

Computer Systems Architecture

Laboratory - Part 0x00

Bogdan Macovei
Ruxandra Bălucea
Cristian Rusu

October 2026

Contents

1	Introduction & Administrative Information	3
1.1	Introduction	3
1.2	Administrative Information	4
2	Virtual Machine & Git	5
2.1	Virtual Machine	5
2.2	Git	5
2.3	Working Environment	6
3	x86 Assembly Language	7
3.1	x86 Architecture Registers	9
3.1.1	General-Purpose Registers	9
3.1.2	EFLAGS	10
3.1.3	Addresses	10
3.1.4	Notation	10
3.2	Data Types and the Basic Structure of a Program	11
3.2.1	Data Types	11
3.2.2	Program Structure	11
3.3	Example of an x86 Program	12
3.3.1	The MOV Instruction	12
3.3.2	System Interrupts	12
3.3.3	A First Program: Source Code	13
3.4	ASM Programs and Debugging	15
3.4.1	Another Program	15
3.5	Hello, world! in Assembly	17
3.6	Arithmetic and Logical Operations	18
3.6.1	Arithmetic Operations	18
3.6.2	Example	18
3.6.3	Logical Operations	19
3.6.4	Logical Shift Operations	20
3.7	Conditional and Unconditional Jumps	21

3.7.1	Unconditional Jumps	21
3.7.2	Conditional Jumps	21
3.7.3	Simulating Repetitive Structures	23
3.7.4	Simulating Repetitive Structures: Solution 1	23
3.7.5	Simulating Repetitive Structures: Solution 2	24
3.8	One-Dimensional Arrays	26
3.8.1	Memory Addressing	26
3.8.2	Load Effective Address (LEA)	26
3.8.3	Memory Representation of One-Dimensional Arrays	26
3.8.4	Declaring One-Dimensional Arrays	27
3.8.5	Examples: Program 1	27
3.8.6	Examples: Program 2	27
3.9	About Inline Assembly	29
4	Proposed Exercises	30
4.1	Laboratory 1	30
4.2	Laboratory 2	30
4.3	Laboratory 3	30
5	Additional Exercises	31
6	Additional Resources	33

1 Introduction & Administrative Information

1.1 Introduction

The purpose of this laboratory is to introduce you to x86 Assembly language and familiarize you with the AT&T syntax for x86 processors. Accordingly, the main emphasis is on learning the fundamental concepts of programming in an assembly language.

The laboratory will put into practice many of the computer architecture principles described in the lectures and seminars. As programmers, there are many advantages to understanding Assembly language:

- you will understand exactly how a computer executes programs;
- you will be able to optimize the code you write at instruction level, when needed;
- you will be able to understand what compilers do with your source code written in a high-level programming language;
- you will understand how the operating system interacts with a program you have written (program execution and termination, working with Input/Output devices through the operating system);
- you will understand how the operating system interacts with your hardware (you will be able to write drivers);
- you will be able to integrate inline-assembly instructions into C/C++ code for efficiency and control (you will be able to program real-time operating systems and operating systems for microcontrollers);
- you will be able to understand how other programs work when their source code is not available to you (disassembling ordinary binary files, analyzing and diagnosing potentially risky programs – viruses, malware, etc.).

It is also important to understand the disadvantages of Assembly language:

- programs written in Assembly can run only on the architecture for which they were written (machine-specific code); for example, programs written for x86 will not run on ARM systems;
- writing a program in Assembly requires much more effort (compared with C/C++, let alone Java/C#/Python);
- debugging a large Assembly program (many lines of code) is extremely difficult.

You should keep these aspects in mind when choosing the programming language in which to work for a particular project.

In this laboratory, although the emphasis will be on writing .asm files from scratch, there will be situations where, for optimization and code readability, we will use assembly-language instructions only in certain portions of C/C++ code.

1.2 Administrative Information

We have 7 laboratory sessions allocated for covering the material, with the last one reserved for the final laboratory test. This document is the first part of the laboratory material and is intended to cover the activities from the first 3 laboratory sessions. We suggest that you work through the material at least at the following pace:

- Laboratory 1: installing a Linux operating system, using a version-control system (Git), using a text editor, and using the Linux command line;
- Laboratory 2: covering the basic x86 Assembly concepts described in this document and executing your first x86 Assembly programs (the examples in the text and 0x00 from the Proposed Exercises section);
- Laboratory 3: understanding the details described in this part of the laboratory material and working through the problems suggested at the end of this document (all, or as many as possible, from the Proposed Exercises section).

The final 3 laboratory sessions will continue working with x86 Assembly language (more advanced concepts) and will be covered in another document (Laboratory - Part 0x01).

The final laboratory grade represents 40% of the final grade for Computer Systems Architecture and is computed as the weighted average of the three criteria above. The grade is reported with two decimal places, obtained by truncation.

The passing criteria are as follows:

1. the grade obtained for the laboratory assignment must be at least 5;
2. the grade obtained for the final test must be at least 5;
3. the final laboratory grade, obtained by computing the weighted average, must be at least 5.

Attendance is not counted and therefore does not affect the final result in any way. Likewise, the problems in the Proposed Exercises section are not graded and do not influence the final grade, but working through them is recommended. The final laboratory test, problems from the lectures, and the final exam will follow the style of the problems described in the seminars and laboratory sessions.

2 Virtual Machine & Git

Before we begin programming in Assembly language, we must make sure that you have an appropriate operating system installed and the programming tools required for our purposes.

