A Survey of Advanced Assembly Language Programming Techniques on the MOS 6502 and Zilog Z80

Sébastien Goutal
sebastien.goutal@gmail.com

July 22, 2023
v1.0

Abstract

This survey documents advanced assembly language programming techniques on 8-bit home computers that address typical limitations of 8-bit microprocessors (reduced instruction set, limited processing power). The detailed techniques include the implementation of mathematical functionalities (integer multiplication and division, floating-point arithmetic, function approximation), as well as optimization techniques (precomputed tables, loop unrolling, self-modifying code, undocumented instructions). The survey is limited to the MOS 6502 and Zilog Z80, and to the 8-bit home computers based on these microprocessors (Apple II, Commodore 64, BBC Micro, Amstrad CPC, ZX Spectrum). It is illustrated with examples of source code and reverse engineering of 8-bit software (built-in ROM, video games, demos) on these home computers.

Keywords— 8-bit home computer, MOS 6502, Z80, assembly language, optimization, loop unrolling, self-modifying code, undocumented instructions, demoscene

1 Introduction

This publication is an attempt at documenting specific programming techniques that can be found in software produced for home computers based on 8-bit microprocessors. The documented techniques address two types of limitations of 8-bit microprocessors. First, limitations from a functional point of view, as these microprocessors provide a limited instruction set. Second, limitations in terms of performance, that are a challenge for real-time applications such as video games. Some of these programming techniques have been marginalized over time, with the increasing sophistication of microprocessors, as well as the evolution of programming techniques and software development good practices. This survey is limited to the MOS 6502[1] and Zilog Z80 as most popular 8-bit home computers are based on one of these microprocessors. For

[1] And more generally, the MOS 6500 microprocessor family, see section 2.

instance, the Apple II, Commodore 64 (C64), BBC Micro and Atari 400/800 use a MOS 6502 or one of its derivatives, while the ZX Spectrum and Amstrad CPC are based on a Z80. Three main sources of documentation are considered.

The first source is the reverse engineering of 8-bit home computers' ROM, which provides many details on the implementation of arithmetic and mathematical features. Indeed, 8-bit home computers were shipped with a built-in ROM that is essential for their use. The ROM typically includes an operating system, that forms an abstraction layer on top of the hardware. For this purpose, functions are provided by the operating system to control the screen, the sound chip, the keyboard, the tape recorder or the disk drive. The operating system also provides core functions for memory management, and for the handling of interrupts and events. It is also customary for the ROM to include a BASIC interpreter. The BASIC interpreter serves as the main user interface for interacting with the operating system – for example, to load and execute a program. The BASIC interpreter also gives the user the ability to program with a high-level programming language. One of the key feature of the BASIC interpreter is its ability to evaluate complex mathematical expressions and perform floating-point calculus. To provide this feature, programmers had to implement from scratch the support of floating-point numbers, arithmetic operations, as well as common mathematical functions – a challenge as 8-bit microprocessors only support natively addition and subtraction of small integers. To this end, we will study the implementation of this feature in popular 8-bit BASIC interpreters – Integer BASIC, Locomotive BASIC, ZX Spectrum BASIC, Applesoft BASIC, Commodore BASIC and BBC BASIC.

The second source is the study of game programming techniques. For game programmers, development on 8-bit home computers or video game systems was seen primarily as a technical challenge, whether it is to achieve smooth and fast animated graphics, or to fit all the game data in memory [Hag18]. Game programmers on 8-bit home computers also faced the extra challenge of software piracy. As a result, they have explored many innovative techniques – including low-level optimization, data compression, procedural content generation, software protection and code obfuscation techniques [Ayc16]. We will illustrate our survey with optimization techniques used in several classic games on 8-bit home computers – including *Prince of Persia* on the Apple II, *Elite* on Acorn computers and *Manic Miner* on the ZX Spectrum [Mec89] [Mox20a] [Mox20b] [Mox20c] [Mox20d][Dym20b].

The third source is the *demoscene*. Since its inception in the mid-1980s, the demoscene has been an ideal playground for experimenting with coding techniques in a limited hardware environment. A notable digital subculture, the demoscene is an international community of computer enthusiasts interested in *demos*, where a *demo* is a non-interactive program that produces an audiovisual presentation [Reu10, ch. 1]. The demoscene first appeared on 8-bit home computers, and in this context the main purpose of a demo was to showcase the skills of the programmer, by pushing the limits of the hardware and showing things that the system should not be able to do [Reu10, ch. 4.3.1]. Being very competitive, the demoscene has contributed to the emergence of advanced optimization techniques, in a never ending quest for performance. With more than 20 millions units sold worldwide, innovative video and audio features for its time – thanks to the VIC-II and SID chips – the C64 attracted many hobbyists which contributed to the emergence of a very active demoscene. Other 8-bit home computers – such as the Atari XL/XE, the ZX Spectrum and the Amstrad CPC – also had their respective demoscenes, however significantly smaller, and usually localized in certain countries, where these models were popular. An example is the Amstrad CPC, the most sold 8-bit home computer in France, that led to the emergence of an active French

demoscene [Act93]. The reputation of *demomakers* for technical excellence was such that some of them contributed with technical articles to specialized magazine, such as *Amstrad Cent Pour Cent* [Pic92] [Pic93a] [Pic93b]. Notably, many demomakers have been involved in the video game industry, as video game development requires strong technical skills. As of today, there is still an active demoscene on 8-bit home computers, as well as on other systems. Our study will present some popular optimization techniques used by demo programmers.

This survey is organized as follows. In section 2, we will present the chosen writing conventions. In section 3, we will give an overview of the documents studied by adopting a chronological approach, from documents published in the early days of home computing to recent materials produced by computer enthusiasts and hobbyists. Then, in section 4, we will provide a detailed description of the MOS 6502 and Zilog Z80 – the history of their development, their architecture, programming model and instruction set. Readers familiar with the MOS 6502 and Z80 can skip sections 4.1 and 4.2, and proceed directly to section 4.3. In section 5, we will study how arithmetic and mathematical features have been implemented on different 8-bit home computers. We will particularly focus on the support of floating-point calculation as it is a key feature for many fields of applications such as education, science, business and finance. We will conclude with the question of performance through the benchmarking of 8-bit home computers, and also address the issue of the trade-off between accuracy and performance. Note that for this section (and in particular sections 5.3, 5.4 and 5.5), it is advised for the reader to be comfortable with mathematics and mathematical notation. In section 6, we will present mathematical tables that have been used historically to simplify operations by hand such as multiplication and division, and will show how these techniques – which favor time over space – are used to speed up computation on 8-bit microprocessors. In the final section, we will discuss low-level optimization to improve performance at runtime, focusing on three particular techniques – loop unrolling, self-modifying code and undocumented instructions.

2 Writing conventions

To facilitate reading, the following writing conventions are used. The term 'MOS 6502' may refer to the MOS 6502 microprocessor specifically, or may also refer to the MOS 6500 microprocessor family as the MOS 6502 is the reference microprocessor for the family. Microprocessors of the MOS 6500 family share the same architecture, programming model and instruction set as the MOS 6502. They however differ in specific features – packaging, range of addressable memory, presence of an on-chip clock, clock speed (see section 4.1). Many examples for the MOS 6502 and MOS 6510 are provided, and most of these examples apply for other microprocessors of the MOS 6500 family – a notable exception being unstable undocumented instructions described in section 7.5. Similarly, we will use the term 'Z80' to refer to the different versions of the Z80 that are based on the same architecture, programming model and instruction set, but are running at a different clock speed (see table 6).

As assembly language source code and data representation may differ depending on the source, the representation of assembly language source code and data has been normalized. The use of uppercase to represent source code is a widely adopted convention for the MOS 6502 [MOS76b, p. 6] [Zak78] [App78a, pp. 135–176] [SS84]. Regarding the Z80, the representation may vary – although Zilog uses a representation in uppercase [Zil76]. For the sake of consistency, assembly language source code and

data are displayed in uppercase, with a typewriter font, e.g., `LD`, `LDA`, `JP $201A`, `$FF`.

Regarding the representation of numeric data, it also varies depending on the source. On the MOS 6502, it is customary to use the dollar symbol `$` to prefix hexadecimal data [Zak78, p. 65–66] [App78a, pp. 135–176] [Com82b, p. 217] [SS84]. However, the built-in assembler on the BBC microcomputer uses an ampersand symbol `&` to prefix hexadecimal data [Bir84, p. 3]. Regarding the Z80, representation of hexadecimal data varies – letter `H` suffix [Zil76] [Zak80, p. 108] [Sin83], ampersand symbol `&` prefix [Ams84b, app. II], hash symbol `#` prefix [Arn85, p. 14], or absence of suffix or prefix, i.e., hexadecimal is the default numeral system [LO83] [JM86]. We adopt the following conventions. First, decimal is the default numeral system. Secondly, we use the dollar symbol `$` to prefix hexadecimal data. Thirdly, we use the percent symbol `%` to prefix binary data [Zak78, p. 104]. For instance, `10`, `$0A` and `%00001010` represent the same number, i.e., ten.

