[Section 1 Write and Run a Short Program: *first* 1](#__RefHeading___Toc570_1088046722)

[Section 2 Concepts explanation: Terminal, Assembler, Linker and Emulator 3](#__RefHeading___Toc572_1088046722)

[Section 3 Programs explanation: apt, as, ld and qemu 3](#__RefHeading___Toc574_1088046722)

[Section 4 Concepts explanation: Von neumann architecture, Bit, Byte and Legacy BIOS 5](#__RefHeading___Toc576_1088046722)

[Section 5 Explanation of program first.s 9](#__RefHeading___Toc578_1088046722)

[Coming sections 11](#__RefHeading___Toc580_1088046722)

[Appendix A Installing Ubuntu Desktop 12](#__RefHeading___Toc582_1088046722)

[References 12](#__RefHeading___Toc584_1088046722)

A Linux operating system is to be used as the platform for the experiments in this book. Please move to the next section if you currently have any distribution of Linux installed on your personal computer, it can be running on a virtual machine or alongside with Windows or macOS.

For these who do not currently have a Linux system in hand, or these who are even new to Linux system, Ubuntu desktop is recommended. Ubuntu is a free and open-source Linux distribution. It can be installed either on a virtual machine which is running on your current operating system or alongside with your current operating system. Please refer to appendix A for more information if you encounter some difficulties in installing Ubuntu desktop. It might take several hours if it’s your first time to install a system. Be patient and keep searching the answer whenever something confuses you.

## Section 1 Write and Run a Short Program: *first*

The best way to learn is by doing. Please note I am using Ubuntu 18.04.2 LTS, the commands or operations can be different from these given in this book if you are using any other distribution of Linux. In this case, I suppose you know how to probably change the commands or operations as you are an experienced Linux user. Log into Ubuntu desktop and do the following steps:

1. Write the first program. Find the **Documents** folder and click into it. Create a text file and name it with **first.s**. Open it, copy and paste the following lines into it then save and close.

.code16

.global \_start

\_start:

mov $26, %cx

mov $0x0903, %dx

mov $0x000c, %bx

mov $msg, %bp

mov $0x1301, %ax

int $0x10

loop: jmp loop

msg: .ascii "My first computer program!"

.org 510

.word 0xAA55

Congratulations! We have finished the hardest part of the whole chapter. Do not worry at the moment if you have no idea on what you have pasted, all these will be explained in the following sections. Please move to the next chapter if you understand fully what the code does.

1. Open a terminal. Right click on the blank area inside of **Documents** folder and then click **Open Terminal** in the context menu.
2. Install **binutils** tool. Key in **sudo apt install binutils** and press Enter button. Input your password when asked.
3. Assemble the program: **first.s**. Inside of the terminal, input the following command and press Enter button:

**as -o first.o first.s**

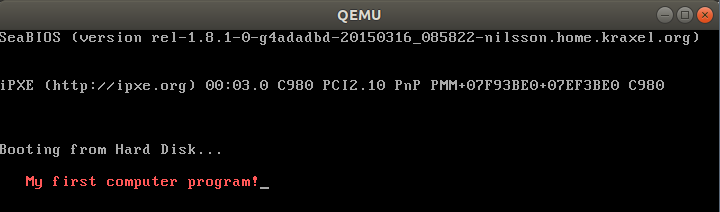
1. Generate the **first.img**. Input or copy the following command and press Enter key :

**ld -Ttext=0x7c00 -o first.img --oformat=binary first.o**

1. Install **Qemu** emulator. Key in **sudo apt install qemu**. Input **Y** and press Entre key when asked “Do you want to continue? [Y/n]”.
2. Run your program. Input or copy the following command and press Entre key:

**sudo qemu-system-x86\_64 -cpu max -drive format=raw,file=first.img**

1. We will see the following window if everything has gone well so far. The light red words My first computer program! on the screen is what the program does.



1. We have finished first program and had it run. Again please move to the next chapter if you understand what we have done by now. Otherwise see the next sections for the explanations.