2.1 Virtual Machine

This laboratory will be taught under Linux, so if you do not use a Linux distribution as your primary operating system, you will need to install one at least in a virtual machine. We recommend downloading an Ubuntu image and installing it in VMWare¹ or VirtualBox². You can find the operating-system image on the Ubuntu website at <https://ubuntu.com/download/desktop>.

An excellent video tutorial for installing VirtualBox, together with several system-configuration suggestions, is available online³. If you have never used Linux before, another *slightly* more detailed tutorial is a worthwhile investment for your future⁴.

For those of you who do not have x86 computer systems (for example, Mac M1/M2/etc.), follow the tutorial available here⁵. You will need to install UTM⁶ and emulate an x86 Ubuntu system; you can find it in the UTM gallery after installing UTM (allocate at least 4–6 GB RAM). If UTM does not work, you can also try OrbStack⁷.

2.2 Git

In this course, all source code you write will be integrated into a version-control system. This is standard practice in the software industry. Your work during the laboratory sessions will be saved in Git. We therefore recommend creating a GitHub account, creating a repository named "CSA Laboratory" or something similar, and keeping all the problems you work on centralized there. When/if you have questions about programs written for this laboratory, we can check them directly online in Git and suggest solutions.

An excellent video tutorial for installing Git on your system is available online⁸.

After Git is installed, we are ready to write and save our first test program. In the terminal, we will enter

```
mkdir Laboratory1
cd Laboratory1
git init

git config user.name "First Last"
git config user.email "email"
```

We will create a basic C-language file,

¹VMWare, <https://www.vmware.com/products/workstation-player.html>

²VirtualBox, <https://www.virtualbox.org/>

³How to Use VirtualBox (Beginners Guide), https://www.youtube.com/watch?v=sB_5fqiyis4

⁴The Complete Linux Course: Beginner to Power User!, <https://www.youtube.com/watch?v=wBp0Rb-ZJak>

⁵<https://linuxhint.com/run-linux-m1-based-mac/>

⁶<https://mac.getutm.app/>

⁷<https://docs.orbstack.dev/machines/#intel-x86-emulation>

⁸Git & GitHub Tutorial for Beginners, https://www.youtube.com/playlist?list=PL4cUxeGkcC9goXbgTDQ0n_4TBz000ocPR

```
nano file.c
```

In the opened editor, we will write our first C program

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

Save it: CTRL+X, Y, Enter.

Next, we add the file to Git and commit it:

```
git add file.c
git commit -m 'first commit for this project'
```

From the browser, save the link to the Git repository; it should have the form `https://github.com/name/repositoryname.git`.

```
git remote add origin https://github.com/name/repositoryname.git
git push origin
```

At this point, the file is saved in Git. Check online that the source file is visible.

2.3 Working Environment

In this course, in future courses, and throughout your career as programmers, you will encounter the need to use a Linux system and a version-control system for software products (usually Git). Consequently, please allocate the necessary time to set up a working environment suitable for a software developer. You should become reasonably comfortable installing these systems because you will use them continuously from now on if you choose a career in IT.

As a last resort, if you cannot install Linux and Git, we recommend an advanced browser-based Assembly programming system⁹.

⁹<https://www.onlinegdb.com/> or <https://ideone.com/>

3 x86 Assembly Language

In this chapter, we will cover the most important fundamental concepts of x86 Assembly language.

x86 is a family of instruction sets introduced with the Intel 8086 (from which the "86" in x86 originates) and 8088 microprocessors. The instruction set¹⁰ was adopted by other microprocessor manufacturers (we particularly note AMD's contribution) and today is the standard for 64-bit microprocessors in personal computers. Over time, the instruction set has undergone many changes, always while maintaining compatibility with previous x86 computer architectures. We note one important difference here: smartphones and tablets generally implement a different computer architecture, *Advanced RISC Machine (ARM)*.

x86 Assembly is a low-level programming language that enables direct control over the central processing unit (CPU) of a computer with an x86 architecture. In many high-level programming languages, the compiler first generates assembly code as an intermediate step. Assembly source code is a piece of text, so a processor cannot directly interpret and execute that text. Assembly is then transformed into machine code: clearly encoded binary machine instructions (opcodes) that can subsequently be executed by the processor step by step. We would like to draw your attention to the fact that x86 Assembly has two distinct syntaxes: Intel (used on Windows systems) and AT&T (used predominantly on Unix systems). The differences between the two are not significant (they mainly involve slightly different syntax), but you can review them online¹¹. In this laboratory, we use only AT&T syntax. If you understand one of the syntaxes, the other is very easy to read and understand.

The main components of x86 Assembly language are:

- Registers. As in any programming language, we need variables to work with. These variables store data (in our case, 32-bit values). We can set and modify register values and perform operations with and between these registers (for example, we can add the values in two registers and place the result in a third register, or in one of the registers that already stores an operand). Some registers store a memory location (for example, we need to know where the next program instruction to be executed is located, so we have an Instruction Pointer – effectively an "Instruction Register");
- Instructions. The processor supports a number of operations between registers and memory locations. First, we need the ability to move data from/to registers to/from memory. In registers, we can perform mathematical, logical, I/O, and other operations.
- Flags. After performing an operation, we have a series of flags (indicators) that are set depending on the result (whether the result of the operation is negative or zero, or whether an overflow occurred);
- Memory addressing. The processor contains only a few registers. If we need to work with a large amount of data (for example, the contents of a file), then we need to be able to refer to an address in memory (*Random Access Memory – RAM*). Accessing registers is fast, but reading/writing data from/to RAM is a much slower operation;

