UCLouvain
École polytechnique de Louvain

Université Catholique de Louvain

- Séminaire en culture scientifique - Computers, from the transistor to the Operating System

Author:
Emile Villette

Supervisor:
Louis Navarre
Referent:
Benjamin Elias

April 19, 2022

Contents

Chapter 1

Introduction

1.1 Subject

Everyday, hundreds of millions of people use a computer, ranging from huge data center servers to personal computers. But these computers need consistent and precisely defined systems to function the way they do. Most people only use the front-most parts of a computer's software: Operating Systems. There are many OSs, the most popular ones being (in descending order) Android, Microsoft Windows, Apple iOS and Apple MacOS [1]. Amongst these, there are multiple versions, from MS-DOS to Windows 11, from MacOs Cheetah to Monterey, passing by Android Cupcake and Oreo, while not forgetting about all the Linux flavors. Moreover, these Operating Systems continually evolve to follow technological innovation, for commercial reasons, to adapt to the changing market or even to simply add new features which enhance the OSs' general usability. This results in multiple versions of the same software architectures. The first known Operating system was the *GM-NAA I/O*, designed by *General Motors* for the *IBM 704* computer in 1956 [2]. It was a quite "bare-bones" OS when compared to the ones we have nowadays and it wasn't usable or accessible to the general public. Today, the software we all used have a functioning Graphical User Interface, complete with WiFi, File Systems and lots of other features. Thanks to the advancements of User Interface (UI)/User Experience (UX) design, the usage and navigation within an OS seems pretty straightforward. But in reality, the underlying complexity of these systems is far more intricate than that of the general belief. How are electrical signals assembled together to piece what we know as computers ? This is the question that I will try to answer to in this essay.

To do so, I will try to describe the different software and hardware layers of computers. It is common knowledge that these are made of multiple smaller hardware components (storage, GPUs, CPUs, RAM,etc.) along with an OS. This is however a very restricted picture of the reality of things. Every second, an average computer processes billions of operations [3], showing that there is actually a lot going on at very small intervals of time.

To do so, I will start from the most basic component in a computer: the transistor. With that properly established, I will hop to the CPU, explaining its different components and how they interact with the "outside". Indeed, there needs to be a place to store data, and I will explain how this works and what the difference is between volatile and persistent storage.

With the basic architecture of a computer well in place, I will elaborate on low level programming, starting with the logical operators and progressively ramping up to assembly code, with the use of operation codes (the tool that allows a CPU to decode and apply instructions), instruction cycles (the process that the CPU follows to fetch and apply instructions one after another).

Further on, I will demonstrate the use of a higher order programming language and its compilation process (i.e. how it translates human-readable code to machine-readable instructions) along with a basic explanation of the kernel.

This upwards path pattern helps the reader to understand the principle of complex key computer science concepts, by establishing solid basis of the underlying notions whilst still being accessible to someone with basic computer science preliminaries.

1.1.1 Approach

This essay is not aimed towards the more technical population, anyone with a basic understanding of the fundamentals of the Computer Science field should be able to comprehend it. The intent is to portray the different software and hardware layers of computers, in a general manner and without going too much in the details whilst still using and referencing accurate information. In such, it should give the reader an overall understanding of how these systems work. It is important to keep in mind that this essay is an **overview** and

a major simplification of how a computer functions. It focuses on key components that are essential and does not cover everything present in a computer system.

1.1.2 Methodology

The methodology will be the following: I will start from the "bottom" (Central Processing Unit (CPU), Random-Access Memory (RAM) and permanent storage) by basically explaining how data is handled by those 3 major components. I will then proceed in an upwards manner, going up the different layers, to end up at the compilation of a programming language (i.e. C) and a basic portray of a kernel.

1.2 Acknowledgements

Firstly, Special thanks to Louis Navarre for supervising me on a weekly basis throughout the making of this essay, providing his expertise and insight to me. This was especially useful as the first university-level consequent essay that I had to write and helped me understand and get experience on good practices. Secondly, I want to thank Prof. Benjamin Elias for organising and managing this course in the scientific culture *mineure*. Lastly, I have to credit and give Kudos to Steven François, Tania Shafiei, Victoria Van Rillaer and Delphine Van Rossum for reading and providing feedback about their take on this essay.

Chapter 2

Hardware

As mentioned in 1.1.2, the focus of this essay is not the hardware. However, it is essential to elaborate on a few components.

2.1 The components

2.1.1 Binary

Most computers work with the binary number system. Binary number representation is a system where, as opposed to the conventional decimal system that we all use (numbers 0 to 9), there are only two states: 0 and 1. For instance, *19* in decimal translates to *10011* in binary [1](this essay assumes that the reader is already somewhat familiar with binary number representation, see section 1.1.1). A 0/1 is called a bit, 4 bits form a nibble and 8 bits form a byte [4].

This is especially critical and useful for a couple of reasons. Hardware-wise, it's easier to implement using transistors: the most basic electronic component in a computer. Transistors are basically tiny switches, they either let the current flow, or they don't (which is analogous with a 0 and a 1, on or off) [5]. This allows for an easy implementation of binary logic (see 2.1.2). It is comparable to a light switch, when it's on, the current passes through and the light switches on, and vice-versa. It would theoretically be possible to create computer circuits that use the decimal system, but it would however be exponentially more technologically demanding and would probably be very imprecise. To put it simply, the binary number representation system is the most convenient one given our modern technology. Another reason to choose binary is that it allows for a straightforward implementation of the basic logical operators (AND, OR, NOT, NAND, NOR, XOR and XNOR) [6].

2.1.2 The transistor

The transistor is the most basic electrical component in a computer. It is used almost universally in digital devices. Its *modus operandi* is very basic and only has two states: on and off. It has 2 inputs (which I define as IN and CTRL) and a single output (OUT).Thus, it only allows for 2 output states: 0 and 1[2]. Figure 2.1 shows a schematic for a basic MOSFET transistor[3], which will be used as a reference for the following explanation.

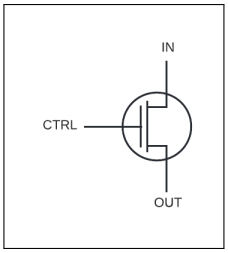

Figure 2.1: MOSFET Transistor

For the purpose of this explanation, let's set aside the notions of data bytes and bits (see 2.1.1) and only consider flowing current (and its opposite, i.e. the absence of flowing current). When current comes in through IN, it only goes through OUT if CTRL is set to 1. Otherwise, it doesn't go through the gate. When IN is set to 0, OUT is also 0 regardless of CTRL's state. However if CTRL is 1 but OUT is 0, it still indicates that the desired output is 0, which is an important information.

Transistors are used to build more complex components, such as logical operators (see section 3.1), which can carry out complex operations. Eventually, layer by layer, component by component, this results in the computers we all use every day. As a complement, Appendix A demonstrates

[1] 19 is $1 \cdot 10^1 + 9 \cdot 10^0$ in base 10. 10011 is in base 2: $10011 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$ in decimal.

[2] On = 1 (flowing current) and Off = 0 (no current).

[3] A MOSFET transistor is one of several types of transistors in existence. Each type has it's particularities, but the basic behaviour detailed in this sections remains identical. Explaining the differences is besides the point.

the transistor implementation of an AND gate. Other logical operators are implemented in a similar manner. Appendix B contains the truth tables[4] of several logical operators.

2.1.3 Central Processing Unit (CPU)

The Central Processing Unit (CPU) is the computer's brain and heart. Basically, it handles all the logical operations of the computer. There are two main different computing architectures: Reduced Instruction Set Computing (RISC) (e.g. ARM) and Complex Instruction Set Computing (CISC) (e.g. Intel x86). The difference between these two architectures goes beyond the scope of this paper, but basically, CISC has more instruction sets but is more power-consuming and RISC is more efficient, but also more verbose [7]. In this essay, I will be based on the RISC architecture, and more specifically ARM.

In a nutshell, the two main components in a CPU are the Arithmetic Logic Unit (ALU) and the Control Unit (CU). The ALU is a component that can handle basic arithmetic/logical operations: addition, substraction, multiplication, division, logical OR and logical AND [8] (and more). It has registers (tiny memory modules that can to handle information that needs to be temporarily retained for further computing/storage. In combination with the Control Unit, it can execute operation such as "if-then-else" or "while" loops. [9]. Control sequences and machine code is explained further in section 3.

