required to sign-off this specification, but you can ask a teaching assistant to check that it is correct. Also when you have questions about this assignment, the teaching assistant will ask for your specification.

3. Implement your `factorial` routine. Test your program thoroughly.

3.2.2 Required Reading

1. This assignment requires you to write a recursive subroutine. For this, check subsection 2.6.2 of the content manual.

3.3 ASSIGNMENT 3: Memory

This section will detail the third mandatory assignment that you need to do. For the assignment description, refer to 3.3.1. A list of the theoretical concepts that you require to do this assignment can be found under 3.3.2.

3.3.1 Assignment description

Hundreds of years ago, archeologists discovered a treasure chest full of ancient scripts and drawings. Sadly, the puny minds of the current civilisation could not comprehend the contents of these precious documents and after decades of searching for a way to unlock their mysteries, people were forced to admit defeat and locked them away until someone worthy came along and could decode them.

Then, a few days ago, a strange machine was discovered. Our knowledge of technology is yet too limited to be able to get it working, but the group of scientists working on it managed to glean how the machine decodes these artifacts.

Now, they have tasked you with recreating it in the primitive ways of our technology. They have found that the artifacts are encoded in 8-byte memory blocks. The bytes in a memory block signify the following (from highest to lowest):

Byte 1 - 2	Unknown - this is still being worked on but they have already determined that it is not crucial knowledge for decoding the messages.
Byte 3 - 6	The next memory block to visit.
Byte 7	The amount of times that character should be printed.
Byte 8	The ASCII[1] character which should be printed.

So executed on the following memory blocks (where B = 00000000):

0	B B	B B B 00001000	00000001	01001000	?	8	1	H
1	B B	B B B 00001010	00000001	01010111	?	10	1	W
2	B B	B B B 00000111	00000001	01101100	?	7	1	l
3	B B	B B B 00000001	00000001	00100000	?	1	1	␣
4	B B	B B B 00000101	00000010	01101100	?	5	2	l
5	B B	B B B 00000011	00000001	01101111	?	3	1	o
6	B B	B B B 00000000	00000001	00100001	?	0	1	!
7	B B	B B B 00000110	00000001	01100100	?	6	1	d
8	B B	B B B 00000100	00000001	01100101	?	4	1	e
9	B B	B B B 00000010	00000001	01110010	?	2	1	r
10	B B	B B B 00001001	00000001	01101111	?	9	1	o

your program should do the following:

1. Start at memory block `0`.
2. Print the character `H` once and jump to memory block 8.
3. Print the character `e` once and jump to memory block 4.
4. Print the character `l` **twice**, and jump to memory block 5.
5. ...
6. Print the character `!` and terminate.

Which would result in the famous `Hello World!`

Exercises:

1. Download the files for this assignment from Brightspace.
   The file you will be writing your code in is "decoder.s". It currently includes the message from "helloWorld.s" (the `.include ''helloWorld.s''` line does this). If you want to change the input message, simply change this line to include the file you want.
2. Write a pseudocode specification of your `decode` subroutine. The subroutine accepts the address of the message as its first parameter and has no return value. The following is an outline for you specification:

```
/**
 * The decode subroutine decodes the messages.
 *
 * Arguments:
 *
 *     address - the address of the message
 *               in memory
 *
 * Return value: none
 */
void decode(int adress) {
        // ...
}
```

[1] `http://www.asciitable.com/`

You may also write helper subroutines which will be called from `decode`.
It is not required to sign-off this specification but you can ask a teaching assistant to check that it is correct. Also, when you have questions about this assignment, the teaching assistant will ask for your specification.

3. Implement your `decode` subroutine. Test your program thoroughly. Provided are test files "abc_sorted.s", "helloWorld.s", and "final.s". We recommend starting with "abc_sorted.s". This should print two lines: 0 through 9 on the first line and a through z on the second line. The memory is sorted, so this should work by each time just taking the next memory block. For Hello World you will need to extract the index of the next memory block. As for the contents of Final: the archeologists have not been able to figure this out yet. Can you?

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

3.3.2 Required Reading

1. For this assignment, you need to be familiar with bit shifting. For this, check chapter 2.10 of the lab content manual.