¹⁰Intel® 64 and IA-32 Architectures Software Developer Manuals, <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

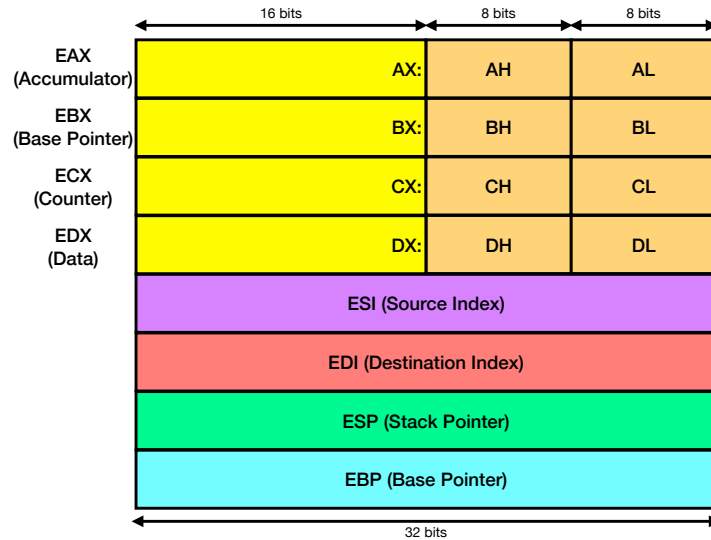
¹¹<http://asm.sourceforge.net/articles/linasm.html#Syntax>

- The program stack. Every running program has a special memory region called the stack (*the program stack*). This is a *Last In First Out (LIFO)* data structure. We will later see examples where this structure is useful (function calls in Assembly);
- Interrupts. The program you execute (regardless of the programming language) does not have direct access to all hardware resources. In most cases, the operating system controls access to memory and peripherals. Therefore, we need to ask the operating system to perform certain operations for our program. This delegation of work to the operating system is performed through interrupts.

Next, we discuss all these components in detail.

3.1 x86 Architecture Registers

The registers of an x86 system can be summarized in the following diagram:



The processor architecture is 32-bit, so the registers used are also 32-bit. A register represents a small amount of storage space on the central processing unit, whose contents can be accessed more quickly than data stored elsewhere (for example, in memory or, even more slowly, on a hard disk).

The registers of the x86 architecture are:

- the 8 shown in the image above, called general-purpose registers (*General-Purpose Registers*);
- a register for *flags*;
- a register that points to the next instruction to be executed (*Instruction Pointer*);
- other registers.

3.1.1 General-Purpose Registers

Register	Name	Description	Restored after call
AX	accumulator	used in arithmetic operations	no
BX	base	used as a pointer to data	yes
CX	counter	used in repetitive instructions	no
DX	data	used in arithmetic and I/O operations	no
SP	stack pointer	pointer to the top of the stack	yes
BP	stack base pointer	pointer to the base of the stack	yes
SI	source index	pointer to the source in stream operations	yes
DI	destination index	pointer to the destination in stream operations	yes

Observation. The **BP** register can also be used as a *frame pointer*.

Another observation. Looking at the table, we notice that registers in the 32-bit x86 architecture have the prefix **E**. In this case, the **EAX** register contains **AX** (*accumulator*), **AH** (*H – high*), and **AL** (*L – low*). For example, if we store the value 0 in **EAX**, but later place the value 1 in **AH**, then the entire **EAX** register will store the decimal value 256. **Why?**

3.1.2 EFLAGS

The EFLAGS registers (indicators) are:

- **CF** – *carry flag*, which is set if a carry occurred after the last operation;
- **PF** – *parity flag*, which is set if the number of 1 bits in the result of the last operation is even;
- **ZF** – *zero flag*, which is set if the result of the last operation was 0;
- **SF** – *sign flag*, which is set if the result of the last operation was negative;
- **OF** – *overflow flag*, which is set if the result of the last operation produced an overflow.

3.1.3 Addresses

To represent memory addresses, we can use one of the following forms:

- the value of a 32-bit register: `(%eax)`;
- the sum of a numeric constant and the value of a register: `4(%eax)` (understood as `%eax + 4`);
- the sum of two registers: `(%eax, %edx)` (understood as `%eax + %edx`);
- the sum of two registers and a numeric constant: `4(%eax, %edx)` (understood as `%eax + %edx + 4`);
- the sum of two registers, one of which is multiplied by 2, 4, or 8, to which a constant may be added: `16(%eax, %edx, 4)` (understood as `%eax + 4 * %edx + 16`);

3.1.4 Notation

It is important to note that, when writing programs, registers are prefixed by the symbol `%`. Thus, to use the **EAX** register, we write `%eax` (we will preserve the lowercase notation convention).

Equally important: just as registers are preceded by the symbol `%`, numeric constants are preceded by the symbol `$`. Thus, we write `$4` to represent the constant 4.

3.2 Data Types and the Basic Structure of a Program

3.2.1 Data Types

The main data types we will use are

1. **byte** – as the name suggests, it occupies 1 byte, i.e. 8 bits in memory;
2. **single** – occupies 4 bytes (32 bits) and is used to store fractional numbers;
3. **word** – occupies 2 bytes (16 bits) and is used to store integers;
4. **long** – occupies 4 bytes (32 bits) and is used to store 32-bit integers or 64-bit fractional numbers. **Double word** (dword) also occupies 32 bits;
5. **quad** – occupies 8 bytes (64 bits);
6. **ascii** – used to declare character strings that are not terminated by the string terminator – not recommended;
7. **asciz** – used to declare character strings that are terminated by the string terminator;
8. **space** – defines a region in memory whose size we specify, for example **.space 4**, meaning that 4 bytes = 32 bits are reserved. It is useful when declaring variables that will be computed within the program and for which we do not initially want a default value.

