Tartu Narva College

Homeworks

Assignments

Computer Hardware

By: Sergei Ivanov

IT systems development

Prof.: Chen-Wan Lin

18.09.2023

# Preface

The reason why I wrote this paper is due to organization of my homework assignments. It is better to place all coherent data in one entire document, which simplifies reading and updating existent knowledge.

The source is available at the [link](https://github.com/derweisskrag/ComputerHardwareColleg).

# Revision history

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Organization/Point of Contact | Description of Changes |
| 1.0 | 02.10.2023 | Sergei Ivanov | Added the following sections:   1. Revision history 2. Preface 3. The implementation of XOR |
| 2.0 | 04.10.2023 | Sergei Ivanov | Added the following information: Python scripts in XOR implementation section. |
| 3.0 | 10.10.2023 | Sergei Ivanov | Add the following sections:   1. Homework IV, 2. Memoization, 3. Cache, 4. Negative numbers, 5. Daemon section |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table 1: Revision history

# Table of Contents

[Preface 2](#_Toc147898049)

[Revision history 3](#_Toc147898050)

[Table of Contents 4](#_Toc147898051)

[Abstract 7](#_Toc147898052)

[Goals of this paper 8](#_Toc147898053)

[Preview of homework assignments 9](#_Toc147898054)

[Homework I 9](#_Toc147898055)

[Theory 9](#_Toc147898056)

[Practice 10](#_Toc147898057)

[Homework II 10](#_Toc147898058)

[Theory 10](#_Toc147898059)

[Practice 11](#_Toc147898060)

[Homework III 12](#_Toc147898061)

[Homework IV 12](#_Toc147898062)

[Introduction 13](#_Toc147898063)

[Computers 13](#_Toc147898064)

[Turing’s Machine 13](#_Toc147898065)

[Programming Language 14](#_Toc147898066)

[Memory 15](#_Toc147898067)

[Hardware 15](#_Toc147898068)

[Supercomputers 16](#_Toc147898069)

[Definition 16](#_Toc147898070)

[Examples 16](#_Toc147898071)

[CPU 17](#_Toc147898072)

[OS 18](#_Toc147898073)

[Compilation process 18](#_Toc147898074)

[Lexical Scope 19](#_Toc147898075)

[Compilation subtasks 21](#_Toc147898076)

[Assembly 21](#_Toc147898077)

[Introduction 21](#_Toc147898078)

[Registers 22](#_Toc147898079)

[Hello World Program 23](#_Toc147898080)

[Printing Strings 24](#_Toc147898081)

[Macros 25](#_Toc147898082)

[Print Numbers 25](#_Toc147898083)

[Arithmetic operations 28](#_Toc147898084)

[Binary Operations 30](#_Toc147898085)

[Basics 30](#_Toc147898086)

[Negative binary numbers 33](#_Toc147898087)

[Implementation in Python 34](#_Toc147898088)

[Iterative approach 34](#_Toc147898089)

[2-complement binaries 39](#_Toc147898090)

[Recursion 39](#_Toc147898091)

[Binary vector 40](#_Toc147898092)

[Assembly 41](#_Toc147898093)

[32-bit floating-point format 42](#_Toc147898094)

[The implementation of XOR 43](#_Toc147898095)

[Introduction 43](#_Toc147898096)

[Applications of XOR 44](#_Toc147898097)

[Linear congruential distribution 44](#_Toc147898098)

[XOR Shift generator for 32-bit 45](#_Toc147898099)

[The implementation of XOR 46](#_Toc147898100)

[Python 46](#_Toc147898101)

[Assembly 47](#_Toc147898102)

[Optimization 49](#_Toc147898103)

[Introduction 49](#_Toc147898104)

[Memoization 49](#_Toc147898105)

[Overview of the technique 49](#_Toc147898106)

[Examples 50](#_Toc147898107)

[Cache 51](#_Toc147898108)

[What is Cache? LRU cache? 51](#_Toc147898109)

[Cache levels 52](#_Toc147898110)

[Cache lines 53](#_Toc147898111)

[Lazy loading 56](#_Toc147898112)

[Iterators 56](#_Toc147898113)

[Generators 57](#_Toc147898114)

[Implementation 57](#_Toc147898115)

[Optimization of loops 58](#_Toc147898116)

[Loops 58](#_Toc147898117)

[Examples of algorithms 59](#_Toc147898118)

[Daemon 59](#_Toc147898119)

[Homework 60](#_Toc147898120)

[Part I 60](#_Toc147898121)

[Part II 62](#_Toc147898122)

[Part III 68](#_Toc147898123)

[Part IV 68](#_Toc147898124)

[References 72](#_Toc147898125)

[Primary 72](#_Toc147898126)

[Secondary 72](#_Toc147898127)

# Abstract

The paper is about gain valuable insight into computer science related to hardware engineering. Alan Turing is the creator of Turing’s machine as a mathematical model, and his contributions to CS are of great importance: engineers realized that using RAM is better than Turing’s infinite tape. Apart from modern computers, there are super computers that are widely used by scientists to deepen their scientific research. One of the greats supercomputers is CRAY-1 proposed by Cray, as it was quite efficient and successful machine sold across the world until 1990s. Cray tempted to develop a scientific programming language, which is later known as FORTRAN. Others such languages include Python, C++, Rust, and others. It is interesting to point out that Python can work together with FORTRAN, which now contains a great amount of “legacy code” (the code that is old but very critical not to overwrite because of compatibility issues). The programming language is a language that is easily understood to humans and is used to send instructions to CPU and then execute them. There are two types of programming languages: 1) interpreted and 2) compiled. For example, Python is an interpreted one, whereas C++ is compiled into machine code; yet, I have to admit that Python is written in CPython programming language, and the source code of Python is compiled into bytecode and then executed by the interpreter. To enhance performance, CPU can distribute the workload on the computer machine system components and each one can send signals to CPU which can signal about an action – dispatching an action based on a state; some components might include: 1) GPU, 2) TPU, 3) QPU and others. Also, we study Assembler and binary operations to gain more insight into CS. Keywords: supercomputer, programming language, hardware, performance, CPU, GPU, RAM, location, year, creator, purpose, Assembler, Python, OS, MS-DOS, UNIX, Linux, binary, 32-bit, 8-bit, floating-point, XOR, AND, SHIFT, and inverting bits.

# Goals of this paper

In this section, I would like to clarify the goal of this paper:

1. Present theoretical knowledge and cite sources of the searched information; by this, you can understand:
   * + 1. forking with existing research paper.
       2. building on top of another search, thereby extending one’s academic paper, and gaining new insights into the subject.
2. Solve homework assignments.

Each homework assignment contains theory and practice. Every time I complete the assignment, I add it to the academic paper, becoming a book.

Methods are quite simple: searching for academic papers and extending them. All sources of information are academic papers and cited as references to my data, which I do not quote, but talk about it by myself.

For writing code, it is generally acceptable to edit and write code here, in the Word document. I will not share screenshots of my code editor; rather, I demonstrate the algorithms here.

# Preview of homework assignments

## Homework I

### Theory

To gain insight into computer hardware, we have to account for various terms used in Computer Science (hereinafter, CS). This is of great importance, because without a solid foundation in CS, it might be challenging to develop, for example, a full-stack NEXT web applications. The full-stack term means here that NEXT (JS framework built on top of React, which is another framework used to build web applications) has two goals:

1. front-end
2. back-end

The first part is the UI/UX (User interface) which is used to display the design of a site; the other one is the server-side part. It is where NEXT server sits. The main purpose is to use SSR and SSG when rendering a page: when a user clicks a button, a request is sent to the server, and upon response, the content is displayed, which besides, the data comes from a database such as MongoDB or mySQL, or even SQLite. The last one is serverless, so the most popular choice is the mongoose or mySQL, using the NOSQL or SQL syntax, respectively. Thus, the full-stack term means that a software engineer develops both front- and back-ends.

One of the most important aspects are RAM, SSD, Bios, DOS, OS, C++/C, Signals, and CPU. Let us briefly refresh the information:

1. RAM – Random access memory, main memory.
2. SSD – Solid State Drive, secondary, memory; this is where most data is stored.
3. BIOS – Basic Input Output System.
4. DOS – Disk Operating System.
5. C++/C – a programming language (C++ has more OOP features than C).
6. Signals – a binary signal: 0 means false and 1 – true.
7. CPU - Central Processing Unit. Also:
   * + 1. TPU – Tensor Processing Unit.
       2. GPU – Graphical Processing Unit.
       3. QPU – Quantum Processing Unit.
8. OS – Operating System such as MS-DOS, Linux, Windows.

Lastly, we had to talk about supercomputers and how they are different from usual computer machines.

**Remark**. A computer can function without any OS as they did in the middle of XX century. The purpose of OS is to improve the performance and enable multiple users to make the most of the computer. Consequently, an OS can allocate and manage CPU resources efficiently, allowing the concurrent processing of multiple threads, enabling several users to utilize it simultaneously. In addition to facilitating multiprocessing, operating systems have also made computers accessible through user-friendly interfaces.

### Practice

The Homework I assignment was to create a data analysis of supercomputers; we had to include the following data:

1. CPU,
2. TPU/GPU/QPU,
3. Location, datetime,
4. Name,
5. Memory.

To create a custom computer, we had to fetch data and build a computer based on custom settings and preferences. This means that the data listed above should have been chosen and included in the PC build. Apart from the technical details, I also had to name the purpose of why I needed a PC, and for what tasks.

## Homework II

### Theory

The lecture was quite long: Binary operations. This section is all about bitwise operations, and how a computer handles those. The concepts that are to be covered are

1. XOR,
2. OR,
3. AND,
4. SHIFTS,
5. Least significant bit,
6. Binary vector.

To demonstrate these in practice, we are going to use Python, and Assembly. Therefore, it is important to gain insight into

1. I/O,
2. Registers,
3. …

Here is the example of Assembly Linux x86 code (OneCompiler, 2023, run and share the code) which runs the Hello, World program:

section .data

hello: db 'Hello, World!',10 ; 'Hello, World!' plus a linefeed character

helloLen: equ $-hello ; Length of the 'Hello world!' string

section .text

global \_start

\_start:

mov eax,4 ; The system call for write (sys\_write)

mov ebx,1 ; File descriptor 1 - standard output

mov ecx,hello ; Put the offset of hello in ecx

mov edx,helloLen ; helloLen is a constant, so we don't need to say

; mov edx,[helloLen] to get its actual value

int 80h ; Call the kernel

mov eax,1 ; The system call for exit (sys\_exit)

mov ebx,0 ; Exit with return "code" of 0 (no error)

int 80h;

### Practice

The second homework assignment is to implement the binary operations. Consider the following example:

or

where is the radix of the numeral system. In this case, it means binary, but because we have 4-digits numbers, it is 4-bit representation of the binary numbers.

Apart from addition, subtraction, multiplication, and division, you can implement various algorithms using bitwise operations:

1. Binary vector,
2. Remove duplicates.

**Remark.** Please, notice that modern CPU are highly optimized, which means that one does not have to rely on bitwise operations to improve the performance of a program. For modern computers, it is more efficient to 2 + 2 rather than 0b0010 + 0b0010. So, the use of bitwise operations DOES NOT necessarily enhance the efficiency. Simple additions or multiplications in the decimal numeral system can be faster, therefore, we should use bitwise operators only and only if it is appropriate. Consider

num **\*=** 2

num <<= 1

As it can be demonstrated, the first one is faster than the other one, because the simple arithmetic operation is highly optimized for modern CPU, 32-bit one.