## Section 2 Concepts explanation: Terminal, Assembler, Linker and Emulator

We go though some basic concepts before we explain what exactly we have done in the last section.

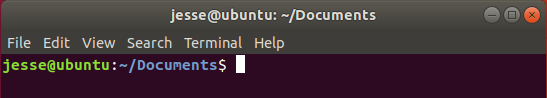
* Familiar yourself with **Ubuntu desktop environment**. New to Linux? Give yourself a quarter to click each icon or button you can find in Ubuntu desktop to see what they are. Refer to the guide below or just search any topic you are interested in if necessary.

Ubuntu desktop guide: <https://help.ubuntu.com/stable/ubuntu-help/>

* **Terminal, command-line** and **shell**. On Linux desktop, the Terminal is a program where command is used to interact with computer. Most people are familiar with GUI or graphical user interface. Instead of using mouse heavily to interact with all kinds of windows, we will use command-line interface or CLI to talk to computer. In case you are new to CLI, you may want to quickly go through chapter 2~4 of this book ***The Linux Command Line*** (<http://linuxcommand.org/tlcl.php>) written by Willian Shotts. Or search something like the most used 10 Linux commands and play with these commands for an hour.
* **Binutils**, **as** and **ld**. GNU Binutils are a collection of binary tools. The main ones are **as** and **ld**. The GNU Assembler, commonly known as **gas** or simply **as**. Assembler is a computer program which assembles assembly language to machine language. Assembly language is a more readable interpretation of a processor’s machine code, allowing easier understanding and programming by human [2]. Machine code is a computer program written in machine language [instructions](https://en.wikipedia.org/wiki/Instruction_set) that can be executed directly by a [computer](https://en.wikipedia.org/wiki/Computer)'s [central processing unit](https://en.wikipedia.org/wiki/Central_processing_unit) or CPU. Each instruction causes the CPU to perform a very specific task [3]. CPUs can only understand machine code. Machine code is some combination of statuses which only use two symbols: typically, “0” and “1”. The **ld** is another program, called linker, that takes one or more object files generated by the assembler and combines them into a single executable file, library file, or another object file [4].
* **Qemu emulator**. QEMU is a generic and open source machine emulator and virtualizer.

## Section 3 Programs explanation: apt, as, ld and qemu

* In step 1 of section 1, we composed a text file using assembly language. It’s OK for now if we do not really understand what the code means. We will examine these assembly code in the next sections. All we need to understand now is we wrote some assembly code and saved it as **first.s**, we call this file **source file** as it contains the source code. The suffix **s** stands for source file.
* In step 2, we opened a terminal window. The window will look like the below picture where **jesse** and **ubuntu** are the user name and computer name. **~** indicate the current user’s home folder which is **/home/yourUserName**. **Documents** is a folder under your home folder. We are now under **Document** folder because we right clicked in the blank area of this folder in the GUI. Believe you already tried to navigate to different folders following some books or web pages which tell the basic Linux commands usage. The place where the cursor flashes is where we key in the commands to interact with computer, just after the **$** sign.