The Control Unit is the part that coordinates all the operations within the Central Processing Unit. It controls the data transfer within, from and into the CPU, it also interprets instructions it's sent, and converts them to control sequences and control signals. It also processes and stores the results of the instructions it sent to the different parts of the CPU (registers, buffers, and the ALU) [10]. It's basically the nervous system of the computer, it tells it when and how to move according to what it's trying to achieve. The details of the inner workings of the Control Unit and the CPU are not covered by this essay, but it should give the reader a basic understanding of the logic of the computer's brain.

On Figure 2.2, Input/Output (I/O) units are interfaces outside of the CPU that can receive and send instructions (e.g. a display, keyboard, mouse, camera, etc.) [11].

CPUs typically have some cache. Cache is essentially like RAM (see 2.1.4), but it is located on the CPU dye itself. It is used to quickly access critical information (that it need to access very often and very fast). It's main advantage over RAM is that its transfer rate is 10 to 100 times the speed of RAM [12]. Another advantage it has is that it is physically closer to the CPU's computing chips, hence reducing the transfer delay. There are generally 3 layers of cache (L1,L2 and L3 cache), which has usages that go beyond the scope of this essay. Cache is very pricey, there are generally only a few Megabytes (MBs) of cache on CPUs (e.g. there is 30MB of it on the Intel Core i9 12900K, which is the most high-end consumer CPU, as of December 2021 [3]). It is also inconvenient because of its need to be close to the compute units (making it complicated to design and physically place on the dye).

See Figure 2.2 for a basic CPU schematic.

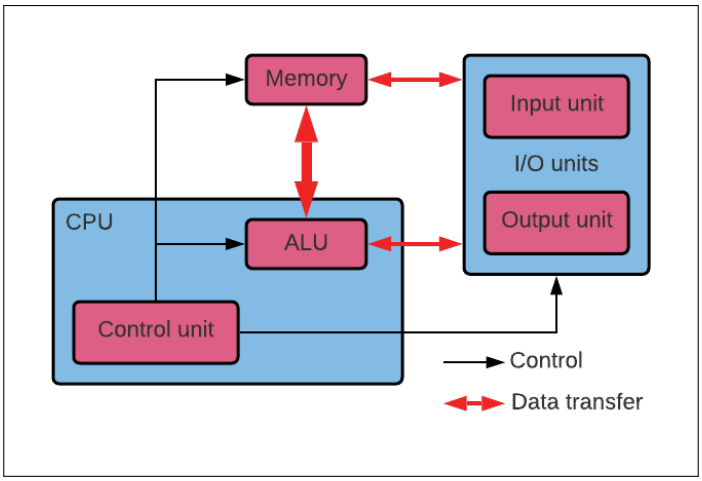

Figure 2.2: Basic CPU schematic

[4]Truth tables are the tables that indicate a given component's output depending on its inputs.

2.1.3.1 Arithmetic Logic Unit

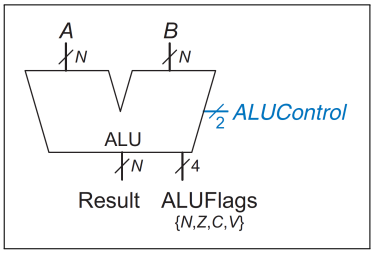

Figure 2.3: ALU Symbol with output flags [13]

The Arithmetic Logic Unit (ALU) is the part of the CPU that handles logical and arithmetical operations. It is the heart of most computer systems. It can perform a number of operations such as (but not restricted to) additions, subtractions, logical OR and logical AND. Figure 2.3 shows the symbol for an ALU with two N-bits inputs and a single N-bit output. The ALUControl input (in blue on Figure 2.3 specifies with operation needs to be carried out by the ALU (For instance $ALUControl = 00$ would mean ADD A and B). There is also a 4-bit output named ALUFlags, that serves as an indication out the output's characteristics. There are four different flags: N, Z, V and C. Respectively, this indicates if the ouput is negative, or zero or if the adder[5]produced a carry-out (for instance, in a single decimal adder, 9+1 produces a carry out of 1) or if it overflowed. Of course, the V and C flags are only useful in case the ALU is asked to perform and addition or a subtraction. As shown on Figure 2.4, the ALU has a Multiplexer (MUX), which is a component that can output one of multiple (in this case 4) inputs depending on a control signal that indicates which of the inputs should be outputted (in this case the control signal is ALUControl). The AND and OR operations are performed with AND and OR logic gates. The addition is simply evaluated with an adder. On the other hand, the subtraction (A - B) is performed with the addition between A's original form and B's two bit complement (meaning that $A + \overline{B} + 1 = A - B$) [13].

Moreover, more advanced ALUs are capable to handle other operations such as multiplication, division, bit shift and rotation, XOR operators, equality comparison, etc [13]. The previous paragraph portrays a simple ALU for the purpose of the explanation and may not be representative of the practical reality.

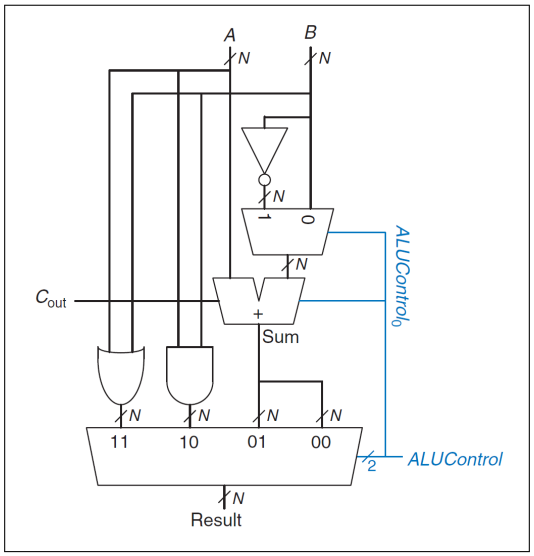

Figure 2.4: N-bit Arithmetic Logic Unit schematic [13]

2.1.3.2 Graphics Processing Unit (GPU)

A Graphics Processing Unit (GPU) is the part of the computer that specializes in graphics. It works similarly to CPUs. Historically, it was integrated in the CPU, but eventually became its own component as the demand for graphics processing power grew exponentially over time. It is usually a standalone card, that speaks directly to the CPU via a 16 lane PCIe bus (PCIe is a data transfer protocol). Many modern CPUs often have an integrated Graphics Processing Unit (iGPU), meaning that there it is directly included in the CPU dye [14]. Of course, it is multiple times less powerful than a standalone card, but it also consumes less energy, making it an advantage for lightweight workloads on battery-powered devices. Some laptops have both a standalone GPU and an iGPU. In that case, they intelligently switch between the two, depending on the workload. In this essay, I assume that the graphics processor is embedded in the Central Processing Unit to make it simpler.

[5]The adder is the component that adds two N-bit (or more) inputs and outputs the result of the operation. It can also determine if there is a carry-out on the operation.

2.1.4 Random Access Memory (RAM)

Random-Access Memory (commonly known as RAM) is a hardware component found in every computer on the planet. It is a volatile storage technology, meaning that data can only be stored in it when it is powered on. It is volatile because of physics: the higher the resistance of the data cell, the hotter and slower it gets. In order to keep these high reading and writing speeds, the resistance of the cells in RAM has to stay low, thus making it volatile (low resistance makes the capacitors leak energy and thus losing their stored bit over time) [15]. It is not made for long-term data storage, but rather for temporary storage. Its advantage is that it's a very fast type of memory, making it perfect to be transferred from/to the permanent storage and accessed quickly by the CPU [16]. It's speed has evolved over time, as technology advanced, ranging from 533 MB/s for PC66 memory to 12.8 GB/s for DDR3 memory [17], and even more for DDR4/DDR5 memory. It is used as an interface between the slower permanent storage and the CPU (for more details see 2.2). How the data is stored accessed and moved is detailed in section 3.3.