Conventions are the following for the addressing modes. For the MOS 6502, the hash symbol `#` is used for *immediate addressing* [Zak78, pp. 65–66] [App78a, pp. 135–176] [Com82b, p. 218] [SS84] [CAL84, p. 432], while parentheses `()` are used for *indirect addressing* [MOS76b, p. 6] [App78a, pp. 135–176] [CAL84, p. 432]. For the Z80, parentheses `()` are used for the concept of *pointer* used by several addressing modes (*Extended addressing*, *indexed addressing*, *register indirect addressing*) [Zil76, pp. 21–22] [Zak80, p. 159]. For instance, `($01FF)` refers to the content of memory location `$01FF`.

Sources also differ regarding the representation of labels. Some sources use the colon symbol `:` as a suffix [Zil76] [Sin83] [Nie], while others omit this suffix [Zak78] [Zak80] [Com82b] [SS84] [App78a] [Bir84]. As a convention, we have decided to use the colon symbol `:` as a suffix for the labels.

Other programming languages source code is also displayed with a typewriter font. For BASIC, we also use uppercase representation, as it is a widely adopted convention for BASIC commands [App78a] [Com82c] [Ams84b] [CAL84] [VB85].

As a convention, we will use *italics* in the text when referring to certain elements of the architecture and programming model of the microprocessor, i.e., pins, registers, flags and addressing modes.

As it is customary in the demoscene [Reu10, ch. 3.4.1], most authors will be cited with their *handle* or *nickname* [Pic93a] [Pic93b] [Jac15] [Gra16] [TWW16] [Cru15] [Gol19a] [Gol19b] [Zik01] [Dre11] [Gro20], unless the document is authored with the proper name [Con20] [Gre18]. A handle will be identified with an asterisk. Note that the use of a handle is not limited to the demoscene, and is a common practice in other computer enthusiast communities.

3 Documents

Many documents related to the MOS 6502 and Z80 have been published over time. From a historical point of view, the timeline can be divided in three periods.

The first period starts with the introduction on the market of these microprocessors – the MOS 6500 microprocessor family was introduced in 1975, and the Z80 microprocessor in 1976. Technical documents – manuals, data sheet – were published by the manufacturers [MOS75a] [MOS76b] [MOS76a] [Zil76]. At the same time, many articles were also published in electronic and computer magazines to discuss the release of these microprocessors (*BYTE* [Fyl75], *EDN* [Cus75], *Microcomputer Digest* [Dig75] [Dig76b]).

The second period starts with the introduction on the market of affordable 8-bit personal computers in the late 1970s (Apple II, Atari 400/800) and early 1980s (BBC Micro, ZX Spectrum, C64, Amstrad CPC). This period is known as the *home computer revolution* as millions of 8-bit computers were sold worldwide. Comprehensive reference manuals and guides were published – whether by the manufacturers [App78a] [Com82b] [Com82c] [Ams84b] or others [CAL84] [BDH83] [GOR86] [Jam84]. Many books on specialized topics were also released, such as books on BASIC programming [App78b] [Ras78] [Ata83e] [VB85], assembly language programming [Sin83] [Bir84] [SS84], graphics [Sta82] [AJ83a] [AJ83b] [SP84] [Pie86], software protection [Mor83] [Sim+85] and system's ROM disassembly [LO83] [Rus85] [JM86]. Many articles were also published in specialized magazines, for instance to present these new computers (*BYTE* [Woz77] [App77] [Wsz83], *Personal Computer World* [Teb82] [Man83] [Kew84] [Wor85]). New magazines dedicated to certain range of 8-bit home computers were also published – *Your Spectrum* for the ZX Spectrum [Jon85], *Amstrad Cent Pour Cent* for the Amstrad CPC [Pic93a] [Pic93b], *Call-A.P.P.L.E.* and *Softalk* dedicated to the Apple personal computers [Tho85] [Wag83].The second period ended in the late 1980s and early 1990s[2], as 8-bit systems were discontinued by the manufacturers, being obsoleted by personal computers based on 16-bit, 16/32-bit and 32-bit microprocessors – such as the Atari ST and Commodore Amiga based on the 16/32-bit Motorola 68000.

The third period started subsequently. An activity was maintained in niche 8-bit home computers enthusiast and hobbyist communities such as the demoscene and the *homebrew scene* [Reu10] [Cam05]. The development of emulators of 8-bit systems for modern computers required a thorough analysis of the emulated hardware and software, and led to an in-depth study of the MOS 6502 and Z80. An illustration is the VICE emulator – whose development started in 1993 and is still ongoing – that emulates 8-bit Commodore computers [VIC]. Accurate emulation relied on a good understanding of undocumented features of the hardware – such as undocumented instructions of the MOS 6502 and derivatives (see section 7.5). More recently, with the passage of time, there has been a growing interest in the golden age of home computing. If this interest is largely driven by nostalgia and childhood memories, there is also a concern with the preservation of artifacts produced during this period – software in their original form (floppy disk, tape) deteriorate over time, hardware in good working condition is becoming rare. An example of digital preservation is the large amount of documents – manuals, data sheet, books, magazines – that have been digitized and that can be easily found online. Another example of preservation is the oral history collection program organized by the *Computer History Museum*, where in-depth interviews of computing pioneers are led, recorded and transcripted [Mus07a] [Mus07b]. There is also an emergent interest coming from academics. One notable publication is *Retrogame Archeology* written by John Aycock and published in 2016 [Ayc16]. In this book, Aycock explores technical challenges faced by games developers on constrained 8-bit systems. Different topics are covered – memory management, data compression, procedural content generation, software protection, obfuscation, optimization – with many examples taken from games developed for 8-bit systems based on the MOS 6502 and derivatives – Apple II, C64, Atari 2600. As a consequence of this growing interest, many documents have been produced recently.

[2]The commercial lifetime differs for each system. For instance, the C64 was introduced in 1982 and discontinued in 1994, while the Amstrad CPC was released in 1984 and discontinued in 1990. The Apple II series was introduced in 1977 with the Apple II, and the Apple IIc Plus – sixth and final model of the series – was discontinued in 1990.

Documents dating from the first and second period – manuals, data sheet, books, magazines – have been originally published as hard copy. If original hard copies may be difficult to obtain, it is however easy to find digitized versions – thanks to the digital preservation effort. Many documents have also been converted to plain text with OCR (Optical Character Recognition) which makes searching much easier – this is an important point given the large amount of documentation available. To give an idea of the amount of documentation, we can consider as an example *BYTE* magazine which was published monthly. The issues of June and July 1983 contain respectively 544 and 576 pages, which means that *BYTE* as a source of documentation consists of tens of thousands of pages [Sta83] [Wsz83]. As dozens of magazines have been published, along with hundreds of books, it is humanly impossible to read all the documents available, and choices had to be made. The vast majority of documents studied were written in English, but there are many relevant documents written in other languages. For instance, the most comprehensive study of the Amstrad CPC ROM was published in German [JM86]. Many relevant documents regarding the Amstrad CPC were written in French – as the Amstrad CPC was a popular home computer in France – and were studied [Pie86] [Pic93a] [Pic93b] [Zik01]. Documents have also been found in Spanish, Italian, Hungarian, Swedish, Danish and Polish – to name a few – but have not been studied. It is also necessary to mention that some documents had a very limited diffusion, and as a consequence may have not been preserved or digitized. This is the case for *fanzines*, amateur publications produced by enthusiasts and hobbyists, and usually produced to several dozens of hard copies. We cite one fanzine in this study (*Quasar CPC* [Zik01]). One critical point must be emphasized – the extreme scarcity of original source code as only a handful of original source codes have been preserved. The failure of the software industry to preserve its digital heritage can be surprising, but it results from an accumulation of difficulties of different sorts – lack of concern in legacy projects for a very fast paced industry constantly driven by profit and novelty, bankruptcies of software companies, decay over time of storage medias such as floppy disks, and legal issues linked to intellectual property to name a few. A notable example of source code preservation is *Prince of Persia* on the Apple II developed by Jordan Mechner between 1985 and 1989. The source code was successfully extracted from a 22 years old floppy disk which required "incredible expertise, perseverance, and well-maintained collection of vintage Apple hardware" and then published by its author in 2012 [Mec89]. In this study, we will rely on two original source codes – *Prince of Persia* source code on the Apple II, and *Elite* source code on Acorn computers published by one of its author Ian Bell and commented in details by Mark Moxon in 2020 [Mox20a] [Mox20b] [Mox20c] [Mox20d] (see sections 6.2.1, 6.3, 7.3 and 7.4). The scarcity of original source code contrasts dramatically with the large amount of paper documentation available.

After the end of commercial lifetime of 8-bit systems, there were still computer enthusiasts and hobbyists active, whether individually or within communities, that produced relevant documents. This activity still exists as of today. Three elements characterize this period. First, the generalized access to Internet had a critical impact on the preservation, distribution and collaboration. This can take many forms, such as forums, blogs, community websites, videos or sources code repositories. An example is the *Codebase64* community website that is dedicated to programming on the C64 and contains contributions from many hobbyists, and in particular demomakers [Jac15] [Cru15] [Gra16] [TWW16]. Second, the sophistication of today's computers is useful for certain technical tasks. The reverse engineering of software to reconstruct the source code – which was a daunting task on the original hardware – is made easier

on modern systems with the right tools [Dym20b] [Dym20a] [Mad]. Similarly, a state-of-the-art emulator such as VICE greatly facilitates development and debugging on Commodore 8-bit computers [VIC]. And last but not least, an expert knowledge has been produced over the years on some specific topics. A typical example is the issue of undocumented instructions for which undisputed reference documents have been produced over the years [You05] [Gro20].