To perform arithmetic operations, we can use both approaches:

1. Arithmetic operators
2. Binary operations

The choice depends on the problem that we are working on. In fact, the simplest arithmetic operation is better than bitwise addition, due to modern CPUs. For this particular homework assignment, I must demonstrate how computers add up numbers, more specifically, integers and floats. To do so, I am going to leverage the power of Python, and ASM; however, I must announce that Python will cover most of algorithms, rather than implementing floating addition in ASM. This is due to the period of time I have to fulfill the assignment. The teacher must conclude that ASM is a highly difficult language.

Remark. To obtain a negative binary number, we have to inverse all bits and add one to it. For example, ‘3’ is ‘0b0000\_0011’, but ‘-3’ would be ‘~0b0000\_0011 + 0b0000\_0001’ which is equal to ‘0b1111\_1100 + 0b0000\_0001’ or ‘0b1111\_1101’, and corresponds to 253 or ‘-3’.

## Homework III

I must account for binary operations and convert floating numbers into binaries following 32-bit twos complement floating standard which takes the following into consideration:

1. The sign bit: if a number is positive, it is 0, and otherwise 1;
2. If the number is negative then the number of shift to the right, we make is negative, so we adding to the exponent we have and then we must convert this decimal into binary form; it would be 8-bit representation and the middle part of the initial number 32-bit floating-point representation.
3. We must convert the fractional part into binary. This is done by multiplying the fractional part by 2 until we get an integer, or we can notice .

Then, we add 23 bits or less to get 23-bit fractional part. The result number is actually 32-bit twos complement floating-point representation of a floating number.

The second task is to implement XOR. We can do it by using logical operations or loop over bits of the number. The first one is also implemented in Assembly.

## Homework IV

During this homework assignment, we learn various techniques such as memoization, and cache to optimize our code and write more efficient and readable code to produce stable and secured software. We cover the following topics:

1. L-levels of cache;
2. Cache system types;
3. Parameters of cache;
4. Implement some algorithms and demonstrate the exact optimization.
5. Do daemons benefit from cache?

# Introduction

## Computers

### Turing’s Machine

Computers are machines specifically designed to simplify computational processes. This way, scientists could increase the performance of their calculations and enhance the quality of research papers – academic works that share a few valuable findings and insights into the concurrent technologies. Throughout human history, there were built and assembled many machines that assisted many professionals in carrying out their duties such as ancient temple’s doors that could open automatically at the specified time, relying on pure mechanical principles for their functionality. The Romans devised the first early vending machines that could operate over water supplies. Doubtless, those machines are completely different from today’s computers; but the evidence of human knowledge that led to the creation of the first computer systems is of great essence. It is a testament to human ingenuity and the foundation of computer science. There begs the question as to how scientists created those computers?

Alan Turing is the most prominent individual in the field of computer science; his contributions were significant to creating and testing the first Turing’s machines that eventually evolved into modern computers. What was Turing’s machine?

This is great question which helps us understand how computers work. Referred to as Turing’s machine, the infinite tape divided into many discrete cells is a theoretical mathematical model of computation introduced by Turing in 1930 [Source: English Wikipedia at [link](https://en.wikipedia.org/wiki/Alan_Turing)]. The machine showcased the following syntax of a programming language that we now refer to as Turing complete programming languages such as C, C++, C#, Rust, Python, Java, JavaScript, and others:

1. The definition of a variable (e.g., int x = 1); the tape could declare int, char, and Boolean variables. I would like to draw the reader’s attention to the fact that the Turing machine did not have string data type, and neither does C++; for this purpose, C++ developers created and deployed (released to commercial use) a standard string library “std::string”.
2. Read and Write functionality, also known as Read/Write Head, is the component of the machine that is used by machine to process the input data, for example, “0b0001” (1 in decimal numeral system), and write result of a computation, also, in binary: “0b0010” which is 2 in our numeral system, with radix or base equal to 10.
3. The crucial keyword is the state of the Turing’s machine based on which the machine could resume or perform an action. For example, in C, we can define a loop to perform a repetitive task, and either could Turing’s machine do so, but in slightly different way, because it did not have built-in loops: it was a tape, not a program written in C, compiled to machine code and executed with CPU. Thus, Turing’s machine was more lower abstraction level than C or Assembly which are used to perform calculations using CPU hardware of a computer machine.
4. Transition functions are critical aspect of Alan Turing’s machine, as they help it operate over binary data and perform actions. The similar concept is how JavaScript can be used to dispatch a state and perform an action based on this; it is then used in React JavaScript’s framework to 1) initialize a state; 2) update it if state changed due to React updated some properties. This way, React can monitor the state of some functional components and re-render if necessary upon the change, otherwise keep the state intact.

Alan’s machine could not assign values to variables [1] in the traditional sense: char h = “h” or int arr[] = {1,2,3,4,5}. It is also true that Turing could not write onto the tape his desired data like “0001\_0001” to get “0010\_0” as result of addition. The reader might wish to pose a question as to whether the question of Turing’s ability to write data onto the tape has any sense. It turns out that the question is meaningless [2]. It is because Turing could design tapes for each specific tasks such as add, multiply, subtract or divide that come with built-in transition functions helping the Turing’s machine to read and process the data to produce an output for the given input data.

Another remark is that this was theoretical mathematical model. This means that from the perspective of compute engineering, Turing could build a machine that consisted of his tape and the hardware needed to operate with the central processing unit, in this case, the tape core functionality. Therefore, he had to create a programming language that could compile the code into the Turing’s machine code and then execute. Apart from execution of instructions, he could likely to create a memory to store the output and input data which is processed by his machine. Namely, Turing’s machine had the sequential memory stored in the tape itself.

### Programming Language

As discussed before, a programming language is the language used to give orders or instructions to central processing unit (CPU) of a computer to execute. It is the language that is easily understood by the people who are specialized in it (hereinafter programmers or developers). Still, we have to tell the CPU apart from hardware that is used to simplify developer’s work when building software which is the application that is used by people who do not belong to professionals of computer science; they are referred to as users.

As of today, there are a great number of programming languages that suit different duties and used to simplify developing in various fields of CS (computer science). Of the most prominent languages, we can list the following: Assembly, COBOL, FORTRAN, C, C++, C#, Java, PHP, Python, JavaScript, Rust, Ruby, Swift, Golang and many others. The Fortran is famous due to its application to the field of science such as physics, chemistry, mechanics, and so forth. Today, we mostly leverage the capabilities of a powerful programming language known for its versality and simplicity – Python. How is the programming language understood by the machine?  
 CPU cannot directly execute the instructions that it obtained from developers. For this purpose, programmers introduced the compilation of a programming code (instructions that provide specific algorithms that operate over data and easily understood by CPU) into the machine code, binary code. Thus, C code gets translated into structured instructions represented in binary code and then becomes executed by the machine core CPU.

Among programming languages, we can encounter high-level ones which means that their code is not directly translated into the machine code; instead, it gets interpretated by an interpreter which understands the syntax of the given programming language. For instance, Python is an interpreted language which means it does not get compiled into machine code but executed by its interpreter, which reads the source code of a program in Python. When reading about Python on GitHub platform, we can encounter the files with hpp extensions, meaning that a few Python’s parts are also written in C [3]. In fact, “for loops” in python are written in C and are highly optimized implying that the performance is quite impressive.

The Python’s code gets interpreted and translated into bytecode, rather than entire machine code that is understood by CPU of a machine. This process involves compiling python code into bytecode and is handled by CPython’s compiler. After the compilation, bytecode is executed by the interpreter.

### Memory

From a practical perspective, computers should store the data after the program terminates. As such, if the user obtains a result, it can refer to it later and reuse, as an input data for another program. This leads us to a modular programming approach and data is processed in chunks and every part of it is of great essence; hence, the leak of any data is a critical indicator of failure.

For storing data, computers utilize RAM and external data storage:

1. Referred to as RAM, the random-access memory is the key aspect of modern computers, unlike Turing’s machine infinite memory stored on the tape. What a program uses when to store temporary data is called RAM. It is volatile memory meaning that upon the termination of a program, all data is erased unless loaded to the other type of memory.
2. External data storage includes many types of storing hardware such SSD (Solid-State Drive), or HDD (Hard Disc Drive). This is where our OS (Operative System) is stored, and once the machine system cannot get a signal from the storage, it cannot boot up the system and start all services like those in Linux operated by SYSTEMD (System daemon). The difference between SSD and HHD is that the former is actually faster in terms of performance as it does not use moving parts which reduces the latency (the delay of a response delivered to the user who sent a request) and also more durable [4]

The signal is a keyword here, representing 1 or 0 to indicate True or False, respectively. It means that some components of a machine can interact with others by sending signals. For example, a component of machine electric circuit can send a signal to CPU meaning that the completion of a task. Thus, CPU distributes the task among other components of machine to boost performance of the entire computation process. In fact, when we hit the turn on button situated on the top of physical machine element, it sends electrical signals to the hardware and then the system distributes the work done. Every component is responsible for their tasks, and once completed, they send the positive signal to a CPU. If no signal from SSD, the system cannot display data on the screen, because GPU cannot obtain the information to display.

### Hardware

A computer machine can have other hardware units that simplify the work and reduce the load on CPU, enhancing the overall performance. These common components include:

1. GPU – Graphical processing unit that is responsible for the work associated with graphical operations such as rendering images, and then applying filters. In CS, images are in fact a collection of ordered bits which sit at specific locations. The software relying on the GPU work can 1) filter the image; 2) change the image; 3) mutate the properties; and others. This is typically done by operating over binary data.
2. TPU – Tensor processing unit that is specialized for working with data represented as tensors. In physics, tensors are the generalization of the vector notion. Thus, working with tensors quantities means that we operate in the three-dimensional (or multi-dimensional) space. The simplest way of it is the strain tensor that is given by a 3-by-3 matrix and exhibits the strain value at the specific direction (xy, yx, zy, yz, …). Computers can and are used to facilitate the computational process conducted by humans and for this reason, some specific hardware can be created ([Google official information](https://cloud.google.com/tpu/docs/intro-to-tpu)).
3. QPU – Quantum processing unit is a tool used to process quantum physics matter. For example, a quantum computer can solve a Schrodinger’s equation more efficiently when compared to CPU.

## Supercomputers

### Definition

One of the most prominent examples of such an example is Cray-1 [Source: Britannica Supercomputer CRAY-1 as [link](https://www.britannica.com/technology/UNIVAC)]. What may be a supercomputer? Cray wished to create the most efficient and successful scientific supercomputer that could speedup science research and contribute to mankind knowledge, leading to new revolutions and changed in how people live and interact with the world. The word “scientific” is quite critical because the supercomputer is a machine that is able to carry out scientific computations much faster than a normal machine. What makes these supercomputers unique?

A supercomputer is highly optimized computational machine that can significantly contribute the scientific research. In modern setting, software engineers train ML-systems to create and train neural networks and artificial intelligence models.