To declare a variable, the syntax is

```
name: .type value
```

for example

```
x: .word 15
y: .floating 23.74
str: .asciz "String"
```

3.2.2 Program Structure

The basic structure of an x86 Assembly program is:

```
.data
    ;// here we declare the program variables

.text
    ;// the actual code starts here
    ;// we define main as a global label
    ;// representing the entry point of our program

    ;// in principle, this is where we will write the procedures
    ;// the main program is written here
    ;// the equivalent of main
.global main
main:
    ;// instructions
```

3.3 Example of an x86 Program

Important. Assembly-language instructions have the form

```
[[label] mnemonic/operation [operands]
```

3.3.1 The MOV Instruction

It is an instruction with the syntax

```
mov source, destination
```

with the effect of moving the source into the destination (assignment from left to right, *source* $\rightarrow$ *destination*). An interesting fact about this instruction is that **mov** is *Turing-complete*¹² (see also Problem 11 in the Proposed Exercises section).

The source and destination can be:

- **mov** register, register; (*mov %eax, %ebx*, i.e., $\%ebx \leftarrow \%eax$)
- **mov** memory address, register;
- **mov** register, memory address;
- **mov** numeric constant, register (*mov \$16, %edx*, i.e., $\%edx \leftarrow 16$);

Important. The MOV instruction can be suffixed with the size of the data type, so we can have **movl** (for long), **movq** (for quad), etc. Suffixed instructions also allow a form in which the source is a constant and the destination is a memory address.

3.3.2 System Interrupts

To write a first Assembly program, we must make sure that we can terminate its execution. Whereas in *medium-* and *high-level* languages the compiler handles this task, here it becomes the programmer's responsibility. The first interrupt we discuss is **SYSTEM EXIT**.

To invoke an interrupt, we use the **int** instruction. The most common one (the one we will use most often) is

```
int $0x80
```

which means **system call** (0x80 is hard-coded and associated with the interrupt that performs a system call). A system call means that we ask the operating system to do something for us (something our program does not have the “permission” to do directly).

For the system to know why it is being called, we must specify two more things:

- which function we are calling;
- what parameters that function has.

In general, we terminate C/C++ programs with **return 0;**, which is equivalent to the system call **exit(0)**. We therefore identify:

¹²Computerphile, <https://www.youtube.com/watch?v=RPQD7-A0jMI>

- the function being called is **exit**;
- its parameter (argument) is **0**.

Just as the system call itself is uniquely identified by the code 0x80, system functions are also uniquely identified by numbers. In this case, the **exit** function corresponds to number 1, and the **register** that stores which function is being called is **%eax**. Arguments are passed through the following registers: **%ebx**, **%ecx**, etc. (you can consult this table to see the function parameters ¹³

Putting this information together, the system interrupt for terminating the program is:

```
mov $1, %eax
mov $0, %ebx
int $0x80
```

To summarize:

- when the system detects interrupt 0x80, it understands that it has a **system call** and checks its registers;
- the **%eax** register stores the code of the function it must call; in this case, code **1** means the **exit** function. How do we know this code? We look it up in the operating-system documentation¹⁴;
- knowing that it is calling **exit**, which has one argument, it looks in **%ebx** to read that argument's value.

Observation: there are different types of system calls, which we will explain at the appropriate time. For example, displaying output on the screen / in a text file is a system call, reading from the keyboard / from a text file is a system call, etc.

Observation: What happens if we have a system call with many parameters? Remember that we have only a small number of registers. Things become slightly more complicated if we have more than 5 parameters¹⁵.

3.3.3 A First Program: Source Code

With all the previous information in place, we will write our first complete assembly-language program:

```
.data
.text
.global main
main:
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

To run the program, we execute the following commands in the terminal:

¹³https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit

¹⁴https://github.com/opennetworklinux/linux-3.8.13/blob/master/arch/sh/include/uapi/asm/unistd_32.h

¹⁵<http://asm.sourceforge.net/articles/linasm.html#Syscalls>

```
gcc -m32 program_exit.asm -o program_exit
```

```
./program_exit
```

Observation: For the 32-bit version to work while the operating system is 64-bit, you will need to install an additional library using the command **sudo apt-get install g++-multilib**.

Observation: If, while running the gcc command above, you encounter the error:

```
/usr/bin/ld:program.asm: file format not recognized; treating as linker script
```

```
/usr/bin/ld:program.asm: syntax error
```

```
collect2: error: ls returned 1 exit status
```

Then you must rename the file to program_exit.S (not .asm, but .S) and try the gcc command again.

Observation: After compilation, you may see a warning such as:

```
/usr/bin/ld: warning: relocation in read-only section '.text'
```

```
/usr/bin/ld: warning: creating DT_TEXTREL in a PIE
```

This warning is generated by a memory-protection mechanism called PIE (Position Independent Executables). We will discuss what this means later in the course.

To avoid these warnings, it is enough to add the **-no-pie** flag to the command, so that it has the following form:

```
gcc -m32 program_exit.s -o program_exit -no-pie
```

```
./program_exit
```

3.4 ASM Programs and Debugging

To better understand how registers change their values, and also to correct certain *bugs* when they appear in the code we write, it is important to know how to use a *debugger*. *Debugging* is an activity every programmer encounters, regardless of the programming language used. To begin, consider the following program written in assembly language (let us name the file **program2.asm**):

```
.data
    x: .long 15
.text
.global main
main:
    movl $0, %eax
    movl %eax, x
    movl x, %ebx

    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

We execute

```
gcc -m32 program2.asm -o program2
```

To debug the program, we will use **gdb**. Run the following in the terminal

```
gdb program2
```

We set a breakpoint at the program entry address, namely at **main**:

```
b main
```

We run the program

```
run
```

And now we will execute, in succession,

```
i r
stepi
```

(i.e. next step and information registers) to debug *step by step*. Observe how the values of registers `%eax` and `%ebx` change.