Finally, one last point needs to be mentioned. Popular 8-bit video game systems such as the Nintendo Entertainment System and the Sega Master System are based respectively on a variant of the MOS 6502 (the Ricoh 2A03) and the Z80A. Despite their commercial success and worldwide distribution, no relevant documentation was produced during their commercial lifetime. As these video game systems were closed systems, it is not surprising. Many documents have however been produced recently, as hobbyist communities are very active on these systems. These documents have not been studied as we consider that there is enough material with 8-bit home computers.

4 Presentation of the microprocessors

4.1 Presentation of the MOS 6500 family

History

Before a formal description of the MOS 6502, it is necessary to remind the historical context of the development of the MOS 6500 microprocessor family. MOS Technology, Inc. introduced the MOS 6501 and 6502 microprocessors in 1975 to offer a low-cost alternative to the popular but rather expensive Motorola 6800 [MOS75b]. The MOS 6501 and 6502 were respectively sold $20 and $25 – the price difference being justified by the presence of an on-board clock on the 6502 – and are the first microprocessors of the MOS 6500 microprocessor family. All the microprocessors in the MOS 6500 family are software compatible within the group and are bus compatible with the Motorola 6800 [MOS75a, p. 1]. In addition, the MOS 6501 is also pin compatible with the Motorola 6800, and uses a two-phase high-level clock input, as required by the Motorola 6800 [MOS75a, p. 8]. Both MOS 6501 and 6502 are housed in a 40-pin DIP package, and their conception was largely inspired by the Motorola 6800. The cost reduction was achieved by simplifying the architecture [Cus75]: three-state control was eliminated from the address bus outputs, the second accumulator B was omitted, the 16-bit index register IX was split in two 8-bit index registers X and Y, the 16-bit stack pointer SP was reduced to an 8-bit register, and the instruction set was simplified – the size of the instruction set dropping[3] from 72 instructions to 56 instructions[4]. Despite the simplification of the architecture, several improvements were implemented in the MOS 6501 and 6502. In particular, additional addressing modes were available, allowing very efficient manipulation of arrays [Fyl75]. This link between the Motorola 6800 and the MOS 6500 microprocessor family is not accidental: in fact, the MOS Technology team headed by Chuck Peddle was composed of former Motorola employees who worked on the development of the Motorola 6800 – including Peddle himself – and MOS Technology benefited from the hindsight gained on the

[3] In particular, the instructions related to the second accumulator B and the branch instructions inspired by the DEC PDP-11 were omitted [Cus75, p. 41].

[4] The initial revision of the MOS 6501 and 6502 only proposed 55 instructions as the `ROR` instruction was missing [MOS75a, p. 6] [Zak78, p. 154] [Fyl75, p. 57]. The instruction was then introduced in 1976 [MOS76b, p. 5].

Motorola project [Cus75, p. 36]. As a consequence, Motorola sued MOS Technology, alleging that seven former employees joined MOS Technology in similar positions, helping to establish the 6500 microprocessor family [Dig75]. The lawsuit led MOS Technology to abandon the MOS 6501 which was pin compatible with the Motorola 6800 [Dig76a]. The MOS 6502 became officially the reference for the MOS 6500 family.

One of the first known use of the MOS 6502 is the Apple Computer – known today as the Apple I – designed by Steve Wozniak and introduced in 1976 [App76]. Released in 1977, the Commodore PET and the Apple II are also based on the MOS 6502 [App77] [Woz77] [BYT77] [WC78]. Several versions of the MOS 6502 were released later on. While the original MOS 6502 was running at 1 MHz, faster models were then introduced on the market – the MOS 6502A, 6502B and 6502C (see table 1). Popular 8-bit systems are based on these models, such as the BBC Electron based on the MOS 6502A [Man83], and the Atari 400 and 800 based on the MOS 6502B [Ata81, p. 20].

Model	**Clock speed**
6502	1 MHz
6502A	2 MHz
6502B	3 MHz
6502C	4 MHz

Table 1: MOS 6502 versions [Com85, p. 1].

Other models of the MOS 6500 processor family were released later on, proposing different features, such as the range of addressable memory, the presence of an on-chip clock, and differences in packaging (see table 2). All these models remain software compatible with the MOS 6502. The MOS 6507 released in 1977 is a cost-reduced variant of the MOS 6502 packaged in a 28-pin DIP (see figure 1). The reduction in pin count was achieved by reducing the address bus from 16 bits to 13 bits – limiting the addressable memory to 8 KB – and removing certain features – such as the interrupt related pins $\overline{IRQ}$ and $\overline{NMI}$ [Com85, p. 11]. The Atari 2600 is based on the MOS 6507 [Ata83b, p. 1-4]. Another model is the MOS 6510 released in 1982 which offers a 6-bit or 8-bit[5] bidirectional I/O[6] port programmable bit-by-bit as well as a three-state address bus that allows Direct Memory Access (DMA) and multiprocessor systems sharing a common memory [Com82a, p. 1]. Unlike the MOS 6502, the MOS 6510 requires an external two-phase clock [Com82a, p. 7]. The MOS 6510 is notably used in the C64 released in 1982 [Mor82, p. 61].

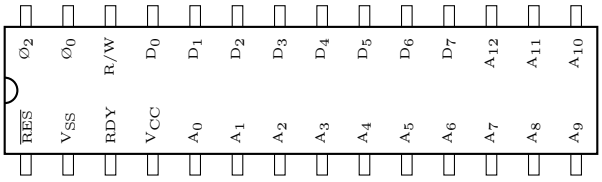

Figure 1: MOS 6507 28-pin DIP package [Com85, p. 11].

[5]Different I/O pin arrangements exist for the MOS 6510. [Com82a] mentions 8 I/O pins, while [Com82b, p. 403] states that a 6 I/O pins MOS 6510 is used in the C64.

[6]Input/output.

Model	Package	Addressable memory	Clock
6501	40-pin DIP	64 KB	External
6502	40-pin DIP	64 KB	On-chip
6503	28-pin DIP	4 KB	On-chip
6504	28-pin DIP	8 KB	On-chip
6505	28-pin DIP	4 KB	On-chip
6506	28-pin DIP	4 KB	On-chip
6507	28-pin DIP	8 KB	On-chip
6510	40-pin DIP	64 KB	External
6512	40-pin DIP	64 KB	External
6513	28-pin DIP	4 KB	External
6514	28-pin DIP	8 KB	External
6515	28-pin DIP	4 KB	External

Table 2: The MOS 6500 microprocessor family [Com82a] [Com85].

Many variants of the MOS 6502 exist. One variant is the *6502C* used by Atari in the Atari XL models and documented as such by Atari[7] [Ata83d, pp. 4–11]. The *Atari 6502C* should not be confused with the MOS 6502C, as they differ in their respective pin arrangement (see table 3), and the Atari 6502C runs at a lower clock speed than the MOS 6502C. Another variant is the Ricoh 2A03, used in the Nintendo Entertainment System[8] (NES) [Dis04, p. 9]. The main difference between the Ricoh 2A03 and the MOS 6502 is the lack of decimal mode support.

Pin	MOS 6502C	Atari 6502C
7	$SYNC$	$N.C.$
34	R/W	$N.C.$
35	$N.C.$	$\overline{HALT}$
36	$N.C.$	R/W

Table 3: Pin differences between the MOS 6502C and the modified Atari 6502C [Ata83c, p. 1-5].

Architecture and programming model

The MOS 6502 is packaged as a 40-pin DIP package (see figure 2) and communicates with the rest of the computer with three buses [Zak78, ch. II] [SS84, pp. 16–17] [Com85]. The 16-bit *address bus* is responsible for the selection of a particular memory location. The address bus is split in two 8-bit low-order byte and high-order byte address buses ABL and ABH. Pins A_0 to A_7 are connected to ABL, while pins A_8 to A_{15} are connected to ABH. The addressable range is 64 KB, i.e., 65 536 bytes. Pins D_0 to D_7 are connected to the 8-bit bidirectional *data bus* which is responsible for sending or receiving data to and from the selected memory location. Note that pins A_i and D_i

[7] This variant is also documented as the *modified 6502* in earlier Atari documentation [Ata83c, p. 1-5] [Ata83a, p. 1-6].

[8] Originally released in Japan in 1983 as the Famicom.

may also be named *ABi* and *DBi*, depending on the documentation [MOS76b, pp. 2, 9]. Pins *R/W*, $\overline{IRQ}$, $\overline{NMI}$, *SYNC*, *RDY*, *S.O.*, $\overline{RES}$, $\emptyset_0$, $\emptyset_1$ and $\emptyset_2$ are connected to the *control bus* and are used to exchange control signals. Pins V_{SS} and V_{CC} are used for power. Finally, pins 5, 35 and 36 are not connected[9].