Изображение выглядит как текст, диаграмма, линия, Параллельный

Автоматически созданное описание

Fig. 1. Basic computer system [Source: CRAY-1 at [link](https://www.ed-thelen.org/comp-hist/CRAY-1-HardRefMan/CRAY-1-HRM.html)]

### Examples

In this section, I would like to demonstrate examples of the modern famous supercomputers and share their details as a table.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Created At | Details | Further reading |
| Summit  [IBM Power System](https://en.wikipedia.org/wiki/IBM_Power_Systems) AC922 | Year: 2018,  Location: [Oak Ridge National Laboratory](https://en.wikipedia.org/wiki/Oak_Ridge_National_Laboratory),  Where: USA. | OS: Linux  (RHEL 7.4),  CPU: 202,752  (9,216 × 22-core IBM [POWER9](https://en.wikipedia.org/wiki/POWER9) @3.07 GHz),  GPU: 27,648 × 80 Nvidia [Tesla V100](https://en.wikipedia.org/wiki/Nvidia_Tesla) | [The summit](https://en.wikipedia.org/wiki/Summit_(supercomputer))  [supercomputer](https://en.wikipedia.org/wiki/Summit_(supercomputer)) |
| Frontier  [HPE Cray EX235a](https://en.wikipedia.org/wiki/HPE_Cray_EX235a) | Year: 2022,  Location: [Oak Ridge National Laboratory](https://en.wikipedia.org/wiki/Oak_Ridge_National_Laboratory),  Where: USA. | OS: Linux ([HPE Cray OS](https://en.wikipedia.org/wiki/Cray_Linux_Environment)),  CPU: 591,872  (9,248 × 64-core [Optimized 3rd Generation EPYC 64C](https://en.wikipedia.org/wiki/Optimized_3rd_Generation_EPYC_64C) @2.0 GHz),  GPU: 36,992 × 220 AMD [Instinct MI250X](https://en.wikipedia.org/wiki/AMD_Instinct). | [The Frontier supercomputer](https://en.wikipedia.org/wiki/Frontier_(supercomputer)) |
| Fugaku | Year: 2020,  Location: [RIKEN Center for Computational Science](https://en.wikipedia.org/wiki/Riken)  Where: Japan. | OS: Linux ([RHEL](https://en.wikipedia.org/wiki/Red_Hat_Enterprise_Linux)),  CPU: 7,630,848  (158,976 × 48-core [Fujitsu A64FX](https://en.wikipedia.org/wiki/Fujitsu_A64FX) @2.2 GHz),  GPU: None. | [Fugaku supercomputer](https://en.wikipedia.org/wiki/Fugaku_(supercomputer)) |
| LUMI  [HPE Cray EX235a](https://en.wikipedia.org/wiki/HPE_Cray_EX235a) | Year: 2020,  Location: [EuroHPC JU](https://en.wikipedia.org/wiki/EuroHPC_JU) [Kajaani](https://en.wikipedia.org/wiki/Kajaani)  Where: Finland. | OS: Linux ([HPE Cray OS](https://en.wikipedia.org/wiki/Cray_Linux_Environment)),  CPU: 150,528  (2,352 × 64-core [Optimized 3rd Generation EPYC 64C](https://en.wikipedia.org/wiki/Optimized_3rd_Generation_EPYC_64C) @2.0 GHz),  GPU: 9,408 × 220 AMD [Instinct MI250X](https://en.wikipedia.org/wiki/AMD_Instinct). | [LUMI](https://en.wikipedia.org/wiki/LUMI_(supercomputer))  [supercomputer](https://en.wikipedia.org/wiki/LUMI_(supercomputer)) |

Table 2: Supercomputers

In the ‘further reading’ column, I cite my sources of information to build my academic paper on.

## CPU

This is an integral component of most computers. The computer OS (Operating System) can vary from architecture to another, but CPU remains the core. As demonstrated from electric circuits which abide by physics laws such as Maxwell, fundamental law of all electro-magnetism, and Kirchoff’s laws which help to compute the voltage of an integrated computer circuit or microchips. Hence, understanding Maxwell’s equations are essential to gain insight into how microchip’s components interact with each other, while sending electrical signals of Boolean nature: False or True.

## OS

Also, referred to as Operating System, OS was designed to introduce multitasking and facilitate the computer’s CPU workload by properly managing its resources. OS is the integral component of computer systems because it introduced user-friendly interface and assisted in improving computer’s performance. Despite computer machine being to process data without any OS, the operating system is of great importance and significantly transformed our interaction with computing machines.

Throughout the history, there have been many instances of computer machines and the most popular ones include Unix and MS-DOS. Both OSs were written in C, the programming language, because C enabled the portability across various CPU architectures which allowed many engineers to execute their code efficiently.

## Compilation process

A compiler carries out critical duty of computer science by allowing developers to run their code and identify bugs and syntax errors, as well as to test it against bugs and errors. The compilation process is intricate and involves several minor tasks (subtasks) to fulfill. What these tasks might be? As an introducing problem, let us consider the human readable sentence:

This is a sentence.

This is a classic example studied in compilation process analysis. When people read it, they automatically, withing milliseconds, understand its meaning. They do so by breaking the sentence into meaningful parts, called, tokens, and the process is referred to as tokenization, as shown below in the picture. We can visualize this as a tree of sentence’s nodes: subject, predicative, article, and object – these are the sentence’s parts. By following Grammar standards and rules, we can ensure our meaning conveyed by a sentence is delivered and understood, thereby creating, and organizing the proper communication. However, there are some problems in English which prevent readers from understanding a sentence correctly: the incorrect usage of pronouns leads to loss of understanding:

Jack said his friend bought a book.

This sentence is correct, and we can understand it well. Consider another example:

Jack said Peter bought his book.

The problem that you start discerning is the meaning of the pronoun “his”, which can refer to various nouns used in the sentences. In this example, “his” refers to “Jack”. Let us analyze a few more examples:

Jack said Jack bought a book.

Jack said that his book was sold to Jack.

Jack bought Jack bought and sold it to Jack.

Jack remembered that Jack sold her book.

You can see that the scope of the nouns is the way in which computer compilers address the problem (the following code is written in Python):

Изображение выглядит как рукописный текст, текст, зарисовка, рисунок

Автоматически созданное описание

Fig.2: The tree of a sentence: tokenization

number = 1

def print():

number = 2

print(number)

return None

print(number)

When we run the script, we can see that even though the name is similar, the variable addressed are different, and, hence, the values stored in them would differ. This is how Python handles the ambiguity problem.

### Lexical Scope

To demonstrate how scoping is used in lexical analysis, we can consider a few examples. Let us start with the simplest one: inner and outer functions with different variables. In the outer function, we declare a variable with name ‘num’, and 1 as value assigned to it. In the inner function, we also declare a variable with similar name but different value: 2. In both functions, we print the variable to see its value in the console, and the inner function returns its variable, while the outer function returns the inner function to ensure “closing”. Thus, we may write the following code:

def foo():

num = 1

print(num)

def baz():

num = 2

print(num)

return num

return baz

#Let us use it – Remember, it returns a function

test = foo() # not a varible

#Let us invoke the function

result = test()

#Let us print the result

print(result)

The output of this program is the following:

1

2

2

It is because when we invoke the function ‘foo’, we see its scoped variable printed. When we invoke the inner function, we also see its variable value printed to the console, and then we can store the value of it in “result” variable. In Python, we can use ‘nonlocal’ keyword to access outer function variables:

def foo():

num = 1

def baz():

nonlocal num

print(num)

return num

return baz

test = foo()

result = test()

The output will be

1

1

This is because now the inner variable accesses the outer function’s lexical scope and can use its variables. Let us consider another example:

#Organize data

english, estonian, russian = TextFormatter.process\_descriptions([

language.get("Description", "No Description")

for language in [

data.get(language,"No such language") for language in data.keys()

]

])

Have you seen the ‘language’ name used twice? Why is it possible? Because the ‘language’ variables are used in different scopes, which is why Python can tell them apart and my code runs successfully. However, I would like to specify the data used in the script: I have a large, nested dictionary which represents the tree storing my data; in this particular case, I store and process all necessary information related to my CV: languages, skills, and so forth. Thus, compilers introduced.

### Compilation subtasks

When our code C++ code is compiled, it is translated into Assembly code for the specific CPU architecture. After this, ASM code is compiled into machine code that is understood by the target CPU. Once our code is compiled, we can run it as .com or .exe files or just as an executable file on Linux machine – for this, we have to create a link to it by using ld command. For example,

nasm -f elf64 -o program.o program.asm

ld -o program program.o

./program

Now we can run the program and get “Hello, world!” as an output.

The compilation process involves the following subtasks:

1. Lexical analysis: This is the first part when a compiler analyzes the source code and groups its part to create meaningful sequences, identifying tokens.
2. Syntactic analysis: Check for syntax, and other syntax-related errors. It constructs a hierarchical structure known as a parse tree with the help of tokens.
3. Semantic analysis: This phase ensures that the components of the program fit together meaningfully. It gathers type information and checks for type compatibility.
4. Intermediate code generation: After semantic analysis, the compiler generates an intermediate code of the source code for further processing.
5. Optimization: This phase optimizes the intermediate code for the better performance by removing unnecessary lines of code and arranging the sequence of statements.
6. Code generation: This is the final phase when the optimized intermediate code is converted into the machine code.

# Assembly

## Introduction

The Assembly is one of the lowest-level programming languages. It is the language used to program and compose hardware for a computer system, which means that to create a coffee machine and design peripherals to use is to leverage the power of Assembly. Additionally, numerous software applications have been developed in this language, ranging from hardware drivers to popular games like Rollercoaster. This is impressive considering that Assembly is one of the most challenging programming languages to master. Why would one have to learn and use the programming language? Why is it important to look at Assembly and invest one’s time into the language?

The first reason is connected to Assembly’s low-level nature. Assembly teaches how to maintain memory allocation, memory management and pointers arithmetic. Hence, Assembly can be used to further optimize some program to enhance their performance, because Assembly is a low-level programming language, closely tied to a computer’s CPU. This argument aligns with the understanding that programming at a low level, such as Assembly, allows for more precise control over hardware resources, which can lead to more efficient and optimized code.

The other reason is that Assembly can extend one’s outlook and assist in designing business ideas such as developing hardware for coffee machines or Tesla cars, for example. Learning Assembly equips individuals with the skills needed to work at the intersection of software and hardware, helping people to bring innovative, bright ideas to live in the technology industry.

Lastly, Assembly can shed light on how arithmetic operations are performed over binary numbers. This knowledge help teacher and educational centers to teach binary numbers to any individual with interest in information technology field.

## Registers

In Assembly, registers are used to store some data in specific memory location that are accessed by CPU when executing the code. For example, the registers RAX, EAX, AX, AL are used to perform arithmetic, while RBX is used for addressing data in memory, RCX is used as counter register, and RDX is used for data operations and often is paired with RAX. The interesting diagram is placed below.

Изображение выглядит как текст, рукописный текст, зарисовка, диаграмма

Автоматически созданное описание

Fig. 3: Register’s RAX hierarchy.

## Hello World Program

In this section, I teach how to write HelloWorld program in Assembly. It may seem arduous at first glance, but with practice, one can get more familiar and gain more insights into Assembly. Let us write some code.

section .data

text db “Hello, world!”, 0ah

section .text

global start

start:

mov rax, 1

mov rdi, 1

mov rsi, text

mov rdx, 14

syscall

mov rax, 60

mov rdi, 0

syscall

Upon the first observation, it is difficult to understand what happening. Therefore, it is important to attempt at mutating the variable ‘text’ value and run the program again. How to run our program? There is a couple of ways how to achieve the goal: 1) one may search for web compilers, specifically, NASM Linux x86, or it can install nasm and run the commands that we discussed earlier:

nasm -f elf64 -o helloworld.o helloworld.asm

ld -o helloworld helloworld.o

If no error has occurred, we just create an executable file with the name helloworld and we can run it using the following command:

./helloworld

As a helpful tip, on a Linux machine, it is possible to reuse previous efficiently by using ‘!!’ or ‘!<the first letter of previous command>’ to recall and execute them. For example,

ls

!l

Here, the ‘ls’ command is actually list of items in the current directory, and one can read more about it by running ‘man ls’ command.