2.1.5 Persistent storage

Persistent (or permanent storage) is a type of computer hardware that can retain information, even when powered off. Typically nowadays we use two types of permanent storage: Hard Drive Disks (HDDs), cheap and high capacity but relatively slow and Solid State Drives (SSDs), blazing fast but expensive and low capacity (they can be used with both the Serial Advanced Technology Attachment (SATA) and Peripheral Component Interconnect express (PCIe) interfaces data transfer protocols, the latter being even faster) [18]. Historically, we used to have a lot of Floppy Disks, tape storage, Compact-Disk Read-Only Memory (CD-ROM), etc. They are persistent storage technology, which are nowadays obsolete due to the technological advancement, which produced faster and more efficient means of storage (such are portable hard drives, USB stick, and generally larger internal storage solutions). This type of storage is used to store the bulk of the data: your pictures, games or whatever is stored on it. I can scale up to Terabytes of information (trillions of bytes !).

Hard Drive Disks (HDDs) are one of the most used storage technologies as of December 2021[6]. As opposed to Flash memory technology, HDDs have moving hardware components: the read/write heads and the platters (also referred to as disks). The data is stored on the platters themselves, on tiny cells, polarized one way or the other (one way being a 0, and the other being a 1, which illustrates the convenience of the binary system, as mentioned in section 2.1.1). Over the years, the amount of data stored in a hard drive has increased (following Moore's law, which predicted a 2x increase of the capacity every two years [19]). There are usually multiple disks stacked on top of each other in a single Hard Drive Disk. The amount of data stored varies from one drive to another. The platters spin around a central spindle. The spin rate can be as low as 1200 RPM[7] in its early stages, and as high as 15000 RPM for modern enterprise-grade drives (although they typically spin at 5400 or 7200 RPM) [20].

How HDDs work is the following (cf. Figure 2.5): the read/write heads are positioned at 5nm above the platters, where the data cells are located. These data cells are made of microscopic metal grains, that when polarized, indicate either a 1 or a 0 depending on the direction. the heads are able to read data thanks to a small magnetic reader at the end of the arm, capable of interpreting the polarization of the cell. To write, it uses a ferrite magnet, also positioned at the end of the arm, that creates a small magnetic field, which is able to quickly change phase[8] to write data at a fast pace. However, the disks are not made out of a single line of data cells, but they are rather lines of cells organized in circles, whose circumference grow the further from the center of the platter they are. In such, the Read/Write heads need to position themselves on the right "data line" to read the correct data. To do so, the arm rotates around the Pivot. The arm is moved by the Voice Coil Motor, also referred to as an actuator. When it is in the correct position, it can finally read the data. Keep in mind that this process happens extremely fast, allowing the drive to read/write dozens of Megabytes per second [19]. There are of course many more complex mechanism involved in this matter, but this goes beyond the scope of this essay.

The other mainstream type of storage devices are Solid State Drives (SSDs), which do not work the same way as HDDs. They do not have any moving part, platters or heads. Instead, it is closer to the modus operandi of RAM (cf. section (2.1.4). It works by storing the data in NAND[9] memory, that itself is made up of floating gate transistors. As opposed to the transistors in RAM, they are designed to be non-volatile (meaning they retain data even when powered-off) whereas RAM transistors have to be refreshed multiple times per second. When compared with Hard Drive Disks, Solid State Drives are multiple times faster, but also tend to degrade faster over time due to due to physical phenomena [22].

[6] I have found no source confirming this, but when searching the keywords "storage device" on popular online retailers (Amazon, BestBuy, etc.), the vast majority of the results are HDDs. Therefore, I think that it is a reasonable assessment.

[7] Rotations Per Minute

[8] Meaning that it alternates between writing ones and zeroes

[9] Not And

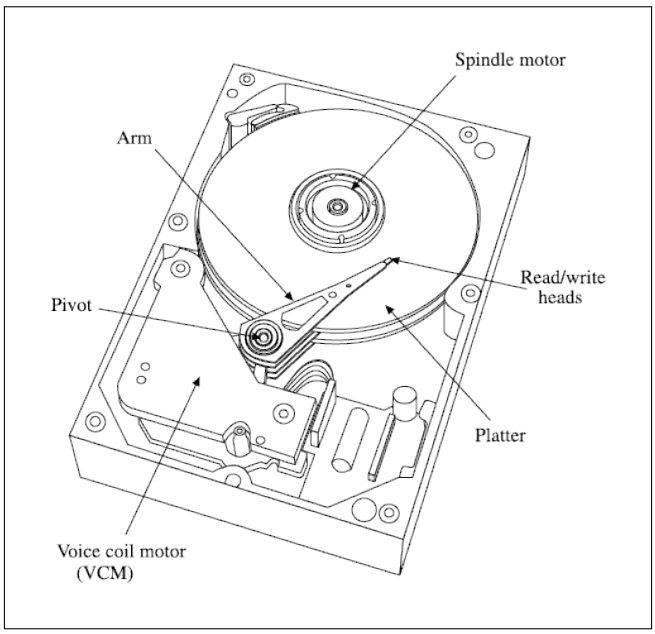

Figure 2.5: Hard Drive Disk schematic [21]

In general, SSDs are used for programs that need to access data quickly (e.g. Operating Systems and other data intensive programs). HDDs are mainly used for bulk data storage.

2.2 Integration between the components

In section 2.1, a brief description of the 3 main components of a computer is given (the CPU, RAM and the persistent storage). They are presented independently from one another, but in order to function, they have to interact with each other.

Figure 2.6 shows a basic organigram of a typical computer and its inner-workings.

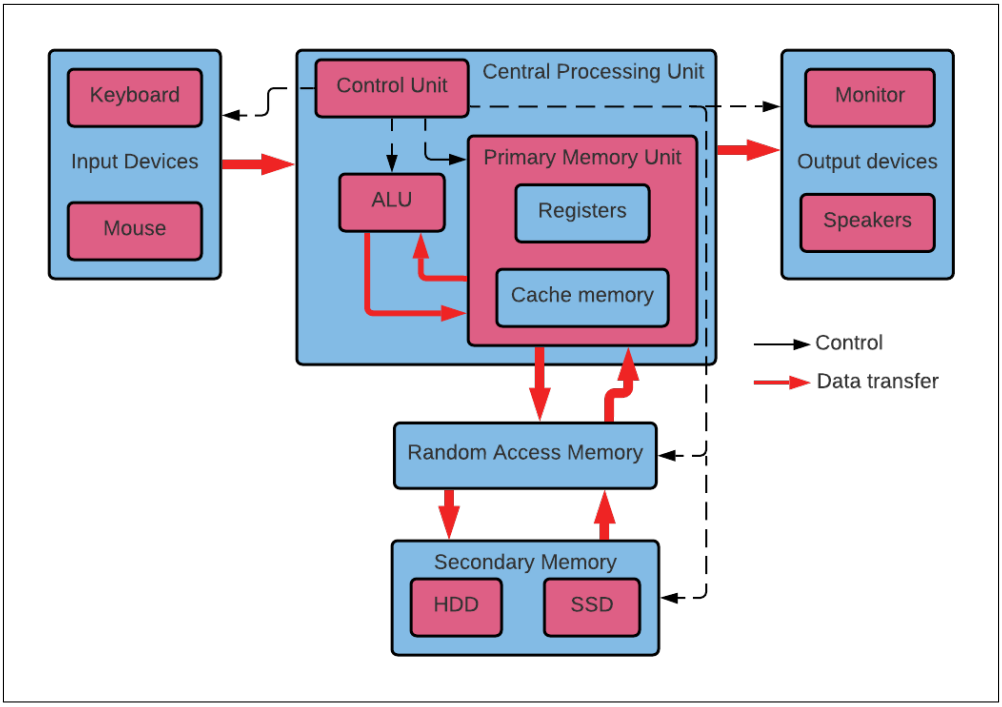

Figure 2.6: Computer block diagram

Chapter 3

Low level code

> *"Machine code, also called machine language, is a computer language that is directly understandable by a computer's CPU, and it is the language into which* ***all programs must be converted before they can be run****. Each CPU type has its own machine language, although they are basically fairly similar.* [...] *Machine code is extremely* ***difficult for humans to read*** *because it consists merely of* ***patterns of bits*** *(i.e., zeros and ones). Thus, programmers who want to work at the machine code level instead usually use* ***assembly language****, which is a human-readable notation for the machine language* [...] *are replaced with alphanumeric symbols (called mnemonics) in order to make it easier to remember and work with them (including reducing the chances of making errors). In contrast to high-level languages (e.g., C, C++, Java, Perl and Python), there is a nearly* ***one to one correspondence*** *between a simple assembly language and its corresponding machine language."*, as mentioned by the Linux Information Project.[23]

3.1 Logical operators

Logical operators are functions which when given two binary inputs provide an output. When assembled in more complex schemes, they can be used to compose more elaborate systems (such as Arithmetic Logic Units). There are 7 logical operators, with 7 unique truth tables (which can be found in Appendix B)[6]. It is possible, using these operators to build other operators which can take in 3 inputs (and more).