3.4.1 Another Program

```
.data
.text
.global main
main:
    movl $0, %eax
```

```
mov $1, %ah
```

```
mov $1, %eax
```

```
mov $0, %ebx
```

```
int $0x80
```

What is stored in register %eax after executing the instruction `mov $1, %ah`? Why? What would the value of %eax have been if we had used %al instead of %ah?

3.5 Hello, world! in Assembly

In this section, we will make a system call to display a character string on the screen.

Just as for **SYSTEM EXIT**, **PRINT STRING** will have a function code (in %eax) and a set of arguments (in this case there are 3: %ebx, %ecx, and %edx).

Thus, to display a string on the screen, we follow these steps:

- load the value 4 into register %eax, corresponding to the *write* function (see the link above where system calls and their associated codes are listed);
- load the value 1 into register %ebx, corresponding to the destination where we display the text – the console, i.e. **standard output**;
- load the message to be displayed into %ecx, in this case a string from memory;
- load the length of the displayed string into %edx. Be careful when declaring it with `.asciz`, because the string terminator is also counted in the length.

Consider the following complete Assembly program:

```
.data
    helloWorld: .asciz "Hello, world!\n"
.text
.global main
main:
    mov $4, %eax
    mov $1, %ebx
    mov $helloWorld, %ecx
    mov $15, %edx
    int $0x80

    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Note:

Test your knowledge so far by taking laboratory tests 2.1, 2.2, and 2.3, which you can find online in the laboratory section.

3.6 Arithmetic and Logical Operations

3.6.1 Arithmetic Operations

In this part of the laboratory, we present the four basic arithmetic operations – addition, subtraction, multiplication, and division, including quotient and remainder.

Instruction	Effect
add op1, op2	$op2 := op2 + op1$
sub op1, op2	$op2 := op2 - op1$
mul op	$(edx, eax) := eax \times op$
imul op	$(edx, eax) := eax \times op$
div op	$(edx, eax) := (edx, eax) / op$
idiv op	$(edx, eax) := (edx, eax) / op$

Observation: for `div`, the notation (edx, eax) means that $2^{32} * edx + eax$ is divided by `op`. If we do not want to divide a very large number, we must initially place the value 0 in `edx`.

Multiplication and division explanations.

- both multiplication and division have implicit operands, namely `EDX` and `EAX`. To multiply a number by a certain value (or divide a number by a certain value), the relevant information must already be placed in register `EAX`;
- we must distinguish between `MUL` and `IMUL`, and between `DIV` and `IDIV`; the instruction prefixed by `I` is used for processing **signed** numbers;
- for multiplication, the result is given by $2^{32} \times edx + eax$;
- for division, `eax` stores the quotient, while `edx` stores the remainder;
- in multiplication and division, the operand `op` **CANNOT** be a numeric constant! Otherwise, `op` may be a register or a memory location.

3.6.2 Example

Write a program that computes the sum, difference, product, as well as the quotient and remainder of the division of two signed integers.

Solution.

```
.data
    x: .long 30
    y: .long 7
    sum: .space 4
    dif: .space 4
    prod: .space 4
    cat: .space 4
    rest: .space 4
.text
.global main
```

```

main:
    mov x, %eax
    add y, %eax
    mov %eax, sum

    mov x, %eax
    sub y, %eax
    mov %eax, dif

    mov x, %eax
    mov y, %ebx
    imul %ebx
    mov %eax, prod

    mov x, %eax
    mov y, %ebx
    mov $0, %edx
    idiv %ebx
    mov %eax, cat
    mov %edx, rest

    mov $1, %eax
    mov $0, %ebx
    int $0x80

```

Important. To inspect the value stored in variables declared with `.space 4`, we can again use the `gdb` debugger, executing at a step:

```
print (long) variable_name
```

for example,

```
stepi
print (long) sum
```

3.6.3 Logical Operations

The logical operations are **and**, **or**, **xor**, and **not**; the first three have the structure **operation** **source**, **destination**, while **not** has the structure **not** **destination**.

Instruction	Effect
not op	op := ~op
and op1, op2	op2 := op2 & op1
or op1, op2	op2 := op2 op1
xor op1, op2	op2 := op2 ^ op1

Requirement. Write a program to test the four operations above. Instead of **and**, also use the **test** operation and compare, using the `gdb` debugger, its effect with that of **and**.

3.6.4 Logical Shift Operations

The logical shift operations are **shr**, **shl**, **sar**, and **sal**; all have the structure **operation number, destination**, where *number* is the number of bits to shift. In the instruction name, s stands for *shift*, the l/r distinction stands for left/right, and the h/a distinction stands for logical/arithmetic (bitwise or arithmetic shifting). In arithmetic shifts, the most significant bit preserves its value (the sign of the number is preserved).

Instruction	Effect
shr number, op	op := op $\gg$ number
shl number, op	op := op $\ll$ number
sar number, op	op2 := op $\gg$ number (preserving the sign of op)
sal number, op	op2 := op $\ll$ number (preserving the sign of op)

Requirement. Write a program to test the four operations above.

3.7 Conditional and Unconditional Jumps

3.7.1 Unconditional Jumps

In many situations, we need to jump to a certain instruction in the code (think of if statements and loops). In this section, we discuss the ways in which jumps can be performed in code.

This is performed using the instruction