Let us return back to the Assembly hello world program. The key idea lies in the line ‘mov rax, 1’. By doing so, you efficiently move 1 to the rax register which associates it with the ID equal to 1. If one looks up the table of IDs to call system procedures, it can see that we are calling the write procedure. Another command, “mov rax, 0’ would indicate the ‘read’ procedure and Assembly can scan our data. It is done by including ‘.bss’ section where we declare ‘data resb size’ as in ‘number resb 8’, 8 bits is the size of our number.

The second line ‘mov rdi, 1’ is also of great importance. It actually places the value 1 to the register rdi, which holds the ID of our system procedure and specifies the standard output. The next command ‘mov rsi, text’ is quite essential, because it stores the address of our text in the ‘rsi’ register. The size of our string is stored in the ‘rdx’ register. The last piece of puzzle is to call the procedure, and since we are going to invoke the system procedure with ID equal 1, we have to call the system function: ‘syscall’ is the correct command for that.

The last segment of code is designed for exiting our program. The symbol ‘0ah’ is equivalent to ‘10’ and this is ‘\n’, or new line character. One can compare the program with Python’s one: ‘print(“Hello, world!”)’.

Remark. ‘db’ stands for ‘define bytes’ and it is used to indicate raw bytes data.

## Printing Strings

To print any string of any size, you want to loop over the string until null character is reached. Let us start writing code:

section .data

text db ‘Hello, world!’, 0ah ; this is our string to print.

section .text

global \_start

;This is a macro.

%macro exit 0

mov rax, 60

mov rdi, 0

syscall

%endmacro

\_start:

mov rax, text ; store text memory address in the rax register

call \_printText ; call the subprocedure

\_printText:

push rax ; push data onto stack to save the current memory address

mov rbx, 0 ; setup our counter

printTextLoop:

inc rax ; moves rax to the next memory location

inc rbx ; increment our counter

mov cl, [rax] ; cl is used to store byte-level information

cmp cl, 0 ; compare if current character is null one (0ah, 10)

jne printTextLoop ; if not equal to 0, then proceed to another iteration

mov rax, 1 ; this gets executes when cl = 0

mov rdi, 1 ; file descriptor 1 to set up standard output

pop rsi ; pop the value and store in the given register

mov rdx, rbx ; set the size of the output data to that stored in the rbx register

syscall ; call the system procedure with ID 1, meaning ‘write’

ret ; return

In the code, we use various techniques such as using macros, loops and registers to operate over data. For example, we store the data in cl register, which is used to work with byte-level operations.

## Macros

In the Assembly program, one can encounter the expressions containing ‘%’ sign and followed by the name ‘macro’. For example,

%macro printText 1 2 ; macros take as parameters 1 and 2

mov rax, 1

mov rdi, 1

mov rsi, %1

mov rdx, %2

syscall

%endmacro

This macro can be used in our main routing called ‘start’:

%macro exit 0 ; macros take no parameters

mov rax, 60

mov rdi, 1

syscall

%endmacro

start:

printText msg, 14

exit

Also, we can import other files in our Assembly program to make our code more modular. This can simplify and make our code more readable and cleaner as well as efficient:

%include ‘module.asm’

## Print Numbers

Let us print some numbers in the Assembly. The first thing to know is that Assembly utilizes UNICODE characters or ASCII forms. To demonstrate this, we can return back to hello world program:

section .data

msg db ‘Hello, world!’, 0ah

Let us change the ‘Hello, world’ to, say, 49, and run the program. Once our program is compiled and made executable, we can run it and see the output, which is the character stored with 49 UNICODE:

1

Thus, the program’s output is ‘1’. Therefore, we can write the following program to display numbers from 0 to 9:

section .data

num db 5, 0ah

section .text

global \_start

\_start:

mov rax, num ; load the address of our chosen number into num

call printNumber ; call our subroutine

mov rax, 60 ; Terminate the program

mov rdi, 0

syscall

printNumber:

mov rax, 48 ; UNICODE for numbers is from 48

add [num], al ; holds the least significant bit of num

mov rax, 1 ; trivial printing

mov rdi, 1 ; file descriptor to 1, meaning STDOUT

mov rsi, num ; the address of num is loaded into the rsi register

mov rdx, 1 ; specifies the size of our digit

syscall ; call the system procedure

ret ; return

When we compile this program, the output will be the number stored in .data section.

Another more complex program is to print the number like ‘123’ to the console. This involves division, as one has to traverse the number from left to right and store each digit in stack, which helps print the character one by one until we reached the end. To illustrate this, I write the following program:

section .bss

digitSpace resb 100

digitSpacePosition resb 8

section .text

global \_start

\_start:

mov rax, 123 ; load our number into the rax register

call printNumber ; call the subroutine

mov rax, 60 ; terminate the programm

mov rdi, 0

syscall

printNumber:

mov rcx, digitSpace

mov rbx, 10

mov [rcx], rbx

inc rcx

mov [digitSpacePosition], rcx

printNumberLoop:

mov rdx, 0

mov rbx, 10

div rbx

push rax

add rdx, 48 ; The ‘rdx’ register is added to 48, which is ASCII for numbers

mov rcx, [digitSpacePosition]

add [rcx], dl

inc rcx

mov [digitSpacePosition], rcx

pop rax

cmp rax, 0

jne printNumberLoop

printNumberSecondLoop:

mov rcx, [digitSpacePosition]

mov rax, 1

mov rdi,1

mov rsi, rcx

mov rdx, 1

syscall

mov rcx, [digitSpacePosition]

dec rcx

mov [digitSpacePosition], rcx

cmp rcx, digitSpace

jge printNumberSecondLoop

ret

The idea behind this program is quite simple: The remainder of division is stored in the rdx register:

mov rax, 10

mov rbx, 2

div rbx

In this case, we can see that ‘rdx’ should be 0, as the remainder of 10 / 2 is 0, and the quotient is 5, because 10 / 2 is 5.

To add up two numbers, we can simply using “add” procedure and perform the addition:

mov rax, 123 ; the first number

add rax, 1 ; the second number

## Arithmetic operations

To perform arithmetic operations over simple numbers such as 5 or 9, we can write the following program:

section .data

num db 0, 0ah

section .text

global \_start

%macro sum 2

; macro consists of several functions.

; Constrait: digits from 0 to 9.

; Args: 1 and 2.

; Returns: the sum of these arguments.

mov rax, %1

add rax, %2

printDigit rax

%endmacro

%macro subtract 2

; macro consists of several functions.

; Constrait: digits from 0 to 9

; Args: 1 and 2.

; Returns: The result of subtraction: 2 from 1: 1 - 2.

mov rax, %1

sub rax, %2

printDigit rax

%endmacro

%macro multiply 2

; macro consists of several functions.

; Constrait: digits from 0 to 9

; Args: 1 and 2.

; Returns: The product of 1 and 2

mov rax, %1

mov rbx, %2

mul rbx

printDigit rax

%endmacro

%macro divide 2

; macro consists of several functions.

; Constrait: digits from 0 to 9

; Args: 1 and 2.

; Returns: The division of 1 by 2

mov rax, %1

mov rbx, %2

xor rdx, rdx ; ensure that rdx is zero

div rbx

printDigit rax

%endmacro

%macro printDigit 1

; Prints from 0 to 9.

; Input: the parameter 1.

; Prints: Number passed to it.

mov rax, %1

mov [num], al

call \_printNumber

%endmacro

%macro exit 0

mov rax, 60

mov rdi, 0

syscall

%endmacro

\_start:

printDigit 1

sum 2, 3

subtract 6, 2

multiply 2, 4

divide 6, 3

exit

\_printNumber:

mov rax, 48

add [num], al

mov rax, 1

mov rdi, 1

mov rsi, num

mov rdx, 2

syscall

ret

# Binary Operations

## Basics

Let us try to implement binary addition in Python. To add up two numbers, we usually utilize the arithmetic operator ‘+’: ‘2+2’ produces 4 as an output. To add up two binary number, we use the bitwise operations such as XOR, AND, OR, and SHIFT. Let us perform these operations.

The first one is XOR and it is called bitwise logical OR:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Result | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

Table 3: XOR between 1 and 2

The 4-bit representations of these numbers is quite known: ‘0b0001’ for 1 and ‘0b0010’ for 2. The result in this example is ‘0b0011’ or ‘0000 0011’ which corresponds to 3. Let try another example.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| Result | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

Table 4: XOR between 5 and 7.

The result of this 5 XOR 7 is ‘0000 0010’ which corresponds to 2 in the decimal numeral system. In Python, we can quickly check the result: ‘5 ^ 7’ which is 2. As it can be seen from the tables, the XOR is the bitwise operation which process our number bit by bit and compares bits of binary numbers. It returns 1 whenever one’s number bit is 0 and the other number’s bit is 1, or, on the contrary, if the first number’s bit is 1, while the second one is 0, consequently, it returns 1; if both bits are 0, then it returns 0, and if both are 1, it also returns 0.

Another bitwise operation is called AND. Let us perform the operation between the same numbers:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Result | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 5: AND between 1 and 2.

Thus, 1 AND 2 results into 0 or ‘0000 0000’, which is also ‘0b0000’ in the 4-bit representation. Let us consider another example: 5 AND 7:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| Result | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

Table 6: AND between 5 and 7.

The result of this operation is the binary number ‘0000 0101’ which corresponds to 5. In Python, you can use ‘&’ operator to perform the operation: print(5 & 7) produces 5 as an output.

The third operation is the bitwise OR operator. For example,

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Result | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

Table 7: bitwise OR between 1 and 2.

The result is 3, as expected. Let us try the second example:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| Result | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

Table 8: Bitwise OR between 5 and 7.

As seen from the table, the result is ‘0000 0111’ or ‘0b0111’ which corresponds to 7, whereas the XOR result was 5.

Another example would be SHIFT. It is also bitwise operator but might be differentiated into 3 types: arithmetic shift, logical shift, and traditional shift. For example,

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Result | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

Table 9: SHIFT LEFTWARDS BY 1 BIT.

The result of the left shift by 1 bit is equivalent to multiplication the number by 2. In this example, the output is 2 for the input number 1. However, we can shift by any bit number:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Result | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Table 10: SHIFT LEFTWARDS BY 2 BITS.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Result | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

Table 11: SHIFT LEFTWARDS BY 5 BITS.

To represent this number in python, we can use ‘b’ notation for ‘binary’: ‘0b0010\_0000’ in 8-bit representation. As it can be demonstrated, the maximum number for 4-bit representation is ‘0b1111’ which corresponds to 15, or ‘0xF’ in the hexadecimal numeral system. The maximum for 8-bit representation is ‘0b11111111’ which corresponds to 255.

We can also shift RIGHTWARDS:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
| 11 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| Result | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

Table 12: SHIFT RIGHTWARDS BY 1 BIT.

The result of this right shift by 1 bit is ‘0b00000101’ which corresponds to 5. In Python, you can perform these operations by using ‘>>’ for the right shift and ‘<<’ for the left shift. For example, for the given example, we could ’11 >> 1’ which produces 5 as an output, or ‘1 << 5’ which yields 32. In other programming languages such as JavaScript, we can encounter logical shift operator ‘>>>’. Consider, the example:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| Result | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

Table 13: LOGICAL SHIFT RIGHTWARDS BY 1 BIT.

Another example,

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
| 12 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| Result | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

Table 14: LOGICAL SHIFT RIGHTWARDS BY 1 BIT.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| Result | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

Table 15: LOGICAL SHIFT RIGHTWARDS BY 1 BIT.

The difference is when we try to perform this operation on the unsigned binary number. It will shift by the specified number of bits and fill those with zero. For example,

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | 32-bit representation | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| r | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

Table 16: The logical right shift by 2 bits.