* Advanced package tool or apt is a program that handles the installation and removal of software Ubuntu and related Linux distributions. **sudo** is short for superuser do, which is a program allows us to run programs with the security privileges of another user, by default the superuser. We use **sudo apt install binutils** when installing the tool collection **binutils**. Program **as** and **ld** which are part of the **binutils** will be used to translate the source file **first.s** to a runnable or executable file.
* Let’s see the command in step 4 **as -o first.o first.s.** **as** is the program name of GNU assembler. Except for ‘--’ any command-line argument that begins with a hyphen (‘-’) is an option. The **-o** in **as -o first.o first.s** means we want to give the output file which is generated by the assembler the name **first.o**. Number of options can be zero or many. Anything that is not an option will be treated as a source file, like the **first.s** here. After the execution of this command, we get an object file **first.o** which is the input file the **ld**.
* In Step 5, the **option -Ttext=0x7c00** tells the linker to locate the text section in the output file at the absolute address **0x7c00**. 0x7c00 is a magic number, this is where the computer loads the data from the external storage like hard disk or USB flash drive into main memory. I understand this explanation might still confuse you. We will explain this in the following sections. Similarly, **-o** **first.img** tells the linker to generate the output file with the name **first.img**. Option **–oformat=binary** specifies the binary format for the output object file. Finally after the execution of this command we get the runnable file **first.img**.
* With this runnable file **first.img**, we would like to find a way to run it now. For now we can image that the Qemu emulator, which has been installed in step 6, is our brand new machine which has exactly the function as the PC you buy from the shopping centre! But it’s not a real machine, it is only an emulator running on your Linux. We imagine it equips with one of the latest Intel or AMD CPUs (Intel and AMD are the companies who produce CPUs, probably not only CPUs). The difference is the PC you buy from the shop comes with an operating system, usually Windows. While this machine does not come with any operating system. We will load the **first.img** file into the Qemu machine. We image the **first.img** file is equivalent to the files which are located on the hard disk of your real PC. I see this analogy might be not accurate enough from a computer scientist’ view, but I wish this is good enough for us to understand the relationship between the **first.img** file and the Qemu emulator for now.
* For the command in Step 7, **sudo qemu-system-x86\_64 -cpu max -drive format=raw,file=first.img, qemu-sysem-x86\_64** is our brand new computer but without the operating system installed. Option **-cpu max** is used to enables all features supported by the accelerator in the current host machine. Option **-drive format=raw,file=first.img** is used to tell Linux to write the **first.img** into the disk of our brand new computer Qemu and then press the Power button. Specifying **format=raw** avoids Qemu detecting the format and believe it’s a trusted fromat. **file=first.img** obviously tells Qemu which file to be loaded into the hard disk. You may have noticed the new term **x86\_64**, also known as **x64**, **x86-64**, **AMD64** and **Intel 64**, is the 64-bit version of the x86 instruction set architecture. We have mentioned the smallest unit the CPU can execute is called instruction. The set of all the instructions the CPU can understand is some kind of abstract model of a computer. Almost all the PC and all kinds of Mac available in the shop in nowadays (in 2019) are using **x86\_64** CPUs. Refer to the below link if you are interested in more information on **x86\_64** and its family: <https://en.wikipedia.org/wiki/X86>

By now we have explained everything regarding the program and the commands we used so far but the most important part: what does the code in the text file mean. That’s the task for the following secitons.

## Section 4 [Concepts explanation:](#_Toc5350072) Von neumann architecture, Bit, Byte and Legacy BIOS

Before we explain the code in the source file first.s line by line, a couple of concepts need to be introduced.

1. Von neumann architecture introduction

In 1945, a mathematician and physicist John von Neumann and some other people wrote a report which describles adesign architecture for an electronic digital computer with these components:

* 1. A processing unit that contains an arithmetic unit and processor registers
  2. A control unit that contains an instruction register and program counter
  3. Memory that stores data and instructions
  4. External mass storage
  5. Input and output mechanisms[5]

Item iv correspondences to the disk or USB flash memory. The traditional hard disk which is an electromechanical device that uses magnetic storage to store and retrieve digital information using one or more rigid rapidly rotating disks. This kind of disk has been serviced the computer industry for more than half century. Since 1990s, a new kind of storage device that uses integrated circuit assemblies to store data appears. It is called solid-state drive or SSD or sometimes solid-state disk although they do not have physical disks. SSD is much faster and expensive than the HDD for hard disk drive. Luckily we do not need to understand too much of their working principles. All we need to know both of them can store data persistently even with power off. No matter what kind of information it is, the existing form on the disk is always a series of two kind of status. We use ‘0’ and ‘1’ to indicate these two kind of status. So the information in a disk or USB drive or DVD disc is just a series of ‘0’s and ‘1’s. Can the ‘0’s and ‘1’s represent all kinds of information like music, movie, cartoons, texts and all kind of pictures?