Virtually every logic component in a computer uses logical operators (a CPU is a complex assembly of billions of transistors, see subsection 2.1.3). But logical operators are also useful in programming. Let's take two predicates: A and B, which are either true or false (this is analogous to a bit). Using these operators, I can build a program that based on the inputs produces a given output (A AND B are present, A OR B is present, A AND NOT B, etc.). Modern programs are all built on these fundamental building blocks.

3.2 Operation codes

When a CPU receives bytes of information (or rather fetches these bytes, see section 3.3), the first step is to fetch an instruction. Within this instruction, there are bits of data which describe its nature (see Table 3.1).

Operation codes (typically abbreviated to opcodes) are a set of instructions addressed to the CPU that define what it should do with the associated data bytes. Opcodes vary widely, it can be mathematical operations to carry out, instructions to fetch data at a defined address, or a specific process to apply on that data (e.g. bit shifts, etc.)[24]. Each opcode is a string of ones and zeroes that the CPU is capable to identify. Though that is not practical for humans to read, opcodes also have a text representation, commonly referred to as *mnemonics*. For instance the operation "MOV" is "1101x" in binary ("x" is a *don't care* bit, meaning it can be either 0 or 1, it does not matter) [25].

Table 3.1 showcases the general binary format of an ARM data-processing instruction. There are other tables for other types of instructions. The first 4 bytes are "condition bits", meaning that if there is a condition set, the CPU will only execute the instruction if the condition is satisfied. Bits 27, 26 and 25 are (in this case) set to 0 to indicate that the instruction is of the "Data Processing type". Bits 24 through 20 are for the opcode itself. Bits 11 to 7 are for immediate operations that can be applied directly. The Op2 field can be used to differentiate the different variants of a given operation. The blank fields are unused in this case.

CPUs have a decoder that can find which instruction they need to carry out depending on the contents of the instruction. See section 3.3 for a more detailed explanation.

There are multiple different instruction sets depending (see Appendix C, which contains a table of all the ARM instructions) on the different industry standards (x86, ARM, see subsection 2.1.3). x86 systems typically

31 30 29 28	27 26 25	24 23 22 21 20	19 18 17 16 15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 0 0	op		imm5	op2	0	

Table 3.1: Encoding of ARM Data-Processing (register) Instruction [25]

have a larger instruction set, but tend to be more power-consuming, whilst ARM systems have a smaller set but are less prone to power consumption. This difference does not translate in their (in-)ability to execute certain tasks, but simply underlines another philosophy they undertake in their approach [7].

3.3 Instruction Cycles

As mentioned in section 3.2, a CPU uses opcodes to process and compute data. To do so, it follows a pattern called an Instruction Fetch Execute Cycle (which will be referred to as an IFEC). It is essentially made up of three repeating steps.

As illustrated by Figure 3.1:

0. (Only executed once) Start the system
1. Fetch the next instruction
2. Decode the received instruction
3. Execute the instruction
4. Repeat from step 1

It is also worth mentioning that there is in reality a 5^{th} step: the interrupt cycle, which essentially performs special instructions before shutting down the system so that the OS can resume it's operations once it's booted up again [26].

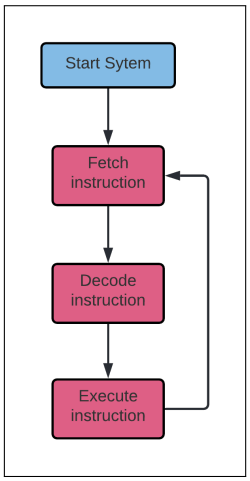

Figure 3.1: Instruction Fetch Execute Cycle Diagram

Figure 3.1 shows 3 main steps, but each of these steps are themselves composed of multiple processes. Although this goes beyond the knowledge presented in subsection 2.1.3, the components' names are somewhat self-explanatory[1]. Figure 3.2 represents in greater details a complete IFEC[2][27]. After the startup cycle, the address in the Program Counter (the register that stores the memory address of the next instruction) is copied to the Memory Address Register (the register that stores the memory address from which the data will be fetched/sent from/to the CPU, depending on the instruction). Then, the PC's pointer[3] is incremented to the next instruction. After that, the instruction found at the address fetched two steps earlier in the MAR is copied to the Memory Data Register (the register that stores the data from which the data will fetched/sent from/to the CPU, depending on the instruction). In the next step, the instruction stored in the MDR is copied to the Current Instruction Register (the register in the CPU that holds the current instruction being executed or decoded). All the steps that were just described constitute the *fetch* cycle. In Figure 3.2, the *Decode* cycle is a single step wherein the Control Unit decodes the contents copied to the CIR in the previous step. As mentionned in section 3.2, the CPU has a component, the decoder which, based on a string of bits (the instruction), can identify the operation (one of those present in the instruction set, see Table C.1), condition bits, etc. Finally, with the instruction fetched and decoded, the *execute* cycle can take place. Knowing exactly what it's asked to carry out, the Control Unit is now able to send the approrpriate signals to the relevant component (i.e. the Arithmetic Logic Unit, etc.). After the execute cycle, the CPU has two options: start a new IFEC or start the shutdown process if it is prompted to (the '5^{th} step' as mentionned above) [26].

[1] The understanding of the different components is left as an exercise to the reader.

[3] A pointer is an abstraction in computer science, which usually describes an object that stores the address in memory where its associated object is stored. Therefore, it 'points' to the location in the memory, making it a pointer.

[3] See entries PC, MAR MDR and CIR of Appendix E.

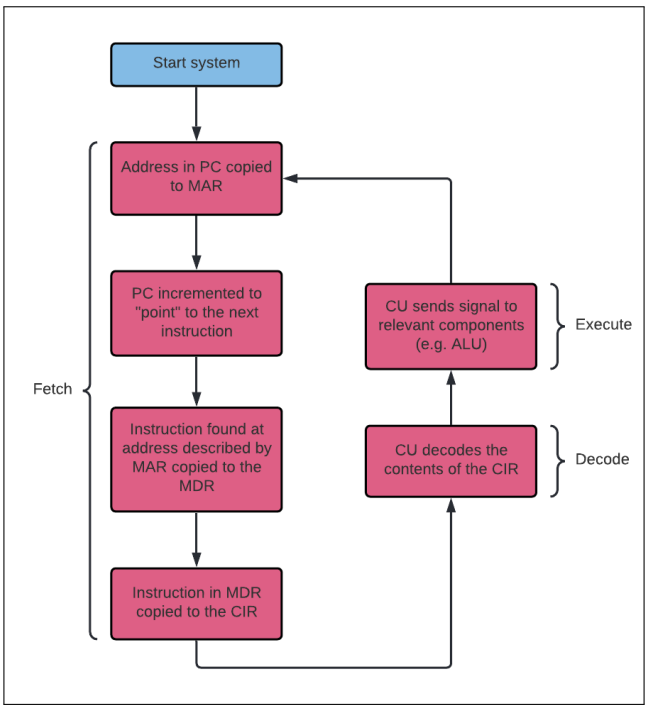

Figure 3.2: Further Detailed Instruction Fetch Execute Cycle Diagram [27][4]

The tedious (and yet still simplified) process described in the previous paragraph is a single full Instruction Fetch Execute Cycle. It is important to keep in mind that modern processors execute these cycles very quickly. To put it in perspective, one of the elements that factors in a CPU's performance is its "frequency" or "clock speed" (in Hertz[5], where 1 Hz is 1 instruction cycle per second). For instance, the Intel Core i9-12900k, which is one of the highest performing consumer-grade processors on the market (as of March 2022) runs at an advertised 5.2 GHz [3]. This means that an average CPU running at 4 GHz goes through 4 billion cycles every second, where a single cycle takes only 0.25 nanoseconds to execute[6] [28]. This quick example illustrates the sheer speed of modern computing.