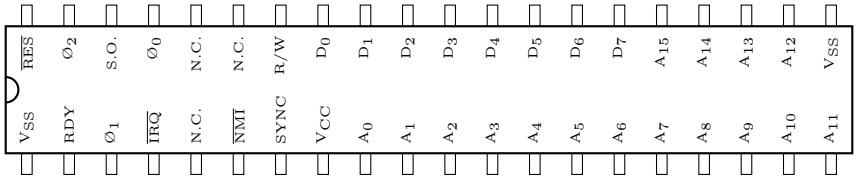

Figure 2: MOS 6502 40-pin DIP package [Com85, p. 10].

The 64 KB memory addressable by the MOS 6502 is partitioned into 256 pages of 256 bytes[10] [Zak78, p. 37]. The two first pages have a specific role. Page 0 is located in the `$0000-$00FF` address range and is called the *zero page*. The zero page offers significant speed advantages and should be used to store frequently accessed data [MOS76b, p. 5]. According to Rodnay Zaks, the zero page should be considered by the programmer as a set of 256 "working registers" [Zak78, p. 179]. All instructions addressing the zero page require only 2 bytes – 1 byte for the opcode and 1 byte for the index within the zero page [MOS76b, p. 4]. Most of these instructions execute in 3 cycles. Careful use of the zero page[11] is critical for performance. Regarding page 1, it is located in the `$0100-$01FF` address range and is used for the stack. From a performance point of view, it is important for the programmer to be aware of the organization of the memory, as each time a page boundary is crossed, the instruction usually requires an extra cycle for its execution [Zak78, p. 37] [MOS76b, p. 6].

The MOS 6502 has a "classical accumulator-based design" [Zak78, p. 29]. The 8-bit *accumulator*, or *A* register, is directly connected to the *Arithmetic Logic Unit* (ALU) as it plays a central role for arithmetic and logical operations. First, the accumulator is used to store one of the two operands prior to an arithmetic (`ADC`, `SBC`) or logic (`AND`, `EOR`, `ORA`) operation. After the operation is performed, the result is stored in the accumulator. This design leads to short instructions as the use of the accumulator is implied. However, the fact that only one accumulator is present tends to be restrictive in organizing efficient machine code [SS84, p. 21].

The MOS 6502 has an 8-bit *processor status register* – or *P* register – that contains the following flags: *C* (Carry), *Z* (Zero), *I* (IRQ disable), *D* (Decimal mode), *B* (BRK command), *V* (Overflow) and *N* (Negative) [MOS76b, p. 6]. These flags are used as a condition by some instructions. For example, the `BEQ` instruction branches only if *Z* flag is set [Zak78, p. 106]. These flags are also affected by some instructions. For instance, the `DEX` instruction sets the *Z* flag when zero is reached [Zak78, p. 126].

[9] *N.C.*: No connection [MOS75a, p. 9].

[10] In relation to the two 8-bit address buses [MOS76a, pp. 6–8].

[11] The system and its applications use extensively the zero page, and by default only a portion of the zero page is available to the end user. On the Apple II, Apple DOS 3.2, Applesoft II Basic and Integer BASIC store data in the zero page [App79, pp. 74–75]. On the BBC Micro, the zero page is also used by the system [BDH83, pp. 267–272] – BBC Basic allows only the `$70-$8F` range to the end user. On the Atari home computers, the `$00-$7F` range is used by the system, and the `$D4-$FF` range is used by the floating-point package [Ata83e, p. 92].

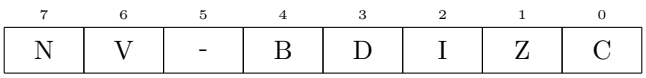

Figure 3: MOS 6502 flags [MOS76b, p. 6].

The MOS 6502 has two 8-bit index registers X and Y. X and Y registers have three main uses [SS84, p. 21]. First, to transfer data quickly from and to the accumulator A (`TAX`, `TXA`, `TAY` and `TYA` instructions [Zak78, pp. 166, 167, 169, 171]). Second, as up-counters and down-counters for loops (`INX`, `DEX`, `INY` and `DEY` instructions [Zak78, pp. 126, 127, 132, 133]). Finally, as an index in the indexed addressing modes supported by the MOS 6502. Index registers allow very efficient processing of arrays [Fyl75, p. 58]. The source code presented in figure 4 illustrates how index register X and indexed addressing may be used to copy an array of `COUNT` bytes from `SOURCE+1` memory location to `DEST+1` memory location.

```
        LDX  COUNT
LOOP:   LDA  SOURCE,X
        STA  DEST,X
        DEX
        BNE  LOOP
```

Figure 4: Copy `COUNT` bytes from `SOURCE+1` to `DEST+1`.

The 8-bit register S is the *stack pointer* [Zak78, pp. 35–37]. The stack is a classic LIFO (Last In, First Out) data structure that allows two operations: *push* to deposit one element on top of the stack, and *pull*[12] to remove one element from the top of the stack. As previously stated, the stack is stored in page 1 located in the `$0100-$01FF` address range. The stack pointer is initialized with `$FF`, i.e., the stack starts at `$01FF`. The stack pointer is updated after each push or pull operation. Instructions are available to push and pull the accumulator (`PHA` and `PLA` instructions [Zak78, pp. 148, 150]) or the process status register (`PHP` and `PLP` instructions [Zak78, pp. 149, 151]). The stack has three main uses. First, to implement subroutines [Zak78, pp. 76–83]. `JSR` and `RTS` instructions use the stack to save and restore the program counter [Zak78, pp. 136, 157]. Second, to handle interrupts [Zak78, pp. 228–232]. In this case, the stack is used to save the program counter and the processor status register, that can be then restored with an `RTI` instruction [Zak78, p. 156]. Finally, the stack may be used to store data temporarily.

The 16-bit register PC is the *program counter* and contains the address of the next instruction to be executed. It is split in two 8-bit registers, PCL and PCH [Zak78, p. 31].

Instruction set

The MOS 6502 instruction set is rather limited and consists of 56 instructions [MOS76b, p. 1]. One way of representing the instruction set is to organize it into different functional categories. A proposed representation by Zaks is a breakdown into the following

[12] Also known as *pop*.

categories: data transfer, data processing, test and branch, and control [Zak78, pp. 87–96]. Data transfer instructions consist of loading (`LDA`, `LDX` and `LDY`), storing (`STA`, `STX` and `STY`), inter-registers transfers (`TAX`, `TAY`, `TSX`, `TXA`, `TXS` and `TYA`) and push/pull operations for the stack (`PHA`, `PHP`, `PLA` and `PLP`). Data processing instructions may be broken in four subcategories: arithmetic operations (`ADC` and `SBC`), logical operations (`AND`, `ORA` and `EOR`), shift and rotate (`ASL`, `LSR`, `ROL` and `ROR`), increment and decrement (`INC`, `INX`, `INY`, `DEC`, `DEX` and `DEY`). Test and branch instructions include conditional branches, unconditional jumps and returns (`JMP`, `JSR`, `RTS` and `RTI`), bit testing (`BIT`) and comparisons instructions (`CMP`, `CPX` and `CPY`). Conditional branches test a specific flag (`BMI` and `BPL` the N flag, `BCC` and `BCS` the C flag, `BEQ` and `BNE` the Z flag, `BVS` and `BVC` the V flag) and branch within the same instruction, where the branch is a signed two's complement 8-bit displacement relative to the first instruction following the branch instruction. Control instructions consist of instructions to set or reset flags (`SEC` and `CLC` the C flag, `SED` and `CLD` the D flag, `SEI` and `CLI` the I flag, `CLV` the V flag), and other instructions (`BRK` and `NOP`).

An instruction of the MOS 6502 consists of an 8-bit opcode that may be followed by an 8-bit or 16-bit operand [MOS76b, p. 6]. An 8-bit operand may be an 8-bit data, address or displacement, whereas a 16-bit operand may only be a 16-bit address. As a consequence, the size of an instruction is either one, two or three bytes. For example, the instruction `INX` is represented with a single byte `$E8`, the instruction `LDA #$03` is encoded `$A9 $03`, and the instruction `JMP $3F00` is encoded `$4C $00 $3F` – as the MOS 6502 is *little endian*, the low-order byte of the 16-bit address is stored before the high-order byte. The size of the instruction is directly related to the addressing mode, as shown in table 4.

Addressing mode	**Size**	**Operand type**
Accumulator	1	-
Implied	1	-
Immediate	2	8-bit data
(Indirect, X)	2	8-bit zero page address
(Indirect), Y	2	8-bit zero page address
Relative	2	8-bit displacement
Zero page	2	8-bit zero page address
Zero page, X	2	8-bit zero page address
Zero page, Y	2	8-bit zero page address
Absolute	3	16-bit address
Absolute, X	3	16-bit address
Absolute, Y	3	16-bit address
Absolute indirect	3	16-bit address

Table 4: Addressing mode, instruction size (in bytes) and operand type.