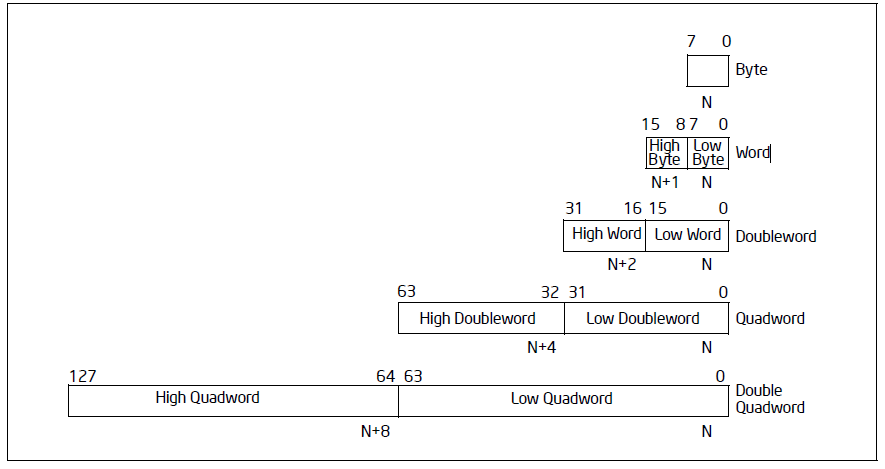
1. Bit and Byte

Now it’s a great time to introduce the concept of bit. We already know that the unit information stored on a disk is either ‘1’ or ‘0’, logically we can imagine the data inside of a disk a combination of ‘0’s or ‘1’s. We call each of these smallest information unit a bit. The x86-64 architecture names a set of different data stroage sizes as follows:

|  |  |  |
| --- | --- | --- |
| Storage | Size (bits) | Size (bytes) |
| Byte | 8-bit | 1 bytes |
| Word | 16-bit | 2 bytes |
| Double-word | 32-bit | 4 bytes |
| Quadword | 64-bit | 8 bytes |
| Double quadword | 128-bit | 16 bytes |

Table: Data type table

The below figure also gives the definition of Low Byte, High Byte, Low Word, High Word etc..

Figure: Fundamental data types [7]

1. Memory is just a pile of boxes

There are two main kinds of memory, volatile and non-volatile in a computer system. Non-volatile memory like ROM is used for storing firmware such as BIOS. We will talk about BIOS shortly. Firmware is a specific class of computer software that provides the low-level control for the device’s specific hardware. All we need to know for now is the BIOS system which is call firmware is stored in ROM, these data in ROM will not lost even when power off. The volatile memory are typically primary storage or main memory is random-access memory or RAM. We can imagine the RAM just like a piles of boxes which can store data. Inside of each box it’s either ‘0’ or ‘1’, each box represent a bit. When the computer system is running, CPU load data from these boxes into CPU to do the computing. As there are so many boxes the CPU has find a way to load the right boxes, naturally the clever early computer scientist worked out a way: give these boxes an address. The minimum addressable unit of these boxes are 8, which is a byte. For now we can image that the main memory has a tall piles of boxes, each level contains 8 boxes (a byte, also 8 bits). The lowest level byte comes with an address 0, the second lowest level byte with address 1, the third lowest level byte with address 2, and so on. And again these ‘0’s and ‘1’s in main memory disappear once power off.

1. CPU is a black box with registers

The text book tells us there are two main components inside of a CPU, the control unit or processing unit. But for now we can think a CPU is just a black box, which means we do not need to understand how it works in the hardware level. All we need to know is it can execute instructions. We can image that a CPU is just like a mini robot. This little robot or just a black box each time read in an instruction, it does something based on the instruction then next instruction will be fed to it. We already talked the minimum unit of command it can understand is an instruction. Each instruction itself is actually a number of ‘0’s and ‘1’s. Where are these instructions stores? Outside of this little black box, there are many storage boxed, which are called registers. These registers reflect the CPU status, store the instructions and all kinds of data. We will use the learn more about the names and function of these registers.

1. I/O devices

We briefly talked about the CPU, main memory and disks, to conclude the first overview of computer architecture, we have a look a the following illustration.