The input number is ‘-5’ which is ‘1111 1111 1111 1111 1111 1111 1111 1011’. We shift it rightwards by 2 bits and fill left-side with zeros: ‘0011 1111 1111 1111 1111 1111 1111 1101’, which corresponds to ‘1073741822’.

## Negative binary numbers

As seen from the previous examples, “~number + 1” is actually “-number” or two-complementary binary number, and the process is called complementing.

Example 1. Find the binary representation of ‘-13’. First, we have to find the binary representation for 13 for 4-bit form:

Thus, to find the binary for 13, we have to divide it by 2:

Thus, the quotient is 6 and remainder is 1 (because ). Let us continue dividing by 2:

The quotient is 3, while the remainder is 0. The next division is whence the quotient is 1 and the remainder is 1 (because ). The last division leads to the zero quotient and the remainder being equal to 1. Thus, we can form a structure based on the remainders:

in reverse order. Thus, our number is ‘0b0000\_1101’.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
| 13 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| Result | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

Table 17: Logical Not.

From the table, we can see the result of ‘~0b0000\_1101’ is ‘0b1111\_0010’. The last step is to add a binary ‘1’ to it:

We can perform the addition according to a table:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
| Addend 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| Addend 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Sum | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |

Table 18: Adding two numbers to get ‘-13.’

Thus, the sum is the binary number ‘0b1111\_0011’, which corresponds to 243 as positive, and ‘-13’ as negative. To verify our calculations, we can use Python here:

num = 0b0000\_1101

inversed = ~num

negative\_num = inversed + 1

print(negative\_num)

Also, you can see our result by running the script:

num = bin(-13 & 0xFF)

print(num)

In this case, we get ‘0b1111\_0011’ as an output. Otherwise, we get ‘-0b1101’ to denote ‘-13’.

## Implementation in Python

### Iterative approach

To add up, two numbers, we can use XOR if bits do not coincide:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
| Addend 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| Addend 2 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| Sum | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Table 19: Adding two numbers to demonstrate how XOR is used.

Thus, ‘0b0010\_1111’ + ‘0b1101\_0000’ is equal to ‘0b1111\_1111’ or . Can we add up any binary numbers using this approach? It turns out that we cannot. To illustrate that, we can consider the addition:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
| Addend a | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| Addend b | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Sum | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Table 20: Adding two numbers using XOR leads to incorrect answer.

Thus, , or , but the sum would be 5, as opposed to the given result. Thus, we have to seek another approach as to how add up two binary numbers. For this purpose, we have to look at the expression , which is crucial here, because it can help to determine the sum.

Remark. Please, do not get confused, because the least significant bit is found by.

.

For example, , and , as seen from the table.

The result ‘3 AND 2’ in this context is denoted by ‘carryover’. Let us apply the left shift to the given ‘carry’:

Now, let us perform another round of these operations:

Now, we take closer look at the expression stored in : . We recognize this as the binary number corresponding to 5, which is our sum. Thus, we can write the following algorithm:

def add(a: int, b: int) -> int:

“””

This function adds up two binary numbers using iterative approach. It does so by efficiently applying XOR, AND and LEFT SHIFT.

“””

while b != 0:

carry = a & b

a ^= b

b = carry << 1

return a

#Example usage

print(add(2, 3))

This algorithm works only for nonnegative numbers, therefore, for and .

To perform subtraction, we have to notice that carry changes to . Let us again perform all steps manually. This time, I show calculations as tables: where and .

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
| Minuend  ~a | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| Subtrahend b | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| Carry | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |

Table 21: The first iteration: performing logical AND and store result in carry.

Thus, ; hence,

I also perform these bitwise operations using tables:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
|  | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
|  | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |

Table 22: The first iteration: performing logical XOR.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
|  | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |

Table 23: The first iteration: performing logical LEFT SHIFT.

So, we have the following data:

Let us perform necessary computations for the next iteration:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|  | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

Table 24: The second iteration: performing logical NOT: bitwise inversion.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
|  | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| Carry | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

Table 25: The second iteration: performing logical AND.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
|  | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| Subtraction | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |

Table 26: The second iteration: performing logical XOR.

Thus,

As we can see, , so we must continue the cycle:

Again, we use tables.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
|  | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |

Table 27: The third iteration: performing logical NOT: bitwise inversion.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Carry | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 28: The third iteration: performing logical AND.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
|  | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Subtraction | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |

Table 29: The third iteration: performing logical XOR.

Upon thorough observation,

Thus, the result of is , which is correct: or

Our example is completed.

Let us write the function:

def subtraction(a: int, b: int) -> int:

while b != 0:

carry = ~a & b

a ^= b

b = carry << 1

return a

#Example usage

print(subtraction(218, 55))

As an output, we obtain the number 163 as expected.

Now we can try to call our function with , so :

print(subtraction(2, 3))

This leads to infinite loop; we can work around this very easy:

1. If , then , so and our function works as expected.
2. If and , then and , so while .

Therefore,

#abstraction layer

def subtract(a: int, b: int) -> int:

if 0 < b < a:

return sub(a, b)

elif 0< a < b:

return -sub(b, a)

elif a > 0 and b < 0:

return add(a, b)

elif b > 0 and a < 0:

return -add(-a, b)

else:

return -add(-a, -b)

The functions ‘sub’ and ‘add’ are listed above.

### 2-complement binaries

Here, I would like to share another approach for negative numbers:

def add(a: int, b: int) -> int:

mask = 0xFFFFFFFF

MIN = -0x80000000

MAX = 0x100000000

while b != 0:

carry = (a & b) & mask

a = (a ^ b) & mask

b = carry << 1

if a <= MAX:

return a | MIN

return ~(a ^ mask)

This function works for negative numbers: , while ; or , but , and . The output will be the negative number.

### Recursion

Let us first rewrite our iterative function into recursion:

from functools import cache

@cache

def add(a: int, b: int) -> int:

if b == 0:

return a

return add(a ^ b, (a & b) << 1).

@cache

def subtract(a: int, b: int) -> int:

if b == 0:

return a

return add (a ^ b, ((~a) & b) << 1)

These functions work for the numbers which are not floating, nor negative. They are decorated with the decorator ‘cache’ to ensure excellent performance by leveraging the power of memorization. Following the same idea, we can use these functions to create one general function:

def add\_numbers(a, b):

if a >= 0 and b >= 0:

# Both positive numbers, use regular addition

return add(a, b)

elif a < 0 and b < 0:

# Both negative numbers, use regular addition and negate the result

return -add(-a, -b)

elif a >= 0 and b < 0:

# a is positive, b is negative, subtract b from a

return subtract(a, -b)

else:

# a is negative, b is positive, subtract a from b

return -subtract(-a, b)

# Example usage:

print(add\_numbers(5, 3)) # Output: 8

print(add\_numbers(-5, -3)) # Output: -8

print(add\_numbers(5, -3)) # Output: 2

print(add\_numbers(-5, 3)) # Output: -2

## Binary vector

Bitwise operations can be well applied to the problem of searching for duplicates in the array. First, we consider the following fundamental bitwise functions:

def is\_set\_bit(bit\_vector: list[int], index: int) -> bool:

row = get\_row(index)

column = get\_column(index)

return (bit\_vector[row] & (1 << column)) != 0

def set\_bit(bit\_vector: list[int], index: int) -> None:

row = get\_row(index)

column = get\_column(index)

bit\_vector[row] |= 1 << column

def get\_row(bit: int) -> int:

return int(bit >> 5)

def get\_column(bit: int) -> int:

return bit % 32

#Usage

nums = [1,2,3,3, 3, 4,5, 5,6,7,8,9]

visited = [0] \* 256

def print\_duplicates(nums: list[int]) -> None:

for i in range(len(nums)):

if is\_set\_bit(visited, nums[i] – 1):

print(nums[i])

else:

set\_bit(visited, nums[i] – 1)

print\_duplicates(nums)

## Assembly

In Assembly, the procedure sub, add are already operating with binaries. However, in order to store binaries, we would have to store them as hexadecimal numbers:

section .data

num db 0x1

section .text

\_global start

\_start:

mov rax, 48 ; translating UNICODE characters

add [num], al

mov rax, 1

mov rdi, 1

mov rsi, num

mov rdx, 1

syscall

mov rax, 60

mov rdi, 0

syscall

Thus, in Assembly, all those bitwise operations are already implemented. Therefore, we do not have to use loops for this purpose. Instead, we can use add and sub subroutines. For example,

section .data

num db 0x2

result db 0x0

section .text

global \_start

\_start:

mov rax, [num]

add rax, 5

mov [sum], rax

mov rax, 48

add [sum], al

mov rax, 1

mov rdi, 1

mov rsi, sum

mov rdx, 1

syscall

mov rax, 60

mov rdi, 0

syscall

This program uses binary addition.

# 32-bit floating-point format

In order to convert a real number into the given format, we have to, first, identify if the number is positive or negative; then, we normalize the binary representation to get fractional part; after, we count how many shifts we have to perform to the right in order to achieve the desired form (nonzero number). The last step is to determine the binary representation of a fractional part.

In Python, we can achieve this using the following script:

def float\_to\_bin(number):

return format(struct.unpack('!I', struct.pack('!f', number))[0], '032b')

def bin\_to\_float(binary):

return struct.unpack('!f',struct.pack('!I', int(binary, 2)))[0]

num = 123.456

binary = float\_to\_bin(num)

print(f"Binary representation of {num} is {binary}")

print(f"Float representation of {binary} is {bin\_to\_float(binary)}"

# 

# The implementation of XOR

## Introduction

As known from previous chapters, XOR is a bitwise operation that performs the following actions:

1. If and , then ;
2. If and , then ;
3. If and , then ;
4. If and , then .

The XOR operation can be expressed in terms of AND and OR bitwise one:

Thus, XOR is implemented using AND, OR, and NOT bitwise operations; this is actually the logic of how we apply XOR. Let us substitute and ; the expression evaluates to , which is equal to ; hence, . Now we can notice that if one of these numbers and happens to be 1, then returns 1, unless the other number is equal to 1 as well. If both are zero, then the result is zero. This is what actually described by the formula.

It is interesting to point out that XOR is reversible, meaning that knowing and the result of , we can find : , or . Let us demonstrate this property of XOR binary operation:

def determine\_left(b: int, result: int) -> int:

return b ^ result

def determine\_right(a: int, result: int) -> int:

return a ^ result

def XOR(a: int, b: int) -> int:

return a ^ b

Here, “^” denotes the bitwise operation XOR: , where “~” is the inversion bitwise operator; “&” is the bitwise AND, and “|” is the bitwise OR.

Suppose that and , then, the result is or 3 in decimal. We can indeed find using and – this is what happens in the “determine\_left” function. On the other hand, we indeed determine based , and , using the second function called “determine\_right”.

For demonstration purposes, we can determine , using the definition of . Let us determine the inversed values: , and . Let us evaluate the left and right expressions: , and . Finally, , which is the expected result. The reader can find out that

which is the modified expression of using algebraic transformations based on axioms. Let us see that the result remains the same:

Thus, the result remained unchanged and equal to , which is because both definitions are equivalent, based on the following axioms [4]:

1. Complement law: ;
2. Distributive law: and ;
3. Associative law: ; and ;
4. Definition of XOR as stated above: ;
5. The De Morgan’s laws: , and .
6. Cumulativeness: In Boolean algebra, , , and .