3.4 Assembly language

Assembly language[7] is a low level language representing the computer's native language [13]. While it is possible to code directly in Assembly, it is highly unpractical. All the examples in this section are written in C, this essay assumes that the reader is already somewhat familiar with High level programming languages. The following example is a comparison between a "Hello World" program written in ARM Assembly and one written in C. As it is clearly visible, it is difficult to read, more tedious and takes an irrefutable longer time to use.

[5]Symbol: Hz
[6]Assuming that it runs in ideal conditions at a perfect and constant 4 GHz.
[7]Assembly differs between the x86 and ARM architectures. As a reminder, this essay covers ARM mainly.

This is written in ARM Assembly [29]:

```
.text
.global _start

_start:
    mov r2, #6   @ strlen
    mov r1, pc   @ load pc
    add r1, #24  @ add str offset
    mov r0, #1   @ stdout
    mov r7, #4   @ nr_write
    svc 0        @ syscall
    mov r0, #0   @ exit_success
    mov r7, #1   @ nr_exit
    svc 0        @ syscall

.asciz "Hello, World!\n"
```

This is the same program written in C:

```
#include <stdio.h>
int main() {
   printf("Hello, World!\n");
   return 0;
}
```

In the ARM Assembly example above, there is a clear pattern that emerges. It is a compilation of instructions followed by arguments. For instance, the instruction "mov r1, pc" copies the content of the Program Counter (PC) to r1, which is a special register in memory (see Appendix C for the details)[30]. This behaviour is clearly analogous to the one described about operation codes in section 3.2. In the aforementioned example, "mov" is the opcode, pc and r1 are the associated "information bytes". As mentioned in the introduction of chapter 3, there is a nearly one to one correspondence between Assembly and machine code, meaning that, with a bit of syntax sugar and processing, it is a quasi-direct translation. Therefore, Assembly (in our case ARM assembly) is called a low level language because it can (almost-) directly talk to the CPU

As the CPU can only understand strings of ones and zeroes, the Assembly instructions are translated into binary and then sent to the Central Processing Unit to execute the desired operation. For instance, the instruction "mov r1,pc" (previously illustrated) translates to "1111 0001 0000 1010 0000 1110 0001" (this may vary depending on the standard used) in binary ("0x0F10A0E1" in Hexadecimal representation). The processor is capable of decoding and executing that instruction using its various components previously mentionned.

Chapter 4

Higher Level Code

In this chapter, I will elaborate on high level code and compilers. Contrasting with low-level programming languages, high-level programming languages are easy to read for humans and more prone to fast development, where programming in Assembly (or binary) is highly unpractical (see section 3.4 for a comparison example). It is portable, easier to debug and develop, etc. This comes at the cost of an additional complexity layer, a compiler. A compiler is a process that translates ordinary code to machine code. As opposed to ARM Assembly (see section 3.4), this is not a straightforward process and requires significantly more effort to convert. While this is a certain trade-off, it is efficient because it increases the overall programmer's productivity. Nowadays, there exist very powerful compilators which are capable of producing assembly code that is as efficient as native assembly code.

4.1 C

Nowadays, there exists a second hierarchy where C is at the bottom of the ladder. This is partly because of its manual memory management (as opposed to Java, Python, and many more that have an automatic garbage collector), which makes this language relatively lower level. Although when compared to assembly or binary, it is considered as a high level language, it all depends on the viewpoint. However, in the scope of this essay, it is significantly higher level than for instance ARM Assembly. Originally developed in 1972 by Dennis M. Ritchie, it was designed as a successor to the B language [31]. To this day it is still being actively maintained and developed, and is one of the most popular languages, with many derived languages (C++, C#, Objective-C, Python/CPython, etc.). Indeed, while it is true that it is a lower level language, it is still being worked on because it is simply much faster than higher order programming languages, making it in uses that need speed over functionality (for example, the Linux kernel is partly written in C).

4.1.1 Memory management in C

As mentioned before, C does not have an automatic persistent memory management. This means that the programmer has to allocate memory manually. But it is also necessary to free (parts of) the memory when not needed anymore to avoid memory leaks (and crashes). In C, there are two memory areas: the stack and the heap. A stack is a computer science data architecture that follows the Last In First Out (LIFO) principle, which means that the last elements inserted are treated first. The heap is a hierarchical non-linear data structure, it can be seen as a space meant to be filled with elements, but not necessarily in order. The stack is not directly accessible by the user (to be clear: the user's code will determine what goes on and off the stack, but it will not directly interact with it) whilst the heap is manually allocated. A key difference between the two is that the Heap can access variables globally, and the stack can only access local variables (arguments, local function variables,etc.). This is a problem for big structures because a stack is essentially a big pile of instructions where the ones that get inserted first get treated last, meaning that a given instruction inserted early on can take a lot of time to get treated if it end up at the bottom of the stack. This is not a persistent structure because it is a series of instructions that do not need to stay in memory as it just begins from zero when the program gets executed. Overall, the stack is more efficient but less flexible, the heap is a good complement [32].

4.2 C compilation

As mentioned in the introduction of this chapter, compiling C to machine code is not a straightforward process. It is made of multiple complex steps. Very basically, the 4 main components are: the preprocessor, compiler,

assembler and linker (+ the loader) [33]. This section explains how the C language is compiled. However, this process is fairly similar in other compiled languages.

As illustrated by Figure 4.1:

1. The C program is expanded through the preprocessor. Macros are applied and libraries are imported, etc.
2. The expanded source code is then translated into assembly code by the compiler. This step is essential to the compilation process.
3. The assembly code gets assembled into object code. The object code's extension is ".obj" in DOS and ".o" in UNIX.
4. the linker links the object code, libraries and other object files into executable code. This is a very important step because a typical program written in C typically consists of multiple files and libraries[1]wherein function calls are made from one file but refer to a function written in another one. This effectively combines the code of the libraries with the code written by the programmer. This outputs an executable file.
5. Finally, the executable code is loaded in memory and gets executed by the CPU which executes numerous IFECs (see section 3.3).

Note: on the left of the blue blocks in Figure 4.1, there are examples of file extensions at each steps. ".exe", short for executable, is the final format that can be directly executed by the processor. Extensions are DOS standard, not UNIX.

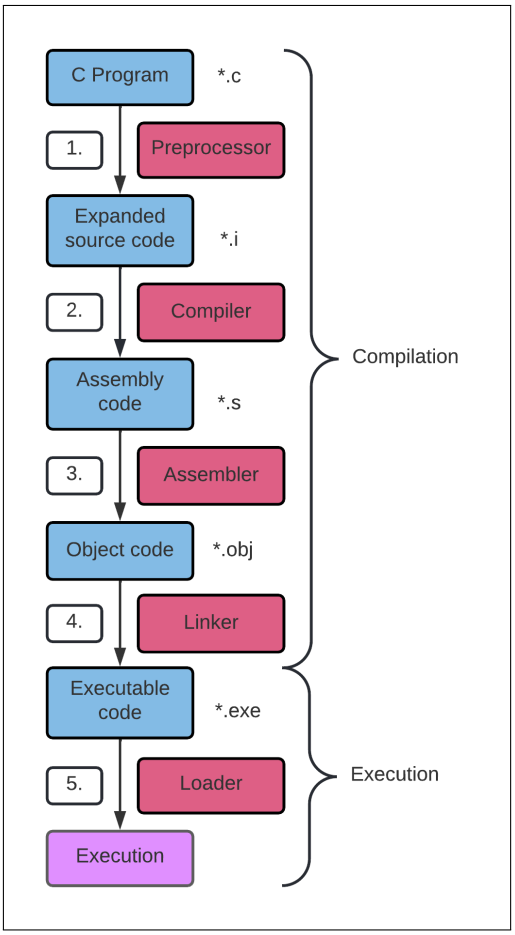

Figure 4.1: C compilation flow diagram [33]

4.3 Operating systems

4.3.1 Definition

C is a Turing-complete (and practical) language, which means it can compute everything that is computable by a computer. With the help of this tool, it is subsequently possible to program everything that constitutes an Operating System. Indeed, according to *Oxford Languages*, an Operating System is "the low-level software that supports a computer's basic functions, such as scheduling tasks and controlling peripherals". Technically, a basic OS would consist of a simple command line interface, with a working file system, and a possibility to interact with the hardware (such as network interfaces, SSDs, graphics processor, USB devices, etc.). Eventually, with these building blocks, it is possible to build features on top of the backbone of the OS in question. For example, it is possible to add a working Graphical User Interface, other programming languages, programs, drivers, functions, etc. In short, everything that is imaginable (as long as it can be implemented using a computer as defined in this essay).