```
jmp et
```

jumping to the program label **et** (a jump is performed in memory, without taking **IP** into account).

For example:

```
.data
    text1: .asciz "Text 1\n"
    text2: .asciz "Text 2\n"
.text
.global main

main:
    jmp et2
    mov $4, %eax
    mov $1, %ebx
    mov $text1, %ecx
    mov $8, %edx
    int $0x80

et2:
    mov $4, %eax
    mov $1, %ebx
    mov $text2, %ecx
    mov $8, %edx
    int $0x80

    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

In this case, the value displayed on the screen is the string “Text 2”.

3.7.2 Conditional Jumps

The instruction is

```
j<condition> et
```

and it is executed only if the required condition holds, jumping to the label **et**. Otherwise, the next line of code is executed.

Both the *flags* already introduced – carry, overflow, zero, sign – can be tested, and instructions corresponding to relational operators (<, <=, >, >=, !=, ==) can also be executed.

Operator	Description
jc	jump if carry is set
jnc	jump if carry is not set
jo	jump if overflow is set
jno	jump if overflow is not set
jz	jump if zero is set
jnz	jump if zero is not set
js	jump if sign is set
jns	jump if sign is not set
Operators for unsigned numbers	
jb	jump if below (op1 < op2)
jbe	jump if below or equal (op1 <= op2)
ja	jump if above (op1 > op2)
jae	jump if above or equal (op1 >= op2)
Operators for signed numbers	
jl	jump if less than (op1 < op2)
jle	jump if less than or equal (op1 <= op2)
jg	jump if greater than (op1 > op2)
jge	jump if greater than or equal (op1 >= op2)
Equality operators	
je	jump if equal (op1 == op2)
jne	jump if not equal (op1 != op2)

Observation. To use one of the operators above, we must be able to compare values. In this case, we use the **cmp** instruction.

Attention! If we compare **cmp %eax, %ebx**, then the $\leq$ operation must be interpreted as $\%ebx \leq \%eax$.

```
.data
    a: .long 5
    b: .long -3
    text1: .asciz "a >= b\n"
    text2: .asciz "a < b\n"
.text

.global main

main:
    mov a, %eax
    mov b, %ebx
    cmp %ebx, %eax
    jge et

    mov $4, %eax
    mov $1, %ebx
    mov $text2, %ecx
    mov $7, %edx
```

```

    int $0x80

    jmp exit

et:
    mov $4, %eax
    mov $1, %ebx
    mov $text1, %ecx
    mov $8, %edx
    int $0x80

exit:
    mov $1, %eax
    mov $0, %ebx
    int $0x80

```

Requirement. Run the program above and execute it with the debugger to see how the values change.

3.7.3 Simulating Repetitive Structures

Suppose we want to simulate the following program developed in C:

```

#include <stdio.h>

int main()
{
    int n = 5, s = 0;
    for (int i = 0; i < n; i++)
    {
        s += i;
    }
    printf("%d\n", s);
    return 0;
}

```

3.7.4 Simulating Repetitive Structures: Solution 1

In the following Assembly program, we will see how to “simulate” computation loops using jump instructions and appropriately defined labels.

```

.data
    n: .long 5
    s: .space 4
.text

.global main

main:

```

```

    mov $0, %ecx
    ;// ecx was the counter register

etloop:
    cmp n, %ecx
    je etexit
    ;// if %ecx == n, we have finished the traversal

    add %ecx, s
    ;// s += %ecx

    add $1, %ecx
    ;// %ecx += 1

    jmp etloop;
    ;// restart the traversal

etexit:
    mov $1, %eax
    mov $0, %ebx
    int $0x80

```

We can use the debugger to see the result. Suppose the program is called **suma.asm** and we have reached the point where we run it with **./suma**. We can use **gdb** as follows:

```

gdb suma
b etexit
run
print (int) s

```

3.7.5 Simulating Repetitive Structures: Solution 2

A solution closer to the language-specific features uses the *keyword* **loop**. Its action is simple: we write **loop label**, meaning that iteration continues at the given label (in the previous program, this was **etloop**), and execution stops when register **%ecx** becomes 0. Note that **%ecx** is decremented automatically; we do not need to make this change manually. We therefore rewrite the previous program:

```

.data
    n: .long 5
    s: .space 4
.text

.global main

main:
    sub $1, n
    mov n, %ecx