## Applications of XOR

The bitwise XOR is widely used in cryptography, for example, in the implementation of RSA algorithm, or SHA-256, or AES. More specifically, in RSA, to proceed to next iteration of 16 total ones, we have to determine the key of the forthcoming iteration. It is done by applying XOR to a matrix of hexadecimal values column by column. As a result, we get another matrix, key, which is the key of the next iteration and we proceed to it.

Another interesting application is pseudo-random generator. It is pseudo-random due to complexities of random distributions in real life as results of physical phenomena such as radioactive decays or entropy when observing the movement of a gas’ particles abided by Maxwell’s distribution law.

### Linear congruential distribution

The algorithm is based on mathematical formula . Therefore, we can implement this in Python as follows:

from enum import Enum

class Params(Enum):

A=1103515245

C=12345

M=32768

x = 1

def random():

global x

x = x \* Params.A.value + Params.C.value

return x % Params.M.value

Here we use the module ‘enum’ to extend ‘Enum’ class and define constants called ‘Params’. Based on this params, we mutate the ‘random’ function scope variable value and return the remainder of the division by the parameter M:

Consider simple example:

#Define a list

#Remember list is preserverd keyword to call a constructor:

#list([1,2,3,4,5]) or list({“a”:1, “b”:2}.values())

lst = [1,2,3,4,5]

length = len(lst)

first = lst[0 % length]

second = lst[1 % length]

another\_first = lst[5 % length]

another\_second = lst[6 % length]

As it can be discerned, we got the same values. Likewise, our linear congruential generator can produce the same output after the define number of iterations.

### XOR Shift generator for 32-bit

We can implement another idea on how to generate pseudo-random numbers:

class State:

def \_\_init\_\_(self, a) -> None:

self.a = a

def xor32shift(state):

#Get the state

x = state.a

#Perform XOR 32-bit shift:

x ^= x << 13

x ^= x >> 17

x ^= x << 5

#Assign the result to state.a:

state.a = x

return state.a

def xor64shift(state):

state = state.a

state ^= state << 12

state ^= state >> 25

state ^= state << 27

state.a = state

return state.a

def main():

state = State(123456789)

print(“XOR 32”)

for \_ in range(10):  
 print(xor32shift(state))

print(“XOR 64”)

for \_ in range(10):  
 print(xor64shift(state))

if \_\_name\_\_ == “\_\_main\_\_”:

main()

## The implementation of XOR

### Python

The first way of implementing the XOR function in Python is to use either definition of it:

def XOR(a: int, b: int) -> int:

return (~a & b) | (a & ~b)

def XOR(a: int, b: int) -> int:

return (a | b) & (~a | ~b)

Or we can loop over the bits of each number and then compare them according to definition. To do so, we implement the following function:

def XOR(a: int, b: int) -> int:

#Convert input numbers into binary strings

bin\_a = bin(a)[2:]

bin\_b = bin(b)[2:]

#Initialize the result that is going to store our ‘answer’

result = “”

#Determine the max length to fill with zeroes

max\_length = max(len(bin\_a), len(bin\_b))

bin\_a = bin\_a.zfill(max\_length)

bin\_b = bin\_b.zfill(max\_length)

#Loop over these binary numbers in the reversed order

for x, y in zip(bin\_a[::-1], bin\_b[::-1]):

if x != y:

result += “1” #XOR logic

else:

result += “0”

#Reverse a string

result = result[::-1]

#Conver back to a decimal (int type), and radix is 2 (binary)

num = int(result, 2)

return num

### Assembly

In Assembly, we already have built-in XOR function. However, following the same idea, we can utilize the fact that XOR is defined in terms of AND, OR, and NOT. For example, consider the following code:

section .data

result db 0

number db 0

section .text

global \_start

%macro exit 0

mov rax, 60

mov rdi, 0

syscall

%endmacro

%macro printDigit 1

; This is used to print a number from 0 to 9

mov rax, %1

mov [number], al

call \_printNumber

%endmacro

\_start:

call my\_xor ; call our subroutine to perform XOR between 1 and 2

printDigit rax ; print the result stored in rax

exit ; exit the program (0).

my\_xor:

mov rax, 0x1 ; load the value of 1

mov rbx, 0x2 ; load the value of 2

mov rcx, rax ; copy the value of 1 into rcx

mov rdx, rbx ; copy the value of 2 into rdx

not rax ; invert the value of rax (using NOT)

not rbx ; invert the value of rbx (using NOT)

and rax, rdx ; performing AND (left part)

and rbx, rcx ; performing AND (right part)

or rax, rbx ; performing OR to combine those and get a result

ret ; return the value stored in rax

\_printNumber:

mov rax, 48 ; move the value in rax to 48 (UNICODE of numbers from 48)

add [number], al ; al for 8-bit values

mov rax, 1 ; ID 1 is for writing, 0 for reading

mov rdi, 1 ; File descriptor 1

mov rsi, number ; what to print

mov rdx, 2 ; length of what to print

syscall ; call the system procuder

ret ; return (None)

However, Assembly has the xor routine, so we can simply

mov rax, 1

mov rbx, 2

xor rax, rbx

printDigit rax

We can attempt at looping over the strings stored in data:

section .data

first db “0b0001”

second db “0b0010”

section .bss

result resb 8 ; store 8 bits

# Optimization

## Introduction

This section talks about techniques and concepts related to optimization. This is of great importance when it comes to making one’s product more efficient in terms of space and time complexities. This is because when deploying the product to production, more users can smoothly use the application. This smoothness ensures the flexibility of the application, meaning it is fast and secure, as well as providing good quality. Throughout computer science history, people devised various ideas on how to speed up and make their applications stable. This pursuit of brilliant ideas on how to achieve such a goal made most developers develop critical thinking while testing their product thoroughly.

One of the most crucial aspects is memoization (do not be confused with React’s memoization which happens when React needs to re-render the UI/UX content when properties, shortened to props, changed; otherwise, React does not re-render, which makes the application reactive – React). The purpose of this technique is to speed up one’s application by returning the previously computed data instead of repeating the computational process.

Another concept is cache, but this is not memoization, but a different concept; the reason why is because the cache is a chip placed onto the CPU which can speed up its work.

## Memoization

### Overview of the technique

In order to understand the technique better, we must consider the classic example – a recursive function which computes the factorial:

def factorial(num: int) -> int:

if num <= 1:

return 1

return num \* factorial(num - 1)

This function computes the factorial recursively and can be suitable for evaluating simple factorials such as 5! = 120 or 10! = 3628800. However, what happens if we store the results and once the number is the same, we simply return? The idea leads to the principle of memoization for the factorial function and is implemented as follows:

\_CACHE = {} # initialize an empty dictionary

def factorial(num: int) -> int:

try:

return \_CACHE[num] # if the result is already computed

except KeyError:

# Compute the factorial

if num <= 1: return 1

result = num \* factorial(num - 1) # once the CALLSTACK has the result

# Store it in cache

\_CACHE[num] = result

return \_CACHE[num]

Please, notice that the \_CACHE decorator is placed outside of the function, which prevents Python from creating the same dictionary over and over again; yet, for larger numbers, this function will not work because of RecursionError. As such, the recursive is not suitable for computing factorials for such large numbers as 10000.

### Examples

Another classic example is Fibonacci numbers:

def fib(num: int) -> int:

if num == 1: return 1

if num <= 0: return 0

return fib(num - 2) + fib(num - 1)

print(fib(7))

You can also optimize this function by introducing \_CACHE dictionary as follows:

\_CACHE = {}

def fib(num: int) -> int:

try:

return \_CACHE[num]

except KeyError:

if num == 1: return 1

if num <= 1: return 0

result = fib(num - 2) + fib(num - 1)

\_CACHE[num] = result

return \_CACHE[num]

However, you can also implement this function using generators:

def fib(n):

a, b = 1, 1

for \_ in range(n):

yield a

a, b = b, a + b

Now we can get traverse the generator:

for num in fib(10):

print(num)

## Cache

### What is Cache? LRU cache?

Cache chip was designed for enhancing CPU capabilities so that it could perform calculations faster. It does not concern only Fibonacci or Factorial functions, but more. For example, in Linux, the systemd is the main service of OS, Linux, which controls other services like NGINX (webserver daemon). Daemons can benefit from cache memory. Not only does cache chip stores data, but also instructions for CPU to execute. As known, the better strategy is to actually split cache large chip into several smaller ones, which enhances the performance of the CPU.

When CPU needs the required data for a program, it does not check RAM first; instead, it checks cache memory, and if it found it, then we call this ‘cache hit’; otherwise, CPU sends the request to the controller and retrieves the data from RAM, which is then stored in cache. There are two ways of it:

1. ‘Cache read’: when we observe “cache miss”, CPU has to fetch the block of data (remember that it does not do so by processing RAM by 1 byte; rather, it gets data in chunks of data by 64 bytes), it stores data in cache memory, but does not immediately update the main memory.
2. ‘Cache through’: It also updates the main memory so that there is a new copy of data stored in cache memory.

The data that is stored in cache memory can be differentiated into the following types:

1. Least recently used,
2. Least frequently used,
3. Most recently used,
4. Most frequently used.

Thus, we can store least recently used data in cache memory or most frequently used one.

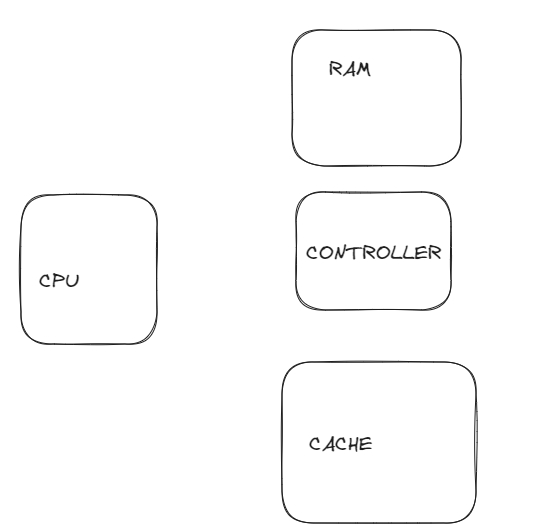


Fig. 2.: What is Cache?

### Cache levels

The first thing to notice is that Intel CPU’s cache chip has three levels: L1, L2, and L3. The first one is the fastest, but has the lowest capacity to store data; the middle one is slower than the previous but offers more space for data; finally, the last one is the slowest, but has the largest capacity. This levels can be exclusive, inclusive or general type.

The idea behind splitting these levels into special types such as L1 and L2 are inclusive lies behind algorithms of searching and retrieving data:

1. Inclusiveness means that L2 copies the data from L1, which means if L1 does not have data, CPU will not proceed searching in L2, saving time; the con is that L2 can store a fewer amount of data.
2. Exclusiveness means that L2 does not copy data from L1, but may have its own cached data. Hence, CPU will proceed searching L2 for the required data unless L1 has it. The con is the performance: it is slower than the previous one, but may store more data.

The general type means that both L1 and L2 may contain similar data or may not. In case of multiple cores, we can the shared level, L3, can be inclusive, exclusive or general; the nature of L3, in this case, affects how CPU is going to search for data:

1. If L3 happen to be inclusive, then CPU will not search in other cores
2. Otherwise, it might search for data in other cores.