[1]In the context of programming, libraries are utilities of pre-written directly usable code that avoids writing multiple times the same function and that often help to reduce programming errors due to their nature. They are also often pre-compiled, which avoids the overhead time necessary to their compilation.

4.3.2 Kernel

In Operating Systems, the kernel is the software component that supplies basic services for the other components of the OSs. It is a tool to manage memory allocation, networking, file systems, device control, etc [34]. The kernel also abstracts the hardware coordination in the computer, it makes sure that there is enough resources to handle a given program. Without that, the user could crash the machine at any given point. It can fulfill a number of functions and help with portability and ease of development of software. It is also useful in the cyber security field, it avoids making some software/hardware calls that would compromise the device's integrity. Figure 4.2 provides a basic diagram of a kernel. Modern kernels such as the Linux Kernel have dozens of components interacting with each other in a complex manner. This is illustrated by Figure D.1 in Appendix D, which is a map of the Linux kernel. It functions in the same manner as described in this paragraph.

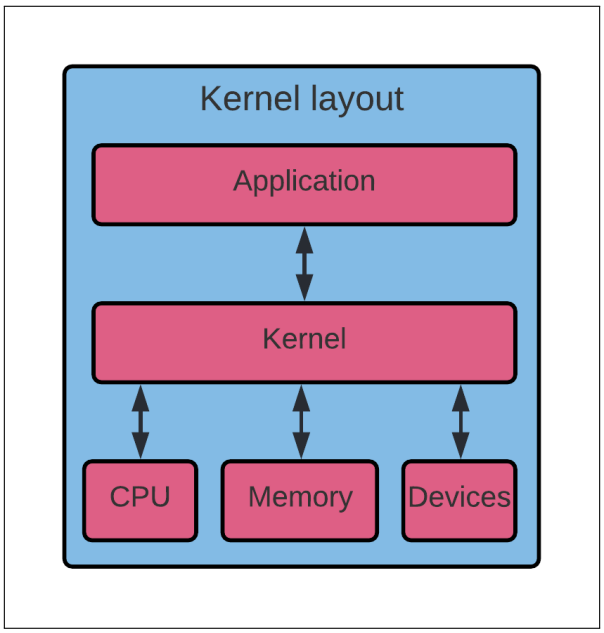

Figure 4.2: Kernel layout [34]

Chapter 5

Conclusion

5.1 Point of arrival

After elaborating on a number of computer science concepts, with all the building blocks established in this essay, the reader should now be able to understand all the fundamental notions that lead up to an Operating System. As mentioned a couple of times, this essay does not portray the complete picture of how a computer work, behaves and interprets signals and code. However, it does give out the bigger picture: the main hardware components, the basic electrical and electronic concepts, and the explanation on how code works and gets interpreted. Overall, this essay tries to enlighten that there is a strong (and voluntary) coherence between all the components of a computer and that making it work this well is an impressive feat. Although however impressive the feat is, it also highlights that there actually **needs** to be a well-thought an well-planned interaction between all the different parts of the machine, as the end-user needs the experience to be working as a whole. Almost every element presented in this essay is absolutely essential, meaning that if on of them dysfunctions, the whole system goes down with it.

5.2 What's next

Now, as this essay does not cover the whole extent of what goes on in a computer, there is a plethora of subjects left to explore, in both the hardware and software field. In the future, I might try to extend this essay to further complete it. On the one hand, concerning the hardware part, there are motherboards, networking, graphical chips, redundancy in storage, more elaborate modern concepts of what is in hardware nowadays, etc. On the other hand, the software applications are multiple, ranging from the simplest drivers to the most complex 3D modelling products. It would be possible to go on and on about what is possible to delve into. In short, I would virtually never run out of domains to explore. The Information Technology (IT) field is so huge that there's really only our imagination that limits the realm of possibilities.

Appendices

Appendix A

AND Gate MOSFET example

Source: [35]

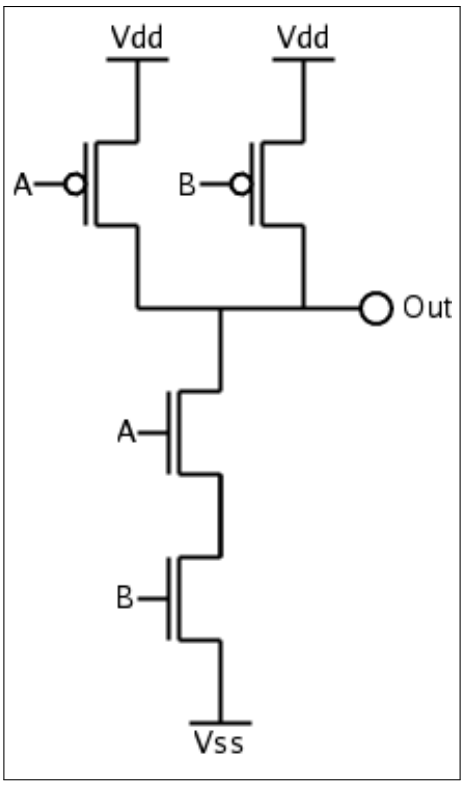

Figure A.1: AND CMOS Implementation

Appendix B

Logical Operator Truth Tables

B.1 Not

A	OUT
0	1
1	0

B.2 AND

A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

B.3 OR

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

B.4 NAND

A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0

B.5 NOR

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0

B.6 XOR

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0

B.7 XNOR

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	1

Appendix C

ARM Register names and instruction set

C.1 ARM register names

Source: [36]

Register	Synonym	Special	Role in the procedure call standard
r15	-	pc	Program counter.
r14	-	lr	Link register.
r13	-	sp	Stack pointer.
r12	-	ip	Intra-procedure-call scratch register.
r11	v8	-	ARM-state variable register 8.
r10	v7	sl	ARM-state variable register 7. Stack limit pointer in stack-checked variants.
r9	v6	sb	ARM-state variable register 6. Static base in RWPI variants.
r8	v5	-	ARM-state variable register 5.
r7	v4	-	Variable register 4.
r6	v3	-	Variable register 3.
r5	v2	-	Variable register 2.
r4	v1	-	Variable register 1.
r3	a4	-	Argument/result/scratch register 4.
r2	a3	-	Argument/result/scratch register 3.
r1	a2	-	Argument/result/scratch register 2.
r0	a1	-	Argument/result/scratch register 1.