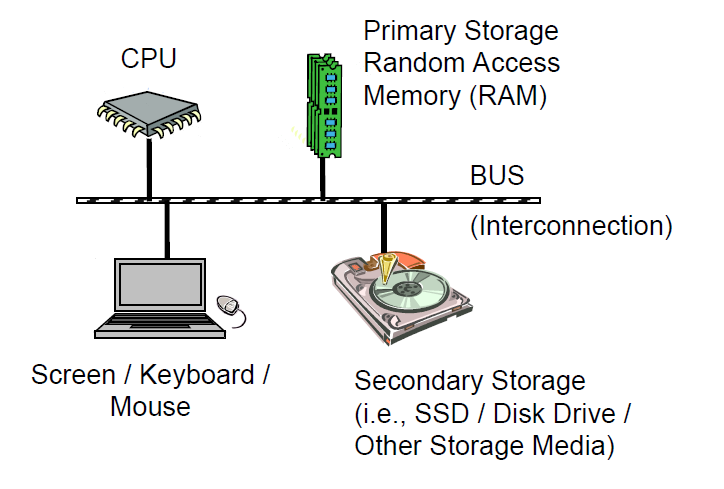


Illustration: Computer Architecture[6]

The above illustration summaries well what have discussed so far. All the other devices we have not mentioned like monitor/screen, keyboard and mouse etc. plus the SSD or HDD and other secondary storage media are all call I/O devices. The reason why we put all these except the CPU and main memory/RAM into one category (the I/O devices) is the CPU treats all of these devices in a very similar way. The BUS is just like the high ways, that’s where the CPU, main memory and I/O devices communicate through.

1. Binary and Hexadecimal numbers

We learned that every kind of information stored in any kind of disks is just a number of ‘0’s and ‘1’s. Now we consider how to indicate the integer numbers like 0, 1, 2, 3 etc. using these magic ‘0’s and ‘1’s. Say if we are give two bits, we have four kinds of different combination of ‘0’s and ‘1’s: 00, 01, 10, 11. It’s very easy to think that we can just use binary value to represent integers. What is the biggest integer a byte can represent, 1111 1111, right? Some one might think it’s decimal number, so we use 0b prefix the number to indicate this is a binary number instead of a decimal number. So 0b11111111 is the biggest integer that 8 bits can represent. A bit hard to count the number of ‘1’s? Then hexadecimal numbers are used to make it much clearer, 0-9 and a-f (or A-F) are used to represent decimal number 0-15. So 0b1111 equals to 15 in decimal and 0xF in hexadecimal. Prefix 0x is used to indicate a hexadecimal. The biggest number a byte can represent which is 0xFF equals to 15x16+15 which is 255.

1. Legacy BIOS

A **ROM BIOS** (**Basic Input/Output System**) is a set of programs permanently stored in a **ROM** (Read-Only Memory) chip located on the computer motherboard. These programs micro-manage the hardware devices installed on the computer. When we turn on the computer, the ROM BIOS initializes and tests these devices. The first job of a ROM BIOS is to initialize and configure the computer hardware when we turn on the computer (system boot). The BIOS runs a series of complex programs called the **Power On Self Test** (POST), which performs a number of tasks, including:[8]

• Test Random Access Memory (RAM)

• Conduct an inventory of the hardware devices installed in the computer

• Configure hard and floppy disks, keyboard, monitor, and serial and parallel ports

• Configure other devices installed in the computer such as CD-ROM drives and sound cards

• Initialize computer hardware required for computer features such as Plug and Play and Power Management

• Run Setup if requested

• Load and run the Operating System

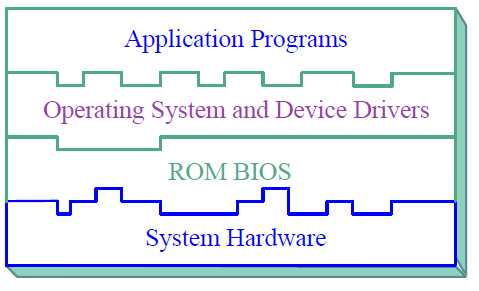
The last task is to load the Operating System code usually from the disk. The executable file **first.img** is loaded into main memory and executed when the last command “**sudo qemu-system-x86\_64 -cpu max -drive format=raw,file=first.img**” is executed.