```

```
etloop:
    add %ecx, s
    loop etloop
    jmp etexit

etexit:
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Requirements:

1. Explain why the instruction `sub $1, n` is necessary (check the x86 documentation for the *inc* and *dec* instructions);
2. Run the program in **gdb** to verify that it is correct.

Note:

Test your knowledge so far by taking laboratory tests 3.1 and 3.2, which you can find online in the laboratory section.

3.8 One-Dimensional Arrays

To discuss one-dimensional arrays (of integers, for now), we need to understand memory addressing and learn a new operation – Load Effective Address (LEA).

3.8.1 Memory Addressing

Consider the following notation

`a(b, c, d)`

which we interpret as $b + c \times d + a$. More precisely, if we express it using registers and constants, `-8(%ebx, %eax, 4)` means $\%ebx + \%eax \times 4 - 8$, while the notation

`mov $5, -8(%ebx, %eax, 4)`

means that we store the constant 5 at the memory address computed as $\%ebx + \%eax \times 4 - 8$.

Sometimes, some information may be omitted, in which case we may have, for example, the instruction

`mov $0, (%edi, %ecx, 4)`

Attention! When working with *arrays*, we cannot use only `mov`; we must suffix it with the required data type (for example, `movl` for *arrays* whose elements have type **long**).

3.8.2 Load Effective Address (LEA)

The `lea` instruction loads the address of the source into the destination and has the form

`lea source, destination`

For example,

`lea v, %edi`

loads the address of `v` into register `%edi`.

3.8.3 Memory Representation of One-Dimensional Arrays

One-dimensional arrays are stored linearly in memory, starting from an initial address and successively occupying as many memory locations as required by the data type used.

For example, if the data type is **long**, then each array element occupies a location of size 4 (long uses 4 bytes). In this case, relative to the initial memory location, the elements are found as follows:

- the first element is at the initial memory location, MEM, plus the size of the data type multiplied by the ordinal number of the element, i.e. $\text{MEM} + 4 \times 0$;
- the second element is at memory location MEM plus the product of 4 (the size of the data type) $\times 1$ (the current index): $\text{MEM} + 4 \times 1$;
- the third element is at $\text{MEM} + 4 \times 2$;
- etc.

We will store the memory address in register `%edi` (destination index register), and we will use register `%ecx` as the current-index counter. In this case, using the triplet notation introduced above, we obtain the elements in the form `(%edi, %ecx, 4)`, i.e. $\%edi + 4 \times \%ecx$.

3.8.4 Declaring One-Dimensional Arrays

They are declared as a label in memory, with the elements listed in sequence:

```
v: .long 10, 20, 30, 40, 50
```

and, in general, we will also store the number of elements in memory:

```
n: .long 5
```

3.8.5 Examples: Program 1

Consider the following complete Assembly program:

```
.data
    n: .long 5
    v: .long 10, 20, 30, 40, 50
.text
.global main

main:
    lea v, %edi
    mov $0, %ecx
    movl (%edi, %ecx, 4), %edx

etexit:
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Run this program with **gdb** and find the value stored in `%edx` at the **etexit** label. Change the counter value and observe the linear arrangement of the elements in memory.

3.8.6 Examples: Program 2

Consider the following complete Assembly program:

```
.data
    n: .long 5
    v: .long 10, 20, 30, 40, 50
.text
.global main

main:
    lea v, %edi
    mov n, %ecx

etloop:
    mov n, %eax
    sub %ecx, %eax
    movl (%edi, %eax, 4), %edx
```

```
    loop etloop

etexit:
    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

Requirements.

1. Understand what this program does, explaining why register `%eax` is necessary.
2. Run it in **gdb** using the instructions

```
gdb program_name
b etloop
i r
c
```

The **c** command is used to continue debugging, so execution will always jump to the **etloop** label. With **i r**, we can inspect the contents of the registers in order to analyze the values stored in `%edx`.

3. Can you find the value 50 while keeping only this breakpoint? What should be changed in order to also observe `%edx = 50`?

Note:

Test your knowledge so far by taking laboratory tests 4.1, 4.2, and 4.3, which you can find online in the laboratory section.

3.9 About Inline Assembly

x86 Assembly can also be used to write *inline assembly* directly in C files. All that is required is to add the `__asm__` directive, inside which assembly instructions will be written, as in the current laboratory. In the example below, we start with a variable `x` and add 1 to the current value it stores.

Observation. For now, ignore the *pusha* and *popa* instructions; they will be explained in the following laboratory sessions. As another observation, try moving variable `x` inside the *main()* function and observe the effect.

Suppose we have the following program, saved in the file **file.c**:

```
#include <stdio.h>
int x = 1;
int main()
{
    __asm__
    (
        "pusha;"

        "mov x, %eax;"
        "add $1, %eax;"
        "mov %eax, x;"

        "popa;"
    );

    printf("%d\n", x);
    return 0;
}
```

The program will be compiled and executed with