The MOS 6502 supports many different addressing modes [MOS76b, p. 5]. An addressing mode refers to the specification – within the opcode of the instruction – of the location of the operand on which the instruction will operate [Zak78, p. 172]. First, there are two addressing modes without operand, the *Accumulator* and *Implied* addressing modes. In *Accumulator* addressing, the operation is performed on the accumulator, while in *Implied* addressing, the location of the operand is implicitly

stated in the opcode. The instruction `INX` is an example of *Implied* addressing, as the opcode implies an operation on the index register X [Zak78, p. 132]. Secondly, there are several addressing modes where there is an 8-bit operand. In *Immediate* addressing, the operand is contained in the second byte of the instruction, and no memory access is required. The instruction `LDA #$03` is an example of *Immediate* addressing, where the value `$03` is loaded into the accumulator [Zak78, pp. 137–138]. Three addressing modes are relative to the zero page: *Zero page*[13], *Zero page, X* and *Zero page, Y* addressing modes. For these addressing modes, the operand is an 8-bit address within the zero page. In *Zero page, X* and *Zero page, Y* indexed addressing modes, the content of the associated index register – X and Y respectively – is added to the 8-bit address. The instruction `LDA $2B,X` is an example of *Zero page, X* addressing where the accumulator is loaded with the byte located at the zero page address resulting from the addition of X to `$2B`[14]. *Relative* addressing is only used with branch instructions and establishes a destination for the branch. In this addressing mode, the 8-bit operand is a signed two's complement displacement ranging from -128 to $+127$, and the displacement is relative to the address of the first instruction following the branch instruction. *Relative* addressing improves performance as the branch instruction is only 2 bytes long. It also allows code to be relocatable. In *(Indirect, X)* and *(Indirect), Y* addressing modes, the operand is an 8-bit address within the zero page. In *(Indirect, X)* addressing, a zero page address is computed by adding the register X to the operand[14]. The resulting 8-bit address points to a 16-bit pointer stored in the zero page, the 16-bit pointer is then used to access the data. In *(Indirect), Y* addressing, the operand is used to retrieve a 16-bit pointer stored in the zero page. Register Y is then added as a displacement to the 16-bit pointer, and the resulting 16-bit pointer is used to access the data. Finally, the remaining addressing modes – *Absolute*, *Absolute, X*, *Absolute, Y* and *Absolute Indirect* – require a 16-bit address operand and instructions are 3 bytes long. In *Absolute* addressing, the operand is used directly. For example, the instruction `LDA $03FF` loads into the accumulator the byte at memory location `$03FF`, while the instruction `JMP $1FE0` loads the address `$1FE0` in the program counter, causing a jump to occur in the program sequence. In *Absolute, X*, and *Absolute, Y* indexed addressing modes, the content of the associated index register – X and Y respectively – is added to the 16-bit address. The instruction `LDA $03FF,X` is an example of *Absolute, X* addressing where the accumulator is loaded with the byte located at the 16-bit address resulting from the addition of X to `$03FF`. In *Absolute indirect* addressing, the 16-bit address operand points to a 16-bit pointer in memory. *Absolute indirect* addressing is only supported by the `JMP` instruction [MOS76b, p. 6]. For example, the instruction `JMP ($2000)` jumps to the 16-bit address stored at `$2000` memory location. Most instructions support several addressing modes, bringing the initial 56 instructions to a total of 166 opcodes [SS84, p. 34]. As an illustration, table 5 presents the different addressing modes supported by the `LDA` instruction, with the associated instruction size and number of clock cycles[15] required to execute the instruction.

We will conclude our overview of the MOS 6502 with a presentation of the instruction execution cycle. Each instruction is executed in a sequence of basic operations [Zak78, pp. 32–35] [SS84, pp. 25–30]. First of all, the address stored in the program

[13]The *Zero page* addressing is equivalent to the *Direct* addressing on the Motorola 6800 [Mot84a, p. 24].

[14]Crossing of the zero page is not allowed. If the addition overflows, the carry is not added to the high-order byte and the address wraps around the zero page [MOS76b, p. 5].

[15]Extra clock cycles may be required under certain circumstances.

Addressing mode	Opcode	Size	Cycles
Immediate	`$A9`	2	2
Zero page	`$A5`	2	3
Zero page, X	`$B5`	2	4
Absolute	`$AD`	3	4
Absolute, X	`$BD`	3	4 or 5
Absolute, Y	`$B9`	3	4 or 5
(Indirect, X)	`$A1`	2	6
(Indirect), Y	`$B1`	2	5 or 6

Table 5: Opcodes of `LDA` instruction.

counter is placed on the address bus, the program counter is incremented, the selected memory location is read and transferred to the *instruction register* (*IR*). The fetched byte is the opcode of the instruction. The opcode is decoded by the *instruction decode unit*[16] [MOS76b, p. 2] and two distinct situations may happen. In the first situation, it is not necessary to fetch additional data and the instruction may be executed immediately. For example, the instruction `INX` (opcode `$E8`) does not require to fetch an operand as it is *Implied* addressing. In the second situation, it is necessary to fetch one or two additional bytes, depending on the addressing mode. For instance, the instruction `JMP` in *Absolute* addressing (opcode `$4C`) requires to fetch two additional bytes – the low-order byte and the high-order byte of the 16-bit address. These extra fetching operations impact the number of clock cycles required to execute the instruction. The fastest instructions on the MOS 6502 are executed in 2 clock cycles, whereas the slowest instructions are executed in at least 7 clock cycles [MOS76b, p. 6]. Extra clock cycles penalties may happen for certain instructions: one extra cycle penalty if a page boundary is crossed, one extra cycle penalty if a branch is taken in the same page, and two extra cycles penalty if a branch is taken in another page.

4.2 Presentation of the Zilog Z80

History

Zilog, Inc. was founded in 1974 by Federico Faggin and Ralph Ungermann, two former Intel employees [Mus07b, pp. 1–2]. Masatoshi Shima, another Intel employee, joined the company later on. Since they were all from Intel, they had an extensive experience in the design of microprocessors. In particular, Faggin and Shima were the conceptors of the popular Intel 8080[17]. Faggin started the preliminary conception of the Z80 in late 1974 [Mus07b, p. 4]. In order to take a significant part of the Intel 8080 market share, the Z80 was designed to be software compatible with the Intel 8080[18] [Zil76, p. 1]. As a consequence, the internal architecture of the Z80 presents similarities with the Intel 8080: both have a 16-bit program counter (*PC*), a 16-bit stack pointer (*SP*), an

[16] Also called *control-unit* [Zak78, p. 33] or *decode matrix* [SS84, p. 30].

[17] Note that the initial version of the Intel 8080 had an issue in its internal ground line [Mus07a, p. 14]. The Intel 8080A was then released to correct this flaw. Other versions were also released, such as the 8080A-1 and 8080A-2 running at different clock speeds [Int75b, ch. 5]. To be consistent with Intel documentation [Int75b] [Int75a], the term 'Intel 8080' may refer to the Intel 8080 microprocessor family.

[18] The Z80 is however not PIN compatible with the Intel 8080 [Has76, pp. 34–35].

accumulator (A), a flag register (F) and six general purpose 8-bit registers that can be used as single 8-bit registers (B, C, D, E, H and L) or as 16-bit register pairs (BC, DE and HL) [Has76, p. 35] [Int75b, ch. 2] [Zil76, ch. 2.1]. The Z80 also supports the Intel 8080 addressing modes – *Register*, *Register indirect*, *Direct*[19] and *Immediate* addressing modes [Int75b, p. 4-2] [Zil76, ch. 5.2]. In order to also compete with the Motorola 6800, the Z80 was designed to provide indexing capabilities. The development of the Z80 officially started in February 1975 and the project was completed in March 1976 [Mus07b, p. 6]. The Z80 was finally introduced on the market in summer 1976 [Dig76b]. The Z80A – a version of the Z80 operating at 4 MHz – was released later on [Zil77, p. 1].

The Z80 offers many improvements over the Intel 8080 [Has76]. The Z80 instruction set consists of 158 different instructions, while the Intel 8080 has only 78 instructions [Zil76, p. 19] [Int75a, p. V]. This increase of 80 instructions over the Intel 8080 instruction set results from additional addressing modes supported by the Z80 – such as *Indexed*, *Relative* and *Bit* addressing – and new instruction types. For instance, the Z80 supports block transfer instructions to move blocks of data around. In one instruction, up to 64 KB of data can be moved anywhere in memory (`LDDR` and `LDIR` instructions [Zil76, pp. 26–28]). Similarly, the Z80 proposes block search instructions (`CPDR` and `CPIR` instructions [Zil76, pp. 26–28]). Bit manipulation instructions are also supported, to test, set or reset a particular bit (`BIT`, `SET` and `RES` instructions [Zil76, p. 31]). Other additional instructions types are supported by the Z80. Regarding the registers, the Z80 has an alternate register set consisting of an alternate accumulator (A'), an alternate flag register (F') and six alternate general purpose 8-bit registers that can be used as single 8-bit registers (B', C', D', E', H' and L') or as 16-bit register pairs (BC', DE' and HL'). Main and alternate register sets can be switched for fast interrupt processing [Zil76, p. 5]. Besides, the Z80 provides 16-bit index registers – IX and IY – for the *Indexed* addressing modes [Zil76, pp. 4, 21–22]. The Z80 also offers additional capabilities regarding interrupt handling: a dedicated *interrupt page address register* (I), support of software maskable interrupt ($\overline{INT}$ pin) and non maskable interrupt ($\overline{NMI}$ pin), and three possible interrupt modes (*Mode 0*[20], *Mode 1* and *Mode 2*) [Zil76, pp. 55–57].