Illustration: Computer System Layers[8]

The above illustration explains well the layers for a computer system. ROM BIOS which is also called firmware sits in between the bare hardware and the Operating System. The application programs like a web browser, a word processing program, or a video player mounts on the top layer.

## Section 5 Explanation of program first.s

In this section we will explain the program wrote in section 1.

Line 1 **.code16** is a directive which instructs the assembler, GNU as or GAS the one we use here, to generate runnable file in 16-bit mode. As by default GAS generates 32-bit or 64-bit code depending on the configuration. The reason why we need 16-bit runnable file is at the time when this program runs the CPU is in 16-bit real mode. All directives have name that begin with a period (‘.’) in GAS. These directives are not translated into any machine code but only tells GAS how to organise or translate the code into machine code.

The second line code **.global** is a directive as well. The function of this directive is to make the symbol, which\_start in this program, visible to linker. Program first.s only has one source file, while for programs which have more than one source file, the .global directive will make symbol is also available to other source files. Back to the third line of the program, \_start is the default entry point. GAS makes the runnable file starts from the first instruction which is just after the symbol \_start.

In line 4 mov $26, %cx, where **mov** is its opcode, $26 and %cx are its two operands. An opcode or operation code can have zero to two operands. 26 with a prefix $ indicates an immediate number 26 in decimal. A register name **cx** with prefix % indicates the content inside of the register. This instruction tells CPU to move the immediate decimal number 26 into the register **cx**, which is a 16-bit register.

Similarly line 5 mov $0x0903, %dx moves the data which is a hexadecimal number 0903 to register dx. Line 6 mov $0x000c moves the data which is a hexadecimal number 000c to register bx.

We look at line 11 msg: .ascii “my first computer program”. .ascii which starts with ‘.’ is a directive, it tells the assembler to reserve space for a string or text, which is the following “my first computer program” here. The msg before the semicolon (‘:’) is called a label. With this label we can refer the address where the string is in this case.

Line 7 mov $msg, %bp moves the address for the string of “my first computer program” to register bp. Similarly line 8 mov $0x1301, %ax moves number 0x1301 to register ax.

The int opcode here is an interruput, which force the CPU stop executing current task and start calling a routine (we understand a routine or a handler or a function is a just another piece of code stored somewhere else). For this case, the operands $0x10 which is decimal number 16 is given to CPU then CPU searches a table called interrupt vector table or IVT to find the location of the handler and than execute the code inside of the handler. Once the interrupt handler finishes, the CPU comes back to execute the next instruction. The concept of Interrupt is not not hard to understand, is it? Actually this kind of interrupt is called software interrupt. There are two other kinds of interrupts one is external interrupt or hardware interrupt, the other one is called internal interrupt or exception interrupt. We will talk these two later.

Line 10 loop: jmp loop lets the CPU jumps to the label loop which again let the CPU to jump to the same instruction. So it’s a dead cycle which will freeze the monitor, the CPU would keep doing this instruction until we close the emulator.

The last two lines are all directives. .org 510 tells the assembler to put the next instruction or data from the 510th byte of the whole executable file. The last line .word 0xAA55 will fill a word which is two bytes at the 510th and 511st byte location of the file. Actually these two numbers are magic numbers. When the BIOS program starts searching any bootable disk, it examine the first 512 bytes of the disk if it finished with 0xAA55, it thinks it’s a bootable disk and then BIOS program copies this 512 bytes into the main memory (put them byte by byte from the address 0x7c00) then CPU starts executing the program for the main memory address 0x7c00. In our example, that’s the runnable code generated from our first assembly source file first.s.

Now the only problem is why assign register cx, dx, bx, bp and ax with numbers 26, 0x0903, 0x000c, $msg and 0x1301. In order to understand this, we need to find out the interfaces the legacy BIOS defines. In other words what numbers or parameters we need to assign to the registers before the interrupt instruction.

The following web pages give an easy way to check the meanings of these numbers.