C.2 ARM instruction set cheat sheet

Source: [37]

Mnemonic	Brief description	Page	Architecture[1]
ADC, ADD	Add with carry, Add	ADD, SUB, RSB, ADC, SBC, and RSC	All
AND	Logical AND	AND, ORR, EOR, and BIC	All
B	Branch	B and BL	All
BIC	Bit clear	AND, ORR, EOR, and BIC	All
BKPT	Breakpoint	BKPT	5
BL	Branch with link	B and BL	All
BLX	Branch, link and exchange	BLX	5T[2]
BX	Branch and exchange	BX	4Tb
CDP, CDP2	Coprocessor data operation	CDP, CDP2	2, 5
CLZ	Count leading zeroes	CLZ	5
CMN, CMP	Compare negative, Compare	CMP and CMN	All
EOR	Exclusive OR	AND, ORR, EOR, and BIC	All
LDC, LDC2	Load coprocessor	LDC, STC	2, 5
LDM	Load multiple registers	LDM and STM	All
LDR	Load register	ARM memory access instructions	All
MAR	Move from registers to 40-bit accumulator	MAR, MRA	XScale[3]
MCR, MCR2, MCRR	Move from register(s) to coprocessor	MCR, MCR2, MCRR	2, 5, 5E[4]
MIA, MIAPH, MIAxy	Multiply with internal 40-bit accumulate	MIA, MIAPH, and MIAxy	XScale
MLA	Multiply accumulate	MUL and MLA	2
MOV	Move	MOV and MVN	All
MRA	Move from 40-bit accumulator to registers	MAR, MRA	XScale
MRC, MRC2	Move from coprocessor to register	MRC, MRC2	2, 5
MRRC	Move from coprocessor to 2 registers	MRRC	5Ed
MRS	Move from PSR to register	MRS	3
MSR	Move from register to PSR	MSR	3
MUL	Multiply	MUL and MLA	2
MVN	Move not	MOV and MVN	All
ORR	Logical OR	AND, ORR, EOR, and BIC	All
PLD	Cache preload	PLD	5Ed
QADD, QDADD, QDSUB, QSUB	Saturating arithmetic	QADD, QSUB, QDADD, and QDSUB	5ExP[5]

RSB, RSC, SBC	Reverse sub, Reverse sub with carry, Sub with carry	ADD, SUB, RSB, ADC, SBC, and RSC	All
SMLAL	Signed multiply-accumulate (64 <= 32 x 32 + 64)	UMULL, UMLAL, SMULL and SMLAL	M[6]
SMLALxy	Signed multiply-accumulate (64 <= 16 x 16 + 64)	SMLALxy	5ExPe
SMLAWy	Signed multiply-accumulate (32 <= 32 x 16 + 32)	SMLAWy	5ExPe
SMLAxy	Signed multiply-accumulate (32 <= 16 x 16 + 32)	SMLAxy	5ExPe
SMULL	Signed multiply (64 <= 32 x 32)	UMULL, UMLAL, SMULL and SMLAL	Mf
SMULWy	Signed multiply (32 <= 32 x 16)	SMULWy	5ExPe
SMULxy	Signed multiply (32 <= 16 x 16)	SMULxy	5ExPe
STC, STC2	Store coprocessor	LDC, STC	2, 5ExPe
STM	Store multiple registers	LDM and STM	All
STR	Store register	ARM memory access instructions	All
SUB	Subtract	ADD, SUB, RSB, ADC, SBC, and RSC	All
SWI	Software interrupt	SWI	All
SWP	Swap registers and memory	SWP	3
TEQ, TST	Test equivalence, Test	TST and TEQ	All
UMLAL, UMULL	Unsigned MLA, MUL (64 <= 32 x 32 (+ 64))	UMULL, UMLAL, SMULL and SMLAL	Mf

Table C.1: ARM instruction set cheat sheet

[1] n : available in ARM architecture version n and above
[2] nT : available in T variants of ARM architecture version n and above
[3] XScale: XScale coprocessor instructions
[4] nE : available in E variants of ARM architecture version n and above, except ExP variants
[5] nE : available in all E variants of ARM architecture version n and above, including ExP variants
[6] M : available in ARM architecture version 3M, and 4 and above, except xM versions

C.3 ARM condition bits

Source: [25]

Condition	Mnemonic Extension	Meaning (integer)	Meaning (Floating-point)	Condition Flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not Equal	Not Equal or unordered	Z == 0
0010	CS	Carry Set	Greater than, equal, or unordered	C== 1
0011	CC	Carry Clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned Higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than, or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	CGreater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL)	Always (unconditional)	Always (unconditional)	Any

Table C.2: Condition code bits

Appendix D

Linux kernel map

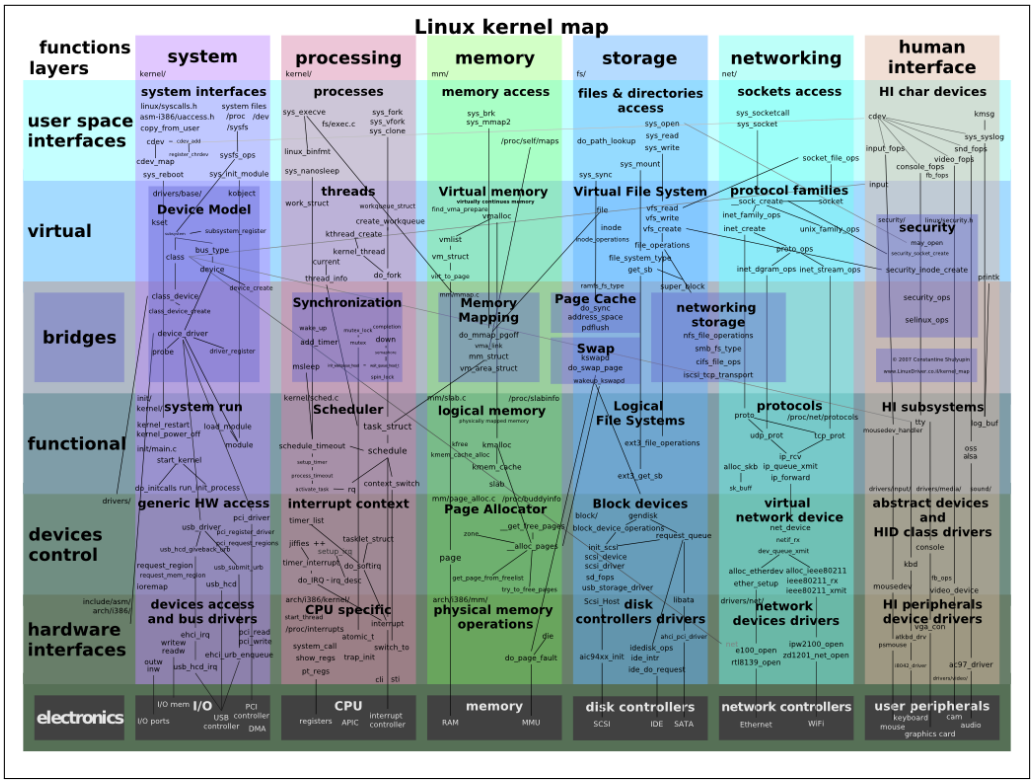

Figure D.1: Linux kernel map [38]

Appendix E

Glossary

E.1 Acronyms

ALU Arithmetic Logic Unit. 4, 5, 8, 9,

ARM Advanced RISC Machines. 4, 8–11,

CD-ROM Compact-Disk Read-Only Memory. 6,

CIR Current Instruction Register. 9,

CISC Complex Instruction Set Computing. 4,

CPU Central Processing Unit. 1, 2, 4–11, 13,

CU Control Unit. 4, 9,

DDR Double Data Rate. 6,

FIFO First In First Out.

FPU Floating Point Unit.

GB Gigabyte. 6,

GPU Graphics Processing Unit. 1, 5,

GUI Graphical User Interface. 13,

HDD Hard Drive Disk. 6, 7,

I/O Input/Output. 4,

IFEC Instruction Fetch Execute Cycle. 9, 10, 13,

iGPU integrated Graphics Processing Unit. 5,

IT Information Technology. 15,

LIFO Last In First Out. 12,

MAR Memory Address Register. 9,

MB Megabyte. 4, 6,

MDR Memory Data Register. 9,

MUX Multiplexer. 5,

OS Operating System. 1, 7, 9, 13–15,

PC Program Counter. 9, 11,

PCIe Peripheral Component Interconnect express. 5, 6,

RAM Random-Access Memory. 1, 2, 4, 6, 7,

RISC Reduced Instruction Set Computing. 4,

RPM Rotations Per Minute. 6,

SATA Serial Advanced Technology Attachment. 6,

SSD Solid State Drive. 6, 7, 13,

TB Terabyte. 6,

UI User Interface. 1,

UX User Experience. 1,

Bibliography

[1] StatCounter Global Stats. *Operating system market share worldwide*. Oct. 2021. URL: https://gs.statcounter.com/os-market-share (visited on 11/26/2021).