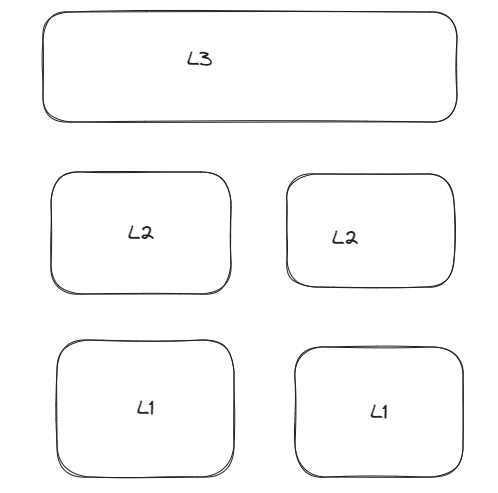


Fig. 3.: Cache level hierarchy

### Cache lines

The data is stored in cache as follows:

1. We store tag, which is address of a block of main memory, RAM; why 26 bits? If we look at the 32-bit representation, the last 6 bits are zeros. The cache lines are different by the last two bits: 00, 01, 10, 11, meaning: 0, 1, 2, and 3 as lines of cache memory. Thus, 32 minus 6 is 26, and tag field is only 24 bits, so we can omit other two zeros.
2. Address of line corresponding to data in main memory, 2 bits;
3. Offset: 4 bits for searching for the data, and 2 bits for searching the byte.

The size of cache which is specified is called “useful size”; but the total size is the sum of the useful size and tag size:

This tag is shown below. Thus, if cache has a useful size of bytes, then each block of memory is 4 bytes and we have 32 such blocks; means that each line in tag is 4 bytes and we have 4 lines. So, bytes added to 128 bytes means the full size.

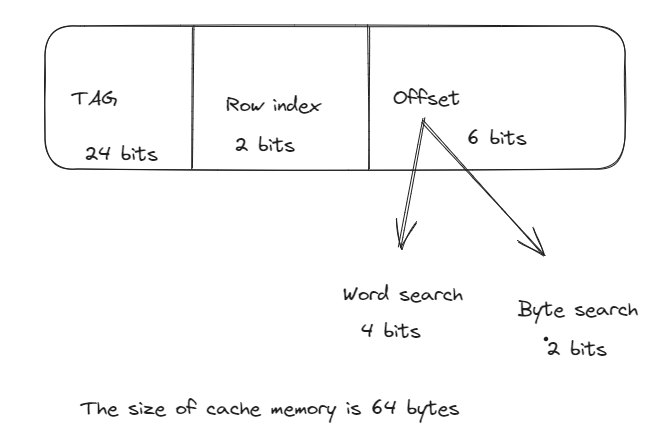


Fig. 4: Cache memory layout (simplified)

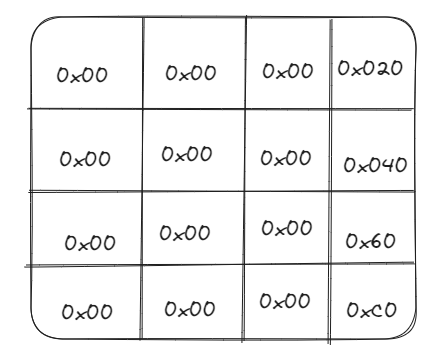


Fig. 5.: Tag memory layout of 32-bit address

The cache memory can have several lines of RAM in one line, and to determine our line we must follow the calculations: we get the number, for example, 129 in RAM, and divide it by cache’s line size 64 bytes, and then we take the remainder of this number by 4, number of lines; the obtained number, 2, is the line in RAM which has 129 (remember the start index is 0). The process is called direct mapping: each block of main memory maps to exactly one line of cache memory.

1. The first line maps to 1, 5, 9, 13 blocks of RAM;
2. The second line maps to 2, 6, 10, 14 block of RAM;
3. The third line maps to 3, 7, 11, 15 blocks of RAM;
4. The last line maps to 4, 8, 12, 16 block of RAM.

The logic pattern is the following: “Memory address” is divided by the number of lines in cache memory and then the remainder is the exactly index of the cache line. The disadvantage of such mapping is that different lines of RAM tend to take the same line in cache memory, thereby colliding.

To lower collision rate, computer scientists came to the concept of set-associative mapping: cache memory is divided into several segments, called, channels; they are associative with RAM blocks of bytes, meaning that those blocks can be situated anywhere in cache memory, where they correspond to exactly one line of cache memory. The number of channels is divisive by .

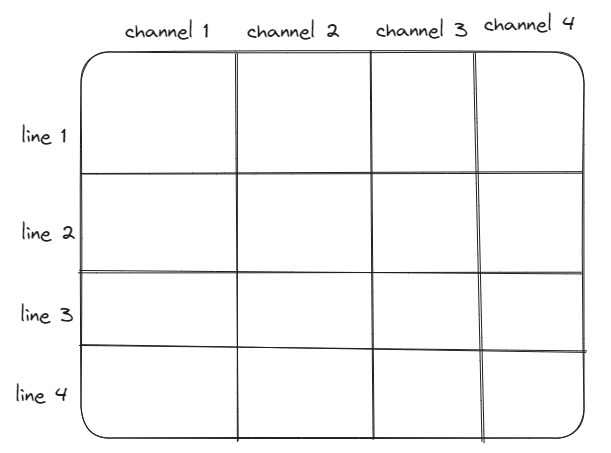


Fig. 6.: Set-associative mapping of cache memory

To determine the line of 129 in cache line, we must follow the same idea discussed previously:

and then compare with the of address. Thus, our 129 might lie in the third line of cache memory in any suitable channel.

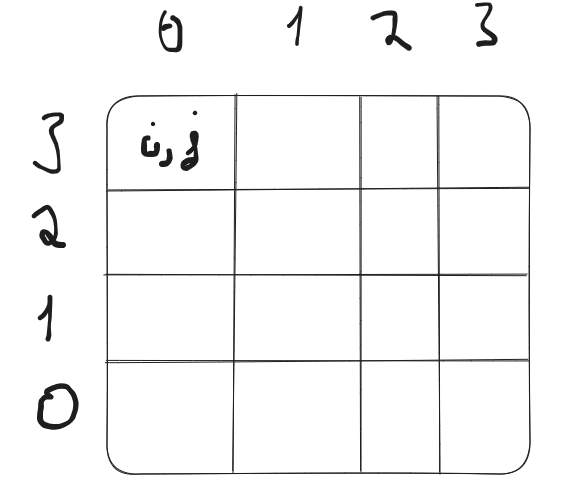


Fig. 7.: sets of set-associative mapping

The full associative mapping is that any line of RAM can be situated in any cache memory line. The problem is that the performance is slow, because we have to loop over all lines of cache comparing their tags.

## Lazy loading

### Iterators

To explain what is lazy loading concept meaning, we have to study iterators. The keyword is ‘iterator’, and it is a method of how objects should be iterated. For example, to traverse an array, we must go over all indexes and perform some actions and then move on. Let us write some Python script:

my\_list = list([1,2,3,4,5,6,7,8,9]) # create our list

it = iter(my\_list) # initialize iterator

for \_ in range(10):

print(next(it))

The explanation of happens is quite simple: we retrieve the elements whenever we call next till the end of the list. When reaching the end, Python raises StopIteration exception and completes iteration. At this point, out iterator is empty and is going to be dereferenced, which is a process of garbage collection.

### Generators

Generators are objects that we can use for implementing lazy traversing: traverse the list by one element. For example,

def traverse(input\_list):

for element in input\_list:

yield element

gen = traverse([1,2,3,4,5])

for num in gen:

print(num)

### Implementation

For example, we can traverse elements from queue and validate them while fetching data from user using generators (Sergei Ivanov, 2023, Homework V, Python programming, [link](https://github.com/derweisskrag/Python-Course-Homework/blob/main/homework/HW5.py)):

def process\_data() -> Data:

"""This function processes the data

"""

# Initialize queue and ordered dictionary to store our data

data = OrderedDict({

"name": None,

"grades": None

})

queue = deque([])

# Process data being fetched

for info, key in zip(fetch\_data(), data.keys()):

queue.append(info)

# handle data in queue

while queue:

value = queue.popleft()

handle(value, key, data)

return data

## Optimization of loops

### Loops

In programming, we use loops to iterate over iterable objects: strings, lists, dictionaries and other objects which must specify “\_\_iter\_\_” in their source code:

name = “Se#rgei”

filtered = “”.join([char for char in name if char != “#”])

print(filtered)

Using for means that the iteration index is incremented automatically, and we should not manually increment it. Whenever we break or return, the loop terminates, so, for example:

def find\_char(input\_string: str, target: str) -> str:

for char in input\_string:

if char == target:

return char

print(find\_char(name, “#”))

The same is true for while loops:

left = 0

right = len(name) - 1

while left <= right:

if name[left] == “#”:

print(name[left])

break

left += 1

right -= 1

The loops can be nested, and it is useful in some algorithms:

def remove\_char(s: str, target: str) -> str:

s\_list = list(s)

right = 0

for left in range(len(name)):

if name[left] != target:

s\_list[right] = s\_list[left]

right += 1

return “”.join(s\_list[:j])

The simplest way of visualizing the nested for loops is actually sorting algorithms.

### Examples of algorithms

The first example is Hoare sorting algorithm. To implement it in Python, we must choose some pivot element, for example, middle one:

def sort\_hoare(arr: list[int], start: int, end: int) -> int:

left = 0

right = len(arr) – 1

pivot = arr[(right + left) // 2]

while left <= right:

while arr[left] < pivot: left += 1

while arr[right] > pivot: right -= 1

if left <= right:

# perform swap

arr[left], arr[right] = arr[right], arr[left]

left += 1

right -=1

return left

def quick\_sort(arr: list[int], start: int, end: int) -> int:

if start > end: return None

left = sort\_hoare(arr, start, end)

quick\_sort(arr, start, left)

quick\_sort(arr, left + 1, end)

# Daemon

Daemon is a process running as a standalone one or in the background of another. For example, we can create a daemon which initializes the server service and makes HTTP requests for us; or we can fork a daemon whose purpose is to re-render webpage based on some file changes. Such a daemon is actually nodemon which is also used in NEXT.

Popular daemons include:

1. Apache webserver;
2. Mongod;
3. NGINX;
4. HTTPd

How to write a daemon? First, we need to understand the concept such a forking.

# Homework

## Part I

In this section, I would like to present my solution to the first home assignment as an academic paper. According to HW’s protocol, I must choose a supercomputer and answer the following questions [Source: Moodle, Home Assignment]:

1. Where is it?
2. Performance?
3. Rank when compared to others in the world?
4. Purpose?
5. Who is the creator and who are users?
6. How much is it if one had to buy it using financial means such as EUR, or USD?
7. Build my dream PC, and share its details.

To satisfy 1-6, I can grab the information from the previous section and present it here as a table. To answer the given question, I have to fetch the following data:

1. Name,
2. Location,
3. Date time,
4. Rank,
5. Purpose,
6. Creator (company)
7. Cost.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Name | Location | Date Time | Rank | Purpose | Creator | Cost |
| Fugaku supercomputer | RIKEN Center for Computational Science | 2020 | 2 | Scientific Supercomputer,  Computational Science | [MEXT](https://en.wikipedia.org/wiki/Ministry_of_Education,_Culture,_Sports,_Science_and_Technology) | [Riken](https://en.wikipedia.org/wiki/Riken) |

Table 30. Homework assignment: Supercomputer details

[Sources: [Wikipedia](https://en.wikipedia.org/wiki/Fugaku_(supercomputer)) and [TOP500](https://www.top500.org/system/179807/)].