Legacy BIOS Interrupt Vector Table: <http://www.ctyme.com/intr/int.htm>

Legacy BIOS colours attributes: <https://en.wikipedia.org/wiki/BIOS_color_attributes>

Spend some time to study the above web pages, then try change the 0x000c to 0x0002 (changed to green colour). Re-assemble, re-link and launch the Qemu to run the program. Actually instead of keying in 3 command lines we can connect the three commands with semicolon(‘;’):

as -o first.o first.s;ld -Ttext=0x7c00 -o first.img --oformat=binary first.o;sudo qemu-system-x86\_64 -cpu max -drive format=raw,file=first.img

## Coming sections

Print Ascii code on screen, print a diagram on screen, Point: ASCII Code, Legacy BIOS, UEFI BIOS

## Appendix A Installing Ubuntu Desktop

You may want to install the Ubuntu on a virtual machine which is running on your current operating system, Windows or macOS. Go and search in your browser on how to install a virtual machine on your current operating system. For these who want to know what is a virtual machine or which virtual machine can be used, you may want to quickly review this page <https://en.wikipedia.org/wiki/Virtual_machine>.

For these who prefer to install the Ubuntu alongside with your current operating system or who have already installed a virtual machine, please move to Ubuntu official website to download the Ubuntu desktop. During the process of downloading, you may need search on how to install Ubuntu desktop on your virtual machine or alongside with your current operating system.

## References

[1] William E. Shotts, Jr., *The Linux Command Line, Fifth Internet Edition*, 2019, <http://linuxcommand.org/tlcl.php>, (accessed 3 April 2019).

[2] <https://en.wikipedia.org/wiki/Assembler>

[3] [https://en.wikipedia.org/wiki/Machine\_code#Instruction\_set](https://en.wikipedia.org/wiki/Machine_code" \l "Instruction_set)

[4] <https://en.wikipedia.org/wiki/Linker_(computing>[)](https://en.wikipedia.org/wiki/Linker_(computing))

[5] Von neumann architecture: <https://en.wikipedia.org/wiki/Von_Neumann_architecture>

[6] Ed Jorgensen, x86-64 Assembly Language Programming with Ubuntu, version 1.1.28, 2019, <http://www.egr.unlv.edu/~ed/assembly64.pdf>

[7] Intel® 64 and IA-32 ArchitecturesSoftware Developer’s Manual

[8] PhoenixBIOS 4.0 User Manual Revision 6

[9]Legacy BIOS Interrupt Vector Table: <http://www.ctyme.com/intr/int.htm>

[10]Legacy BIOS colours attributes: <https://en.wikipedia.org/wiki/BIOS_color_attributes>

[11]<https://en.wikipedia.org/wiki/Virtual_machine>

[12]<https://help.ubuntu.com/stable/ubuntu-help/>

[13]<https://en.wikipedia.org/wiki/X86>

as manual https://sourceware.org/binutils/docs-2.32/as/index.html#SEC\_Contents

ld manual https://sourceware.org/binutils/docs/ld/Options.html#Options

qemu manual

Introduction to x86 Assembly: <https://software.intel.com/sites/default/files/m/d/4/1/d/8/Introduction_to_x64_Assembly.pdf>

x86 assembly referenct: <https://www.felixcloutier.com/x86/>

online x86 & x64 assembler and disassembler: <https://defuse.ca/online-x86-assembler.htm>

x86 assembly wikibook: <https://en.wikibooks.org/wiki/X86_Assembly>

x86 assembly with Ubuntu: <http://www.egr.unlv.edu/~ed/assembly64.pdf>

using as, The GNU Assembler: <https://web.eecs.umich.edu/~prabal/teaching/resources/eecs373/Assembler.pdf>

<http://www.phoenix.com/resources/specs-bbs101.pdf>

[https://firmware.intel.com/sites/default/files/resources/A\_Tour\_Beyond\_BIOS\_Memory\_Map\_in%20UEFI\_BIOS.pdf](https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Memory_Map_in UEFI_BIOS.pdf)

<https://www.cs.cmu.edu/~410/doc/minimal_boot.pdf>