```
gcc file.c -o file -m32
./file
```

Requirement. Implement other programs from this laboratory material using **inline assembly**.

4 Proposed Exercises

4.1 Laboratory 1

1. Given two natural numbers **x** and **y**, write an assembly program that swaps them. Observe the effect using the debugger.
2. Let **s** be a 12-byte allocated space. Using the function code for read (see the link above where the system calls and associated codes are listed) and knowing that the loading convention is the same as for the write function, read 12 characters from the keyboard into **s** (the code for stdin is 0). Display **s**.

4.2 Laboratory 2

1. Let **x**, **y** be two natural numbers stored in memory. Write a program that computes, in two different ways, $(x/16) + (y \times 16)$.
2. Add a final step to the previous program to check whether the two computed values are equal. If they are, display "PASS"; otherwise, display "FAIL".
3. Let **a**, **b**, **c** be three numbers stored in memory and **min** a 4-byte allocated space. Store the smallest of the three numbers in **min**.
4. Check whether a given number is prime (use both variants of repetitive structures).

4.3 Laboratory 3

1. Determine the maximum value in an array and the number of times it occurs.
2. Let **s** be a string stored in memory and **t** an allocated space with the same number of bytes. Store the reverse of string **s** in **t** and display it on the screen.

5 Additional Exercises

1. Write x86 Assembly code sequences that compute the following:
 - (a) $EAX \leftarrow EAX \& 1$
 - (b) $EAX \leftarrow EAX \wedge EAX$
 - (c) $EAX \leftarrow (EAX \wedge EBX) \wedge EBX$
 - (d) $EAX \leftarrow EAX \& (EAX - 1)$
 - (e) $EAX \leftarrow (EAX \& \sim EBX) \mid (\sim EAX \& EBX)$
 - (f) $EAX \leftarrow (EAX \& \sim EAX) \mid (\sim EBX \& EBX)$
 - (g) $EAX \leftarrow (EAX \mid \sim EBX) \& (\sim EAX \mid EBX)$
 - (h) $EAX \leftarrow (EAX \mid \sim EAX) \& (\sim EBX \mid EBX)$
 - (i) $EAX \leftarrow EBX \wedge ((EAX \wedge EBX) \& \neg(EAX < EBX))$
 - (j) $EAX \leftarrow EAX \wedge ((EAX \wedge EBX) \& \neg(EAX < EBX))$
 - (k) $ECX \leftarrow EAX + EBX - ECX$
 - (l) $EBX \leftarrow ((EAX \times 2 + EBX) \times 2 + ECX)/2$
 - (m) $EAX \leftarrow (EAX + EBX + ECX) \times 16$
 - (n) $ECX \leftarrow (EAX/16) + (EBX \times 16)$
 - (o) $ECX \leftarrow (EAX \& 0xFFFF0000) + (EBX \& 0x0000FFFF)$
 - (p) $EAX \leftarrow (EAX + EBX)/(EAX + ECX)$
 - (q) $EAX \leftarrow (EAX \times EBX)/(ECX^2)$
 - (r) $EAX \leftarrow EAX \times ((EAX + EBX)/ECX + 1)$
 - (s) $EAX \leftarrow 1 + EAX + EAX^2/2 + EAX^3/6 + EAX^4/24 + \dots$
2. Register EAX contains an integer. Check whether the number is negative, positive, or zero.
3. Registers EAX and EBX contain two distinct values. Implement SWAP(EAX, EBX) using only logical operations and no additional register or memory. (Check whether x86/x87 already has an instruction that performs this operation; if so, think about why this operation is important.)
4. Register EAX contains a natural number. Find the smallest power of 2 greater than this number. (Check whether x86/x87 already has an instruction that performs this operation; if so, think about why this operation is important.)
5. Given a natural number in register EAX, count how many “1” bits occur in the binary representation of the number. (Check whether x86/x87 already has an instruction that performs this operation; if so, think about why this operation is important.)
6. Given a strictly positive natural number in register EAX, compute $\lfloor \log_2 EAX \rfloor$. (Check whether x86/x87 already has an instruction that performs this operation; if so, think about why this operation is important.)
7. Given a natural number in register EAX, compute the length of the longest sequence of consecutive “1” bits in the binary representation of the number.

8. Given a natural number in register EAX, check whether the number is a palindrome (in its binary representation).
9. Given a signed integer, determine all of its divisors.
10. Using only **mov** instructions, obtain the number 2318 in a register. (Do not use the instruction `mov $2318, %eax`.)
11. Determine the n -th term of the Fibonacci sequence, where n is read from the keyboard. It is not guaranteed to be a positive integer.
12. Compute the sum of the elements in a one-dimensional array. Also compute the arithmetic mean of the array elements (division using quotient and remainder).
13. Compute $EAX \leftarrow EAX!$ (EAX factorial).
14. Determine the two smallest elements in an array using a single traversal.
15. Consider an array of odd length stored in memory. We know that every element occurs twice except one; for example, $v = \{ 1245, 342, 543523, 342, 4234, 1245, 543523 \}$, where 4234 is the unique element. Write an efficient algorithm that determines this element (space complexity $O(1)$ and time complexity $O(n)$, where n is the length of the array).
16. Using the binary-search algorithm, check whether an array contains a given element. If it does, determine the position of that element; otherwise, the position should be -1.
- 19*. Compute the integer square root of a number¹⁶. (For a fast approximate square-root computation, see a famous piece of code: the source code for `Q_rsqrt` from Quake¹⁷.)
- 20*. Sort a one-dimensional array. Write a C program that uses as many inline-assembly instructions as possible to sort a one-dimensional array (pure ASM and inline ASM).
- 21*. Following in the footsteps of the first programmer, Ada Lovelace, determine the n -th Bernoulli number¹⁸.

¹⁶https://en.wikipedia.org/wiki/Integer_square_root

¹⁷`Q_rsqrt`, https://github.com/id-Software/Quake-III-Arena/blob/dbe4ddb10315479fc00086f08e25d968b4b43c49/code/game/q_math.c

¹⁸<https://writings.stephenwolfram.com/2015/12/untangling-the-tale-of-ada-lovelace/>

6 Additional Resources

Additional tutorials and explanations can be found at the following online resources.

For AT&T syntax:

1. Adam Ferrari, x86 Assembly Guide¹⁹;
2. Alexander Mishurov, AT&T assembly syntax and IA-32 instructions²⁰.

For Intel syntax:

1. Vlad Radulescu, ASM Laboratories²¹;
2. David Evans, x86 Assembly Guide²²;
3. Cristian Botau, Introduction to Assembly²³.

¹⁹<http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html>

²⁰<https://gist.github.com/mishurov/6bcf04df329973c15044>

²¹<https://sites.google.com/view/danton-asm/asm1?authuser=0>

²²<http://www.cs.virginia.edu/~evans/cs216/guides/x86.html>

²³<https://www.infoarena.ro/introducere-in-asamblare>