The next task is to build my dream computer. Basically, this question implies the specification of the target computer machine such as

1. OS,
2. GPU,
3. CPU,
4. Performance,
5. Goal.

To address hardware questions, I have to specify the SSD, HHD or other more specific hardware questions details. Also, I must provide a name for my computer. Now I cite my response to the HW as a table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | OS | CPU | GPU | Performance | Goal |
| AI polygon | Linux | 591, 872  (9,248 × 64-core [Optimized 3rd Generation EPYC 64C](https://en.wikipedia.org/wiki/Optimized_3rd_Generation_EPYC_64C) @2.0 GHz), | 9,408 × 220 AMD [Instinct MI250X](https://en.wikipedia.org/wiki/AMD_Instinct) | Motherboard:  ASUS ROG Strix B550-F,  RAM:  32GB DDR4 3600MHz,  PSU:  850 W Gold-rated PSU by EVGA,  Cooling:  Liquid Cooling Solution Corsair,  Storage:  SSD (1 TB NVMe), | The goal of this compute is to perform scientific research and  Develop AGI (Artificial General Intelligence) |

Table 31: My PC build

Thus, I completed my HW assignment.

## Part II

This is the second homework assignment due to 26.09.2023. It is related to binary operations. The challenge statement is the following: ‘Assume numbers are represented in 8-bit twos complement representation. Show the calculation of the following:

1. 6 + 13
2. -6 + 13
3. 6 – 13
4. – 6 – 13

Solution. The first one is the simplest here, but it can be used to evaluate expressions, as shown in my theory section where we derived and defined the algorithm for adding and subtracting any integers, except for floating. However, we can still evaluate floating by separating them by the delimiter period: becomes and . When separated, we can compute the main parts and then compute the period ones; once processed, we can merge them back into the single number.

For the first one,

def add(a: int, b: int) -> int:

while b != 0:

carry = a & b

a ^= b

b = carry << 1

return a

Thus, we divide the task into 4 subtasks:

1. Determine the carry.
2. Perform XOR between the input numbers.
3. SHIFT the carry.

Here, we denote and .

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
|  | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| Carry | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Table 33: The first iteration: performing logical AND.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
|  | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| Sum | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

Table 34: The first iteration: performing logical XOR.

Thus, we get the following results:

Since , we continue our iteration:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
|  | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Carry | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

Table 35: The second iteration: performing logical AND.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
|  | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Sum | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

Table 36: The second iteration: performing logical XOR.

Thus, we got the following:

Because , we continue the iteration:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|  | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Carry | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 37: The second iteration: performing logical AND.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|  | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Sum | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |

Table 38: The second iteration: performing logical XOR.

We got the following results:

With the being equal to zero, we terminate our iteration and return the result:

Thus,

or

Remark. We can find the binary representation by dividing the number by 2 and analyzing the remainder 8 times. The result is all remainders read in reverse: 1, 0, 1, 1 yields 1, 1, 0, 1 as an output for the binary representation.

The second one ‘-6 + 13’ is actually subtraction process, as we can read it: ’13 – 6’. Therefore,

def sub(a: int, b: int) -> int:

while b != 0:

carry = (~a & b)

a ^= b

b = carry << 1

return a

For demonstration, we compute the binary number for and , following the algorithm:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
|  | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

Table 39: The first iteration: performing logical NOT: bitwise inversion.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| Carry | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

Table 40: The first iteration: performing logical AND.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
|  | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| Subtraction | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

Table 41: The first iteration: performing logical XOR.

Thus, for the first iteration, we have

Because , we proceed to the next iteration.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
|  | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |

Table 41: The second iteration: performing logical NOT: bitwise inversion.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Carry | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Table 42: The second iteration: performing logical AND.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
|  | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Subtraction | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

Table 43: The second iteration: performing logical XOR.

Thus, we got the following results:

Again, because , we continue iterating:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|  | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

Table 44: The third iteration: performing logical NOT: bitwise inversion.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Carry | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 45: The third iteration: performing logical AND.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|  | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Subtraction | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

Table 46: The third iteration: performing logical XOR.

Thus, we get the following:

As , we terminate the iteration. The result of the subtraction is stored in :

The third example, ‘6-13’ is actually . So, we perform the binary subtraction and then multiply the result by or inverting and adding 1 in binary. Let us do it. In our previous calculation, we concluded that

So, is the answer for the operation . To get , we must simply invert and add one to it:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
|  | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
|  | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

Table 47: Performing logical NOT: bitwise inversion.

Now, let us add one to it:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Sum | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

Table 48: Performing the logical XOR.

Thus, corresponds to , and is the result of .

The last example, can be regarded as . Therefore, we first compute the sum as usual, and then invert and add one to the result. In this case, the result was , so

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number | 8-bit representation | | | | | | | |
|  | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
|  | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |

Table 49: Performing logical NOT: bitwise inversion.

Now, let us add one to it:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 8-bit representation | | | | | | | |
|  | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Sum | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |

Table 50: Performing the logical XOR.

Thus, the result is which corresponds to .

We should recall that all actions performed during this exercise, are fulfilled by the functions ‘add’, and ‘sub’ by adopting the input data to call them.

Express the following numbers in IEEE 32-bit floating-point format:

1. -5
2. -6
3. -1.5
4. 384
5. 1/16
6. -1/32

Solution. a) The sign bit is 1, because it is negative. The next step is to normalize the binary representation. Let us notice that is 5 in binary; omitting zero, we get . From here, we shift twice to get , and so the exponent is 129, and therefore, is the 8-bit part. The fractional part should also be represented in binary: . Adding 21 zeros, we get

Combining our findings yields the correct format:

This binary representation corresponds to .

The second one is the same; we see that the sign bit is again 1, because the number is negative; to construct the normalized binary representation, we seek the fractional part: where we removed one zero: . Therefore, the normalized representation is , and the exponent is 2; adding it to the bias, we get , which corresponds to in binary. Finally, the fractional binary representation is . Thus, putting results together, we obtain

which is in the 32-bit floating-point binary representation.

The third example is that we have to convert to 32-bit floating-point binary representation. To do so, we must notice that the sign bit is 1, because the number is negative. The number is already normalized, as , so the exponent is , and adding it to the bias, we get which corresponds to in binary or . The fractional part is actually . Hence,

The fourth example is that we must convert 384 in 32-bit floating-point binary representation; let us notice that the number is already positive, so the sign bit is 0. The binary representation of 384 is . Thus, we must shift 8 times to get ; the exponent is 8 and adding it to the bias, we get or . The fraction part is , so

During the fifth example, we are converting to 32-bit floating-point format. The number is positive, so the sign bit is 0. Next, we find the binary representation of : , since . To normalize it, we have to shift 4 times to the right to get ; therefore, the exponent is and, subsequently, adding it to the bias gives which is . The binary representation of the fractional part is ; hence,

The last example is . The number is negative, so the sign bit is 1; the second essential point is to represent the number in the binary format. Because , we find which is binary representation of the given fraction. During normalization, we are shifting 5 times to the right; therefore, the exponent is , and adding it to the bias yields . The number is . Hence,

Thus, the number is represented in 32-bit floating-point format, as desired.

## Part III

The first exercise of the current homework assignment is already done and present at the end of the previous one. The other one is to implement XOR. This was done in our theory part “XOR”. Here, we simply recall that XOR can be expressed in terms of AND, OR, and NOT as follows:

Also, we can implement this XOR logic in Python by looping over bits of both numbers in reverse. If bits are different, add 1 to the result string, and otherwise 0. After the loop terminates, we can reverse our result string and return it. In Assembly, this procedure would involve complexities. Hence, I leave it empty for now, even though I could successfully implement XOR subroutine using logical AND, NOT, and OR.

Let us try derive this expression. We define operation between and , willing to get the result , and write the expression as . The idea behind our is that different bits produce the bit in the binary of , otherwise . Therefore, we must compare bits using NOT, AND and OR to merge the results into single variable:

1. We compare bits starting with inverting and applying AND operator:
2. We do the same thing but now with : .
3. The obtained result is merged into single binary number using OR:

## Part IV

The fourth homework assignment is more difficult, because we must gain insight into how CPU gets data and can work continuously without interruptions. Moreover, multiple users can make use of the same computer and host their servers, while working independently. These ideas of how CPU can be optimized cover the current homework assignment topic.

The homework assignment is given by the picture:

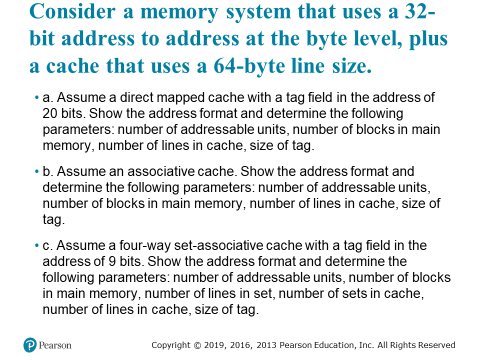


Fig. 8.: Homework assignment statement

Let us start completing it. The first subtask is defining a directly mapped cache memory with a tag field in the address of 20 bits. We must show the address format and determine the number of addressable units, the number of blocks in the main memory and number of lines in the cache, and the size of the tag.

We already know that the size of our cache is 64 bytes, and the line format is 32 bit. We have 4 lines in cache memory, and 16 lines in RAM. The index is 2 bits:

|  |  |  |
| --- | --- | --- |
| Tag field | Row index | Offset |
| 20 bits | 6 bits | 6 bits |

Tab. 51.: Address format for direct mapped cache memory

The number of lines in cache is lines, and the block 64 blocks of memory in RAM. We have addressable units, and the size of tag is 20 bits.

The second subtask is to assume an associative cache. We must show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in the cache, and the size of the tag, provided that the CPU uses the cache line size of 64 bytes and 32-bit address. Here, we do not have the tag size specified, so for 64-byte cache memory we have: 24 bits for tag, 2 bits for index line and 6 bits for offset (4 bits for machine word search and 2 bits for searching the byte), meaning that

|  |  |  |
| --- | --- | --- |
| Tag field | Row index | Offset |
| 24 bits | 2 bits | 6 bits |

Tab. 52.: Address format for associative cache memory

The number of addressable units is the same, ; the number of sets is 16 (indexed from 0); the number of blocks in RAM is 64 blocks.

The last one is to assume a four-way associative cache memory with a tag field in the address of 9 bits, and we must determine the following parameters: number of addressable units, number of blocks in RAM, number of lines in set, number of sets, number of lines in cache, and the size of tag. We have:

|  |  |  |
| --- | --- | --- |
| Tag field | Index | Offset |
| 9 bits | 17 bits | 6 bits |

Tab. 53.: Address format for a four-way associative cache memory

The size of the cache is not defined here, but we can assume it to be because:

1. The number of sets is ;
2. The index is .

From the second one,

Hence,

This is quite large, and we can consider it as L3 cache system. The number of lines in a set is the associativity level, which is, 4, so . The number of lines in the cache is

The number of addressable units is

Possible critical thinking leads to the fact that index is an integer, so

If we assume index equal to , then , and then

But this would be bit system. The number of addressable units would be . The number of blocks would be .

Let us put the answers together:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| System | Tag field (bits) | Index field  (bits) | Number of blocks in RAM | Number of Sets | Number of Lines in set | Number of lines in cache |
| Directed mapping cache |  |  |  |  |  |  |
| Associative cache |  |  |  |  |  |  |
| Four-way set-associative cache |  | (size dependable) |  |  |  |  |

Tab. 54.: Answer table

# References

## Primary

1. Standford Encyclopedia of Philosophy, 2018, Turing machines. <https://seop.illc.uva.nl/entries/turing-machine/>
2. NASM, The Netwide Assembler, Nasm official documentation of the version 2.16.01. https://www.nasm.us/doc/nasmdoc1.html
3. OneCompiler, 2023, Run and Share the code. <https://onecompiler.com/assembly/3zncgdzae>

## Secondary

1. XOR gate. (2023, September 12). In *Wikipedia*. https://en.wikipedia.org/wiki/XOR\_gate.