A notable early use of the Z80 is the TRS-80 released in 1977 by Radio Shack [Mor77]. Two popular home computers series introduced on the market in the early 1980s are based on the Z80A – the ZX Spectrum and the Amstrad CPC home computer series [Teb82] [Kew84]. The Z80 is also used as a second processor to run CP/M operating system and its large library of business software. For instance, the BBC Micro can be equipped with the Z80 Second Processor that features a Z80B running at 6 MHz [Web84]. Released in 1985, the Commodore 128 (C128) has two main processors, a MOS 8502 software compatible with the MOS 6502, and a Z80A to run CP/M [Wor85, p. 141]. The Z80A is also used in video game systems. For instance, the Sega Master System and the Sega Game Gear use a Z80A [Seg, p. 13] [Seg92, p. 5], while the Sega Mega Drive[21] based on a Motorola 68000 uses a Z80A for sound processing and to preserve backward compatibility with previous systems [Seg94, pp. 4, 14] [SW14, pp. 304–305]. Different models of the Z80 have been released over time. These models propose the same features, they only vary in clock speed (see table 6).

[19] *Direct* addressing is known as *Extended* addressing on the Z80.

[20] This interrupt mode is identical to the Intel 8080 interrupt response mode [Zil76, p. 56].

[21] Known as the Sega Genesis in North America.

Model	Clock speed
Z80	2.5 MHz
Z80A	4 MHz
Z80B	6 MHz
Z80H	8 MHz

Table 6: Z80 versions [Zil84, p. 5].

Architecture and programming model

The Z80 is packaged as a 40-pin DIP package (see figure 5). Pins A_0 to A_{15} constitute a 16-bit unidirectional address bus. The address bus is used to select the address for memory and I/O device data exchanges. The total memory addressable range is 64 KB. In the context of I/O addressing, pins A_0 to A_7 are used to select up to 256 inputs, while pins A_8 to A_{15} are used to select up to 256 outputs. Pins D_0 to D_7 constitute an 8-bit bidirectional data bus. The data bus is used for data exchanges with the memory and devices. Pins related to control signals can be divided into four groups [Zak80, pp. 91–93]: the clock input (Φ), bus-control signals ($\overline{BUSRQ}$, $\overline{BUSAK}$), Z80 control signals ($\overline{INT}$, $\overline{NMI}$, $\overline{WAIT}$, $\overline{HALT}$, $\overline{RESET}$), memory and I/O control signals ($\overline{MREQ}$, $\overline{M_1}$, $\overline{IORQ}$, $\overline{RD}$, $\overline{WR}$, $\overline{RFSH}$). Finally, pins *GND* and *+5V* are used to power the Z80.

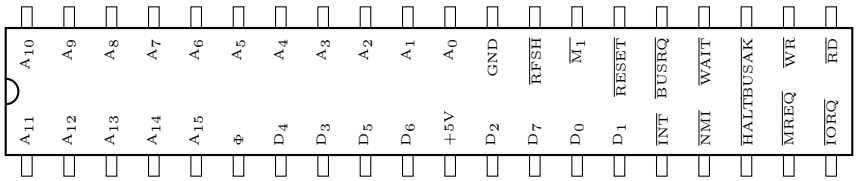

Figure 5: Z80 40-pin DIP package [Zil76, pp. 7–9].

The Z80 includes two 16-bit register pairs *AF* and *AF'*. The first register pair *AF* consists of the accumulator *A* and the flag register *F*, while the alternate register pair *AF'* consists of the alternate accumulator *A'* and the alternate flag register *F'* [Zil76, p. 4] [Zak80, pp. 61–62]. *AF* may be used as a working register pair, while *AF'* is only used for storage. The content of *AF* and *AF'* may be exchanged via the `EX AF,AF'` instruction [Zil76, p. 46] [Zak80, p. 248]. The accumulator holds the results of 8-bit arithmetic (addition, subtraction) and logical operations (and, or, exclusive or). The flag register (see figure 6) contains six bits of information which are set or reset by various Z80 operations [Zil76, pp. 39–40] [Zak80, pp. 174–179]. Four of these bits are testable and may be used as conditions for certain instructions. The flags are the following: *C* (Carry), *N* (Subtract), *P/V* (Parity/Overflow), *H* (Half-Carry), *Z* (Zero), and *S* (Sign). For example, `DEC C` sets the *Z* flag when zero is reached [Zak80, pp. 238–239], while `JP Z,nn` instruction jumps to the specified address if the *Z* flag is set [Zil76, p. 52].

The Z80 has two sets of general purpose registers, each set containing six 8-bit registers that may be used individually (see table 7) or as 16-bit register pairs (see table 8). The first set consists of *BC*, *DE* and *HL* registers, while the alternate set consists of *BC'*, *DE'* and *HL'* registers. As previously, *BC*, *DE* and *HL* may be used

7	6	5	4	3	2	1	0
S	Z	-	H	-	P/V	N	C

Figure 6: Z80 flags [Zil76, pp. 39–40].

as working registers, while *BC'*, *DE'* and *HL'* are only used for storage. The content of the two sets of registers may be exchanged via the `EXX` instruction [Zil76, p. 46] [Zak80, p. 256]. The general purpose registers may play distinct roles depending on the context. In the context of block transfer instructions (`LDD`, `LDI`, `LDDR` and `LDIR`) and block search instructions (`CPD`, `CPI`, `CPDR` and `CPIR`), *BC* is used as a 16-bit byte counter, *HL* as a 16-bit source memory location and *DE* as a 16-bit destination memory location [Zil76, pp. 26–28] [Zak80, pp. 163–164]. The source code presented in figure 7 illustrates how `LDIR` (*LoaD, Increment and Repeat*) is used to copy `COUNT` bytes from `SOURCE` memory location to `DEST` memory location.

```
LD BC,COUNT
LD HL,SOURCE
LD DE,DEST
LDIR
```

Figure 7: Copy `COUNT` bytes from `SOURCE` to `DEST`.

Main registers		**Alternate registers**	
B	C	B'	C'
D	E	D'	E'
H	L	H'	L'

Table 7: Z80 main and alternate 8-bit general purpose registers [Zil76, p. 5].

The Z80 has also two 16-bit index registers *IX* and *IY* [Zil76, p. 4] [Zak80, p. 53]. An index register holds a 16-bit base address used in *Indexed* addressing mode. As stated previously, indexing capabilities in the Z80 design were inspired[22] by the Motorola 6800 [Mus07a, p. 5]. An additional signed two's complement byte is included in indexed instructions to specify a relative displacement. Indexing simplifies access to any sequential block of data. For instance, the instruction `LD E,(IX+8)` loads register *E* with the byte pointed by `IX+8` [Zil76, p. 23].

The 16-bit register *SP* is the stack pointer [Zil76, p. 3] [Zak80, pp. 53–54]. The stack pointer holds the memory address of the current top of the stack. Data may be *pushed* onto the stack from specific registers and *popped* off of the stack into specific registers with the `PUSH` and `POP` instructions [Zak80, pp. 373–384]. Like the MOS 6502, the stack has three main uses. First, to implement subroutines, as the stack is required to store and restore the program counter [Zak80, pp. 142–150]. Second, to

[22] In this regard, the Motorola 6800 uses a 16-bit index register also named *IX* [Mot84a, p. 12].

Main registers	Alternate registers
BC	BC'
DE	DE'
HL	HL'

Table 8: Z80 main and alternate 16-bit general purpose register pairs [Zil76, p. 5].

handle interrupts [Zak80, pp. 495–510]. And third, the stack is convenient to store data temporarily.

The Z80 has other registers that will be described briefly (see table 9). The program counter (PC) is a 16-bit register that holds the address of the next instruction to be executed [Zil76, p. 3]. A more detail analysis of its role will be presented in the next section. The *interrupt page address register* (I) is an 8-bit register that holds the high-order byte of a memory location containing routines called in response to an interrupt [Zil76, p. 3]. The *memory refresh register* (R) is a 7-bit counter automatically incremented each time an instruction is fetched. This register is used to refresh dynamic memories automatically [Zil76, p. 4].

I	R
IX	
IY	
SP	
PC	

Table 9: Z80 special purpose registers [Zil76, p. 4].

Instruction set

The Z80 can execute 158 different instructions, including the 78 instructions[23] of the Intel 8080 as the Z80 is fully software compatible with the 8080 [Zil76, pp. 23–37]. As proposed by Zaks and similarly to the MOS 6502, the instruction set may be broken down into different functional categories[24]: data transfers, data processing, test and jump, I/O, and control [Zak80, pp. 158–188]. Data transfers instructions may be broken down in several subcategories: 8-bit data transfers, 16-bit data transfers, interregisters exchanges (`EX` and `EXX`), stack operations (`PUSH` and `POP`), and block transfers (`LDD`, `LDDR`, `LDI` and `LDIR`). The load group of instructions `LD` which moves 8-bit and 16-bit data between registers or between registers and memory plays a central role in the instruction set [Zak78, fig. 4.2, 4.3]. Data processing instructions consist of several subcategories: arithmetic operations (`ADC`, `ADD`, `SBC`, `SUB`, and special instructions `DAA`, `CPL` and `NEG`), bit manipulation (`TEST`, `RES`, `SET`, `CCF` and `SCF`), increment and decrement (`INC` and `DEC`), logical operations (`AND`, `OR`, `XOR` and `CP`), shift and rotate (`RLC`, `RRC`, `RL`, `RR`, `SLA`, `SRA`, `SRL`, `RLD` and `RRD`), and block search (`CPI`, `CPIR`, `CPD` and

[23] Refer to [Zak80, pp.615–616] for a mapping between the Z80 and the Intel 8080 instruction sets.