Chapter 4

Bonus Content

This chapter contains only bonus assignments. For each assignment, you can receive extra points towards your final grade (the amount is always listed with the assignment).

4.1 ASSIGNMENT 4: A Colourful Discovery (max 500 points)

This section will detail the first bonus assignment that you can do. for the assignment description, refer to 3.3.1. Note that the first part of the assignment yields 200 points, and the second, 300. A list of the theoretical concepts that you require to do this assignment can be found under 4.1.2.

4.1.1 Assignment Description

Good news!

The team of scientists has figured out the last two bytes of our mysterious messages from the previous assignment: colours. So simple! The first byte is supposedly the colour of the background and the second one is the colour the character should be (foreground). Both of them are in the format of the ANSI 8-bit escape codes.

So returning to our example from section 3.3.1, the complete table would look as follows:

0	00000000	00000010	B B B 00001000	00000001	01001000	0	2	8	1	H
1	00000010	00000000	B B B 00001010	00000001	01010111	2	0	10	1	W
2	00000010	00000000	B B B 00000111	00000001	01101100	2	0	7	1	l
3	00000000	00000000	B B B 00000001	00000001	00100000	0	0	1	1	␣
4	00000000	00000010	B B B 00000101	00000010	01101100	0	2	5	2	l
5	00000000	00000010	B B B 00000011	00000001	01101111	0	2	3	1	o
6	00000000	00000010	B B B 00000000	00000001	00100001	0	2	0	1	!
7	00000010	00000000	B B B 00000110	00000001	01100100	2	0	6	1	d
8	00000000	00000010	B B B 00000100	00000001	01100101	0	2	4	1	e
9	00000010	00000000	B B B 00000010	00000001	01110010	2	0	3	1	r
10	00000010	00000000	B B B 00001001	00000001	01101111	2	0	9	1	o

Here is the what the colours used here stand for:

0	BLACK
2	GREEN

And so, here is what the message is actually supposed to look like:

Hello World !

If foreground and background colour are the same you would end up with unreadable text. The scientists have not figured what to do when this happens, so for now you should ignore the colours if the foreground and background colours are the same.

Exercise 4.1: +200 points

1. Copy over your solution from "decode.s".
2. Write a pseudocode specification of your new `decode` subroutine which now also prints the message in its correct colour scheme.
   It is not required to sign-off this specification but you can ask a teaching assistant to check that it is correct. Also, when you have questions about this assignment, the teaching assistant will ask for your specification.
3. Implement your new `decode` subroutine. Test your program thoroughly.

Science is advancing rapidly here! The scientist figured out what to do with memory blocks where the foreground and background colour are the same. These are apparently used for special effects. The scientists have not figured them all out yet, but we already know of the following:

0	reset to normal
37	stop blinking
42	bold
66	faint
105	conceal
153	reveal
182	blink

Note for WSL users: Support for these special effects depends on the terminal you are using. To get them to work properly, we recommend using "Windows Terminal", which is available in the Microsoft Store: `https://www.microsoft.com/store/productId/9N0DX20HK701`. See also 2.1.2

Exercise 4.2: +300 points

1. Copy over your solution from "4.1".
2. Implement the features in your `decode` subroutine. Test your program thoroughly.
3. Implement the special effect codes. Keep in mind that a special effect code should not change the colours. Now let's see what that final message actually should look like!

4.1.2 Required Reading

1. For this assignment, you must be familiar with ANSI Escape Codes. Kindly check section 3.1 of the lab content manual.
2. In addition, information about lookup tables and switch statements can be found in section 3.2.

4.2 ASSIGNMENT 5 (500 points)

This section contains information about the second bonus assignment. Note that there are three different options, out of which you are allowed to choose **only one**. For the assignment about the Unix "diff" program, check 4.2.1. For the one about a simplified "printf" function, kindly go to 4.2.2. If you would like to implement a hashing function, refer to 4.2.3.

4.2.1 Option A: Implement "diff" in Assembly

This subsection will detail the first option of this bonus assignment that you can do. For the assignment description, refer to 4.2.1. A list of the theoretical concepts that you require to do this assignment can be found under 4.2.1.