[2] George F. Ryckman. "17. The IBM 701 Computer at the General Motors Research Laboratories". In: *Annals of the History of Computing* 5.2 (Apr. 1983), pp. 210–212. ISSN: 0164-1239. DOI: 10.1109/MAHC.1983.10026.

[3] Intel. *i9 12900K Product Specifications*. 2021. URL: https://www.intel.com/content/www/us/en/products/sku/134599/intel-core-i912900k-processor-30m-cache-up-to-5-20-ghz.html (visited on 12/01/2021).

[4] National Science Foundation and Indiana State university. *What are bits, bytes, and other units of measure for digital information?* Jan. 2018. URL: https://kb.iu.edu/d/ackw (visited on 12/02/2021).

[5] Michael Riordan. *transistor | Definition & Uses | Britannica*. Mar. 2020. URL: https://www.britannica.com/technology/transistor (visited on 12/02/2021).

[6] Computer Hope. *What Is a Logical Operation?* Computer Hope. Nov. 16, 2019. URL: https://www.computerhope.com/jargon/l/logioper.htm (visited on 12/02/2021).

[7] Advanced RISC Machines. *Documentation – Arm Developer*. 2013. URL: https://developer.arm.com/documentation/den0013/d (visited on 11/30/2021).

[8] Encyclopædia Britannica. *arithmetic-logic unit*. URL: https://www.britannica.com/technology/arithmetic-logic-unit/images-videos (visited on 11/29/2021).

[9] Britannica. *computer - Supercomputer | Britannica*. May 2021. URL: https://www.britannica.com/technology/computer/Supercomputer (visited on 11/30/2021).

[10] *Introduction of Control Unit and Its Design*. GeeksforGeeks. Sept. 24, 2018. URL: https://www.geeksforgeeks.org/introduction-of-control-unit-and-its-design/ (visited on 12/06/2021).

[11] *What Is the Difference between an Input and Output Device?* URL: https://www.computerhope.com/issues/ch001355.htm (visited on 12/06/2021).

[12] Ben Lutkevich. *What is Cache Memory? Cache Memory in Computers, Explained*. URL: https://searchstorage.techtarget.com/definition/cache-memory (visited on 12/01/2021).

[13] Sarah I. Harris and David Money Harris. "Digital Design and Computer Architecture ARM Edition". In: (2016), p. 712.

[14] Intel. *What Is a GPU? Graphics Processing Units Defined*. Intel. URL: https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html (visited on 12/08/2021).

[15] Jason Fitzpatrick. *Why Does RAM Have to Be Volatile?* How-To Geek. URL: https://www.howtogeek.com/171872/why-does-ram-have-to-be-volatile/ (visited on 12/06/2021).

[16] Stacey Peterson and Rodney Brown. *What is RAM (Random Access Memory)? - Definition from SearchStorage*. Aug. 2021. URL: https://searchstorage.techtarget.com/definition/RAM-random-access-memory (visited on 11/30/2021).

[17] Joaquin Romo. "DDR Memories Comparison and overview". In: *Beyond Bits* (), p. 70.

[18] Techopedia. *What is Storage? - Definition from Techopedia*. Sept. 2020. URL: http://www.techopedia.com/definition/1115/storage (visited on 11/29/2021).

[19] TED-Ed, director. *How Do Hard Drives Work? - Kanawat Senanan*. Oct. 29, 2015. URL: https://www.youtube.com/watch?v=wteUW2sL7bc (visited on 12/07/2021).

[20] Puja Chatterjee. *What Is Hard Drive RPM - FAQ, List & Work Process*. eComputerTips.Com. July 9, 2020. URL: https://ecomputertips.com/what-is-hard-drive-rpm/ (visited on 12/07/2021).

[21] Da-Wei Gu, Petko H. Petkov, and Mihail M. Konstantinov. "Robust Control of a Hard Disk Drive". In: *Robust Control Design with MATLAB®*. Ed. by Da-Wei Gu, Petko H. Petkov, and Mihail M Konstantinov. Advanced Textbooks in Control and Signal Processing. London: Springer, 2013, pp. 249–290. ISBN: 978-1-4471-4682-7. DOI: 10.1007/978-1-4471-4682-7_14. URL: https://doi.org/10.1007/978-1-4471-4682-7_14 (visited on 12/07/2021).

[22] Joel Hruska. *How Do SSDs Work? - ExtremeTech*. May 3, 2021. URL: https://www.extremetech.com/extreme/210492-extremetech-explains-how-do-ssds-work (visited on 12/08/2021).

[23] BELUG. *Machine Code Definition*. URL: http://www.linfo.org/machine_code.html (visited on 02/06/2022).

[24] *Instruction Sets - Instructions - GCSE Computer Science Revision*. BBC Bitesize. URL: https://www.bbc.co.uk/bitesize/guides/z2342hv/revision/3 (visited on 02/07/2022).

[25] RealDigital. *Introduction to ARM Assembly Language*. URL: https://www.realdigital.org/doc/2f85bd55ef59ddbfcf64206f4421d5de (visited on 02/20/2022).

[26] Bosky Agarwal. *Instruction Fetch Execute Cycle - Archived*. June 11, 2009. URL: https://web.archive.org/web/20090611211308/http://www.cs.montana.edu/~bosky/cs518/ife/IFE.pdf (visited on 02/07/2022).

[27] HydenB. *English: This Is a Simple Diagram Illustrating Visually the Individual Stages of the Fetch-Decode-Execute Cycle*. Feb. 27, 2019. URL: https://commons.wikimedia.org/wiki/File:Fetch-Decode-Execute_Cycle.png (visited on 02/08/2022).

[28] Intel. *CPU Speed: What Is CPU Clock Speed?* Intel. URL: https://www.intel.com/content/www/us/en/gaming/resources/cpu-clock-speed.html (visited on 02/28/2022).

[29] *ARM Assembly – Hello World*. DEV Community. URL: https://dev.to/hextrace/arm-assembly-hello-world-33ep (visited on 02/16/2022).

[30] *Assembler User Guide: MOV*. URL: https://www.keil.com/support/man/docs/armasm/armasm_dom1361289878994.htm (visited on 02/16/2022).

[31] Dennis M. Ritchie. *Chistory*. The Development of the C Language*. URL: https://www.bell-labs.com/usr/dmr/www/chist.html (visited on 03/12/2022).

[32] Matthew Martin. *Stack vs Heap: Know the Difference*. Guru99. July 14, 2020. URL: https://www.guru99.com/stack-vs-heap.html (visited on 03/30/2022).

[33] JavaTpoint. *Compilation Process in C - Javatpoint*. www.javatpoint.com. 2022. URL: https://www.javatpoint.com/compilation-process-in-c (visited on 03/16/2022).

[34] *What Is Kernel? - Definition from WhatIs.Com*. SearchDataCenter. URL: https://www.techtarget.com/searchdatacenter/definition/kernel (visited on 04/02/2022).

[35] Dr. Lawlor. *CS 441 Lecture*. URL: https://www.cs.uaf.edu/2011/fall/cs441/lecture/09_05_semiconductors.html (visited on 02/15/2022).

[36] *ARM Developer Suite Developer Guide*. URL: https://developer.arm.com/documentation/dui0056/d/using-the-procedure-call-standard/register-roles-and-names/register-names (visited on 02/16/2022).

[37] *ARM Developer Suite Assembler Guide*. URL: https://developer.arm.com/documentation/dui0068/b/ARM-Instruction-Reference?lang=en#id2762298 (visited on 02/17/2022).

[38] *Linux Kernel*. In: *Wikipedia*. Mar. 31, 2022. URL: https://en.wikipedia.org/w/index.php?title=Linux_kernel&oldid=1080374498 (visited on 04/02/2022).

[39] The Editors of Encyclopaedia Britannica. *central processing unit | Definition & Function | Britannica*. Mar. 2021. URL: https://www.britannica.com/technology/central-processing-unit (visited on 11/30/2021).

[40] BBC. *Common CPU components - Computer systems - AQA - GCSE Computer Science Revision - AQA*. 2021. URL: https://www.bbc.co.uk/bitesize/guides/z7qqmsg/revision/4 (visited on 12/01/2021).