[24] Other classifications exist though, such as the one proposed by Zilog [Zil76, ch. 5.3].

CPDR). Test and jump instructions include instructions for relative jumps (JR), absolute jumps (JP), subroutine calls (CALL) and returns (RET) that may be conditional or unconditional [Zak80, fig. 4.18]. This category also includes other specialized instructions (DJNZ, RETI, RETN and RST). Conditional jumps and calls may test one flag among Z, C, P/V and S flags. I/O instructions allow to communicate with I/O devices. Basic I/O instructions are provided (IN and OUT), as well as block I/O transfers instructions (INI, INIR, IND, INDR, OUTI, OTIR, OUTD and OTDR). Control instructions consist of various instructions related to synchronization and interrupts (NOP, HALT, EI, DI, IM0, IM1 and IM2) [Zak80, fig. 4.22].

An instruction of the Z80 consists of an 8-bit or 16-bit opcode, that may be followed by one or two operands [Zil76, ch. 5.3]. The fact that an opcode may be 16-bit is related to the software compatibility between the Z80 and the Intel 8080: the additional instructions on the Z80 were implemented first by using the few available 8-bit opcodes on the 8080, and secondly by extending existing 8-bit opcodes on the 8080 with an additional byte [Zak80, p. 158]. Depending on the instruction, an operand may be 8-bit or 16-bit. As the Z80 is little endian, the low-order byte of a 16-bit address or data is stored before the high-order byte. Table 10 presents several examples of instructions encoding.

Instruction	**Opcode**	**First operand**	**Second operand**
LD C,B	$48		
LD H,$36	$26	$36	
LD D,($0659)	$11	$59 $06	
LD A,($6F32)	$3A	$32 $6F	
LD E,(IX+8)	$DD $5F	$08	
LD (IX-15),$21	$DD $36	$F1	$21

Table 10: Examples of Z80 instructions encoding [Zil76, ch. 5.3].

The Z80 supports 10 addressing modes: *Immediate*, *Immediate extended*, *Modified page zero*, *Relative*, *Extended*, *Indexed*, *Register*, *Implied*, *Register indirect* and *Bit* addressing modes [Zil76, ch. 5.2]. In *Immediate* addressing, an 8-bit operand follows the 8-bit or 16-bit opcode, while in *Immediate extended* addressing, a 16-bit operand follows the 8-bit or 16-bit opcode. *Modified page zero* addressing is specific to the RST p instruction [Zak80, pp. 418–419]. The eight different 8-bit opcodes of this instruction ($C7, $CF, $D7, $DF, $E7, $EF, $F7 and $FF) allow the program to jump to eight memory locations in page zero ($0000, $0008, $0010, $0018, $0020, $0028, $0030 and $0038). However, the Z80 does not support addressing modes similar to the MOS 6502 zero page addressing modes (see table 4). In the case of *Relative* addressing, the 8-bit operand following the 8-bit opcode specifies a displacement to which the program may jump. The displacement is a signed two's complement number ranging from -128 to $+127$, and is relative to the address following the jump instruction. The two major advantages of this addressing mode are that it improves performance and it allows code to be relocatable [Zak80, p. 441]. In *Extended* addressing[25], the 16-bit operand is a memory location to which the program may jump or where data is located. *Indexed* addressing was mentioned previously and is related to index registers [Zil76, p. 4]

[25] Also known as *Absolute* addressing [Zak80, p. 446].

[Zak80, p. 53]. In this type of addressing, the 8-bit operand following the 16-bit opcode is a signed two's complement displacement added to an index register to form a pointer to a memory location. Depending on the instruction, the 8-bit displacement operand may be followed by an additional 8-bit operand. *Indexed* addressing allows for relocatable code and is used extensively to process blocks of data, tables and lists [Zil76, pp. 21–22] [Zak80, pp. 447–448]. *Register* addressing refers to the fact that for some instructions the opcode contains bits which specifies which registers are used for an operation. An illustration of register addressing is the instruction `LD r,r'` where a register is specified with three bits – for example, `%000` designates register *B* while `%001` specifies register *C* [Zak80, pp. 297–298]. In *Implied* addressing, the opcode implies that one or more registers are used. For instance, the `EXX` instruction implies that *BC*, *DE* and *HL* registers are exchanged with *BC'*, *DE'* and *HL'* registers [Zak80, p. 256]. In *Register indirect* addressing, the 8-bit or 16-bit opcode specifies a 16-bit register pair (*BC*, *DE* or *HL*) used as a pointer to a location in memory. As the Z80 is little endian, if the register points to a 16-bit data, then the pointed memory location holds the low-order byte while the next memory location contains the high-order byte. Finally, *Bit* addressing may be used in conjunction with *Register*, *Register indirect* or *Indexed* addressing modes to test, set or reset a specific bit of the 8-bit operand [Zil76, p. 33]. Table 11 presents examples of instructions in different addressing modes.

Addressing mode	**Instruction**
Immediate	`LD A,0`
Immediate extended	`LD BC,$1000`
Modified page zero	`RST $08`
Relative	`JR $E0`
Extended	`JP $201A`
	`LD HL,($4A59)`
Indexed	`LD (IY+2),A`
Register	`LD C,B`
Implied	`EXX`
Register indirect	`INC (HL)`
Bit	`SET 5,B`

Table 11: Examples of Z80 instructions in different addressing modes.

To conclude our presentation of the Z80, we will present the instruction execution cycle. Each Z80 instruction is executed in a sequence of basic operations known as *M cycles* – or *machine cycles* – labeled *M1*, *M2* and so on [Zil76, p. 11] [Zak80, p. 69]. Each of these M cycles can take from 3 to 6 clock periods to complete[26]. A clock period is known as a *T cycle* – or *T state* – and the T cycles within a M cycle are labeled *T1*, *T2* and so on. The number of M cycles and T cycles required to execute an instruction depends on the instruction and its addressing mode – a simple `NOP` instruction requires 1 M cycle and 4 T cycles, while a complex `INC (IX+d)` requires 6 M cycles and 23 T cycles [Zil76, ch. 7.0]. The first M cycle of any instruction (*M1*) is known as the *fetch cycle* and is used to fetch the opcode of the next instruction to be executed. As such, the content of the program counter is placed on the address bus (*T1*), the program

[26]The execution of an instruction may be lengthened for synchronization purpose [Zil76, p. 12].

counter is incremented (*T2*), and the opcode is fetched and stored in the instruction register (*T3*) [Zak80, pp. 70–71]. The instruction is then decoded and executed (*T4*). Depending on the instruction, additional M cycles may be required for additional operations – memory read or write, I/O operations, bus request/acknowledge, interrupt request/acknowledge, non maskable interrupt request/acknowledge, or exit from a `HALT` instruction [Zil76, ch. 4.0]. As an illustration, we consider the instruction `LD A,(nn)` in *Extended* addressing mode [Zil76, p. 44] [Zak80, pp. 86–89]. This instruction is represented by an 8-bit opcode `$3A` (byte 1) followed by the low-order byte (byte 2) and high-order byte (byte 3) of a 16-bit address. This instruction loads into the accumulator the content of the defined memory location. The execution of this instruction requires 4 M cycles and a total of 13 T cycles. As explained previously, during the first machine cycle *M1* (4 T cycles), the program counter is placed on the address bus, incremented, then the opcode is fetched, stored into the instruction register and decoded. The control unit then finds out that it is necessary to fetch two additional bytes – byte 2 and byte 3 – that will be stored temporarily in special registers *Z* and *W*. During the next machine cycle *M2* (3 T cycles), the program counter is deposited on the address bus, incremented, then byte 2 is fetched and stored in *Z*. During *M3* (3 T cycles), again, the program counter is placed on the address bus, incremented, then byte 3 is fetched and stored in *W*. In the last machine cycle *M4* (3 T cycles), *W* and *Z* are output on the address bus, and the content of the memory location is fetched and stored in the accumulator. The execution of the instruction is completed.