Assignment Description

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 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.

Note : The -i and -B options should be read from the command line arguments; not hardcoded. By convention you get your command line arguments the following way: first an integer that tells you how many arguments were provided (always at least 1; the name of your own program) and then the actual parameters. More information can be found online.

Required Reading

1. To further understand 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`.
2. It might be useful to check section 3.2 of the content manual, which tackles the subject of switch statements and lookup tables.

4.2.2 Option B: Implement a Simplified `printf` Function

This subsection will detail the first option of this bonus assignment that you can do. For the assignment description, refer to 4.2.2. A list of the theoretical concepts that you require to do this assignment can be found under 4.2.2.

Assignment Description

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.

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. Note that any number means that you need to support more than 6 arguments.

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 using system calls.
2. Write a new subroutine to recognize format specifiers in the format string. Initially, you can discard the format specifiers rather than process them. The rest of the string can be printed using the subroutine from the previous hint.
3. Implement the various format specifiers. It may help to implement %u before %d. Again, you may use the print function from the first hint if you implemented it.
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.

Required Reading

1. In order to successfully complete this assignment, the information in Chapter 3.3 of the content manual might be useful. This chapter elaborates on how I/O can be done without the C library.

4.2.3 Option C: Implement a Hashing Function

Assignment Description

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.

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[1] (especially the pseudo code). As you will see, the algorithm has to split up the data 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. After you finish this exercise, you can implement the rest of the code too, but that is not required.

Our part of the code can be downloaded from Brightspace. You can combine this with your own code by adding "`./sha1_test64.so`"[2] 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
```

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`.

Required Reading

1. For this assignment, it might be useful to refresh your bit-shifting knowledge. This can be found under section 2.10 of the content manual.

[1] `http://en.wikipedia.org/wiki/Sha1`

[2] there is also a sha1_test32.so for 32 bit compilation available

4.3 ASSIGNMENT 6: Hide Data in a Bitmap (750 points)

This section will detail the third bonus assignment that you can do. For the assignment description, refer to 4.3.1. A list of the theoretical concepts that you require to do this assignment can be found under 4.3.2.

4.3.1 Assignment Description

The scientists from Assignment 3 (3.3.1) suggested that our generation should also leave a message for future generations. Frustrated at how long it took them to implement a decoder, the decision was made that this new effort should also be an "interesting" puzzle.

You are tasked by the scientists to encrypt messages using assembly, into everyday barcodes. This assembly code will then later be hacked into a popular barcode software package. In detail:

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

Data: The Message

Complexity: Easy
To demonstrate that your assembly code works, encrypt and decrypt the message `The answer for exam question 42 is not F.`, 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 following pattern:

$$8 \times C, 4 \times S, 2 \times E, 4 \times 1, 4 \times 4, 8 \times 0$$

Here, `<number>` $\times$ `<character>` means that *character* is repeated *number* times, e.g., $8 \times C$ means `CCCCCCCC`. The added parts are called the "lead" and the "trail" of the message.

Data Compression: Run-Length Encoding (RLE)

Complexity: Moderate
After saving the message as a data chunk, the next step is to compress it. You will do so by using RLE-8. You have to devise your own algorithms for RLE-8-encoding and RLE-8-decoding the message described in the previous paragraph.

Data: Barcodes

Complexity: Easy
The message will be encoded in the first part of a barcode. This subsection details how the barcode can be created. For simplicity, imagine that the barcode looks like this:

```
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
```

This pattern is 31 pixels long. In this pattern, `W` stands for a white pixel and `B` stands for black. This pattern should always be followed by a red pixel.

Create a sequence by repeating this pattern, followed by a red pixel, 32 times. This will form a 32×32 image, where each pixel is either white, black, or red. Note that this key represents an RGB image, where each pixel is represented using three bytes.

Data Encryption: XOR

Complexity: Moderate

Now that the barcode is created, the RLE-encoded message can be encrypted. Your task is to encrypt the RLE-encoded message using the barcode image as key. Note that the key consists of $32 \times 32 \times 3$ bytes, while the message is much shorter. This means that only a small part of the image will change.

Data Representation: the BMP Format (Simplified)

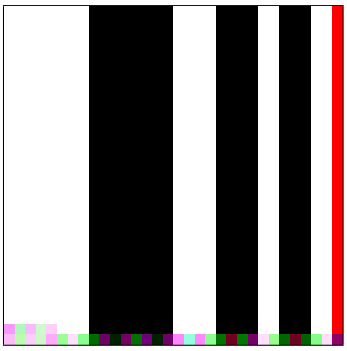
Complexity: Difficult

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. Therefore, your task is to save the encrypted message from step 4 in this data format.

Notes:

1. You are not allowed to use any other format. You can check the content manual for additional explanations on BMP.

2. The message will be quite more visible in this example than what one would do in reality. As an example, the following image contains the message `The quick brown fox jumps over the lazy dog` encrypted in the barcode pattern: the message will be quite more visible in this example than what one would do in reality. As an example, the following image contains the message `The quick brown fox jumps over the lazy dog` encrypted in the barcode pattern:



Last But Not Least

Make sure that you are also able to decrypt the message. After this, 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!

4.3.2 Required Reading

1. You can learn more about RLE in section 3.5 of the content manual.

2. For additional information about XOR, kindly check section 3.6 of the content manual.

3. In addition, section 3.7 of the content manual describes the bitmap format that you are expected to use.

4.4 ASSIGNMENT 7: Implement an Interpreter for Brainfuck (500–800 points)

This section details the fourth bonus assignment. For the assignment description, check 4.4.1. For a link of possible programs to test the speed of your interpreter on, check 4.4.2.

4.4.1 Assignment description

For this assignment, you will implement an interpreter for the very simple "programming language" called Brainfuck. [3] In Brainfuck, 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:

You can find a tarball containing assembly code and instructions for reading a file specified as a command line argument on Brightspace. You should write a `brainfuck` subroutine that takes a pointer to the Brainfuck code as argument and executes this program.

Examples:

- `hello.b`:

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

  Now, executing your program as follows

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

  should make it give

  ```
  Hello World!
  ```

  as output.

- `cat.b`:

  ```
  ,[.,]
  ```

  This program is the equivalent of the Linux command `cat`; running this program will copy whatever you enter in the console. By sending the null character to the console, the loop halts (this is done by pressing `Ctrl+2` in the terminal).

Bonus points:

The larger the Brainfuck program, the slower your interpreter might get. At some point, it might feel even tedious to wait for the result! Can you make your interpreter faster?

Find ways to significantly improve the speed or memory usage of your interpreter. At the end of this course, we will run all submitted Brainfuck interpreters and see which ones are the fastest. The creators of the top 5 interpreters will receive 800 points, places 6–10 will receive 700 points, and places 11–20 will receive 600 points. Everybody who at least submits a valid interpreter will receive 500 points.

[3] `https://esolangs.org/wiki/Brainfuck`

4.4.2 Required Reading

1. We encourage you to try more complex programs to test your interpreter. For instance, we recommend `http://esoteric.sange.fi/brainfuck/utils/mandelbrot/mandelbrot.b`.

4.5 ASSIGNMENT 8: Implement a Game (1,000 points)

This section details the last bonus assignment of the course. For the description, refer to 4.5.1. For some hints, kindly check 4.5.2

4.5.1 Assignment description

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, you must hand in a specification (about one A4, PDF) of your game via Submit (Bonus 8 Specification) and send an email to co-cs-ewi@tudelft.nl (subject "A8 Bonus Specification"). We will check whether your idea is reasonable and sufficient. We will reply to your email with details and approve your specification or request changes.

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

Requirements

1. Your implementation should correctly implement 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.

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).

4.5.2 Hints about Solving this Assignment

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 OS/library that we made and that will set up most of the basic things for you. This library switches the processor to 64 bit mode, makes sure you have a stack, and provides an API for you. After that, you are basically on your own. Again, many advantages, but these can turn easily into disadvantages.

This basic OS can be found on `https://github.com/thegeman/gamelib-x64`.

4.5.3 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!