4.3 Conclusion

In his seminal work written in 1945 and entitled *First Draft of a Report on the EDVAC*, John von Neumann has described the first known design of a general purpose digital computer[27] with the particular objective of solving mathematical problems[28] related to ballistics [Neu93] [GH93]. According to von Neumann, the digital computer should be able to perform sequences of operations, such as arithmetic operations (addition, subtraction, multiplication and division) and mathematical functions (square root, cube root, logarithm and so on) on *real numbers*[29]. While emphasizing the need for the digital computer to support natively fundamental operations such as addition and subtraction, von Neumann is more nuanced regarding other arithmetic operations and mathematical functions. Indeed, these operations and functions can be derived from addition and subtraction in different ways [Neu93, ch. 10.3]. Early microprocessors' designs were largely inspired by von Neumann's work[30], and in particular these microprocessors were designed to provide a minimal arithmetic, logic, and control circuitry – the bare essential to perform the functions of a digital computer. This is the case for the MOS 6502 and the Z80, with a 8-bit accumulator-based design and an Arithmetic Logic Unit only supporting addition and subtraction. In this situation, it is left to others to provide a complete *mathematical package*, that supports floating-point numbers, arithmetic operations and usual mathematical functions. For 8-bit home computers, this mathematical package was typically included in the built-in ROM. In the next section, we will present how floating-point numbers, arithmetic operations and com-

[27] Known as the *EDVAC* (*Electronic Discrete VAriable Computer*) [GH93, p. 1].

[28] Such as "solve a non-linear partial differential equation in 2 or 3 independent variables numerically" [Neu93, ch. 1.2].

[29] In the case of the EDVAC, a real number is a sequence of binary digits with a fixed-point representation [Neu93, ch. 7.1].

[30] Refer to section 7.4 for a discussion about the von Neumann architecture.

mon mathematical functions were implemented in 8-bit home computers. To this end, we will rely on the reverse engineering of 8-bit home computers' ROM, and more particularly the reverse engineering of the BASIC interpreter – an essential component of the built-in ROM.

5 Arithmetic and mathematical functions

5.1 Overview

In this section, we will study the arithmetic and mathematical features provided by the BASIC interpreter that is shipped with 8-bit home computers. The BASIC interpreter is a key component of 8-bit home computers, as it provides many features for easy high-level programming such as different data types (numbers, strings, arrays), program flow control, I/O commands, as well as the evaluation of mathematical expressions. It is usually provided by the manufacturer in the form of a built-in ROM [App79, pp. 72–73] [Com82b, pp. 260–267] [CAL84, pp. 493–495] [Jam84, pp. 28–38] [Ams84b, ch. 3]. More specifically, we will study implementations of BASIC interpreters for home computers based on the MOS 6502 and derivatives, such as Commodore BASIC[31], BBC BASIC, Integer BASIC and Applesoft BASIC. We will also study the *Floating Point Package*, that was provided by Apple to supplement Integer BASIC with floating-point arithmetic [App78a, pp. 1, 94, 95]. It is worth noting that since Commodore BASIC and Applesoft BASIC both derive from Microsoft 6502 BASIC, they present many similarities [Ste08] [Ste15]. Likewise, we will also study BASIC interpreters for Z80-based computers, such as Locomotive BASIC and ZX Spectrum BASIC[32].

BASIC interpreters, and more generally 8-bit home computers' ROM, have been studied thoroughly, in particular during the commercial lifetime of 8-bit home computers. As such, we will mostly rely on existing period documentation. The *Floating Point Package* source code written and commented by Wozniak was published by Apple in the *Apple II Reference Manual* [App78a, pp. 94, 95]. Wozniak provided additional explanations later [WA79, p. 109–117]. Ian Logan and Frank O'Hara have published a very thorough study of the ZX Spectrum ROM [LO83]. The ZX Spectrum ROM is notably divided in three major parts: I/O routines, BASIC interpreter and the mathematical expression evaluator. Jeremy Ruston has published an in-depth study of the BBC Micro ROM[33] [Rus85]. Jörn W. Janneck and Till Mossakowski have published detailed and documented ROM listings of each computer of the Amstrad CPC home computer series (Amstrad CPC 464, Amstrad CPC 664 and Amstrad CPC 6128) [JM86]. Note that the Amstrad CPC ROM is divided between the operating system and the Locomotive BASIC interpreter, and the operating system provides the arithmetic and mathematical routines to the BASIC interpreter. Michael Steil has written several articles on Microsoft 6502 BASIC, on which Applesoft BASIC and Commodore BASIC are based, and has also published the source code [Ste08] [Ste15]. It should be noted that in all the documents listed, a particular care has been taken by the authors to study thoroughly the arithmetic and mathematical routines. For ease of reading in the case of Locomotive BASIC and ZX Spectrum BASIC, we will not disassociate the BASIC interpreter itself from the routines provided by the operating

[31] Also known as CBM BASIC.
[32] Also known as Sinclair BASIC.
[33] More specifically, it is a study of BBC BASIC II.

system. Thus, the terms 'Locomotive BASIC' and 'ZX Spectrum BASIC' may also encompass the operating system. This simplification is justified by the fact that the ROM is often presented as a single entity whose primary goal is to provide a interactive BASIC interpreter to the end user. Furthermore, as the BASIC interpreter and operating system are tightly linked, there is no need to make such a distinction.

This section is organized as follows. First, we will present how integers and floating-point numbers are represented. Second, we will study the implementation of the fundamental arithmetic operators – addition, subtraction, multiplication and division. We will discuss the case of integers, and the case of floating-point numbers. Then, we will present how mathematical functions are approximated, with a particular focus on the different methods used. We conclude our study with the question of the trade-off between accuracy and performance, in the context of BASIC interpreters benchmarking.

5.2 Representation of numbers

All BASIC implementations supports integers. Most BASIC implementations require the programmer to declare explicitly integer variables. For this purpose, it is customary to use the `%` type marker to suffix the variable name (Locomotive BASIC [Ams84b, ch. 4.7], Commodore BASIC [Com82b, p. 8], Applesoft BASIC [App78b, p. 18], BBC BASIC [CAL84, p. 55]), or use a specific keyword to declare the variable (`DEFINT` keyword in Locomotive BASIC [Ams84b, ch. 4.7]). A notable exception is ZX Spectrum BASIC where the programmer does not need to declare integers as the type is determined dynamically – if the number fits into the range of an integer, then it is stored as an integer, otherwise it is stored as a floating-point number [Jam84, p. 44–45]. The representation of integers differs from one BASIC implementation to another, as shown in table 12. Locomotive BASIC and Commodore BASIC store an integer in 16 bits with a two's complement little endian representation, allowing values in the $[-32\,768, +32\,767]$ range [JM86, p. 134] [Com82b, p. 5]. Integer BASIC stores an integer on 16 bits as well, but only allows values in the $[-32\,767, +32\,767]$ range [Ras78, p. 27]. Applesoft BASIC – which superseded Integer BASIC as the default BASIC on Apple II computers – reproduces this limitation, likely for backward compatibility purpose with its predecessor [App78b, pp. 17–18]. BBC BASIC stores integers on 32 bits with a two's complement little endian representation, resulting in a large $[-2\,147\,483\,648, +2\,147\,483\,647]$ range [CAL84, p. 55–56]. ZX Spectrum BASIC uses 40 bits to store any number, whether it is an integer or a floating-point number [VB85, p. 170]. In the case of an integer, the first byte is set to `$00`, the second byte is equal to `$00` if it is a positive number or `$FF` if it is a negative number, the third and fourth byte are the less and most significant bytes of the *magnitude* of the number, and the fifth byte is set to `$00`. The magnitude of the number is stored in a form that is similar to two's complement representation, but it is not a true two's complement representation [LO83, p. 229]. For instance, `$00 $00 $FF $FF $00` represents $+65\,535$ and `$00 $FF $01 $00 $00` represents $-65\,535$, but the value `$00 $FF $00 $00 $00` which should theoretically correspond to $-65\,536$ is not handled correctly by ZX Spectrum BASIC [LO83, p. 230]. As a result, ZX Spectrum BASIC allows integers in the $[-65\,535, +65\,535]$ range.

The use of integer variables whenever possible is a good programming practice, as calculation on integers is handled with a perfect accuracy, and the use of integers limits its memory usage and speeds up computation [Ams84b, ch. 4.7] [CAL84, pp. 55, 168–169]. Integer arithmetic routines are indeed faster than their floating-point number

BASIC	Range	Format
Integer BASIC	$[-32\,767, +32\,767]$	16 bits
Applesoft BASIC	$[-32\,767, +32\,767]$	16 bits
Locomotive BASIC	$[-32\,768, +32\,767]$	16 bits
Commodore BASIC	$[-32\,768, +32\,767]$	16 bits
BBC BASIC	$[-2\,147\,483\,648, +2\,147\,483\,647]$	32 bits
ZX Spectrum BASIC	$[-65\,535, +65\,535]$	40 bits

Table 12: Range and format of integers.

counterparts as they require less operations and memory accesses. To this purpose, many BASIC implementations propose dedicated routines for the arithmetic operations on integers (Locomotive BASIC, BBC BASIC, ZX Spectrum BASIC), as we will show later. There are however BASIC implementations that do not propose dedicated routines for integer arithmetic operations. For instance, Applesoft BASIC and Commodore BASIC perform arithmetic operations on floating-point numbers only [App78b, p. 18] [Com82b, pp. 10–11]. When an arithmetic operation on integers is carried out, integers are first converted to floating-point numbers, the operation on floating-point numbers is performed, and the floating-point result is converted to an integer in the case it is assigned to an integer variable. In this situation, arithmetic operations on integers are even slower than arithmetic operations on floating-point numbers, as extra conversions are necessary. If the use of integer variables is a good programming practice – both in term of memory usage and runtime performance – for BASIC interpreters that use dedicated integer computation routines (Locomotive BASIC, BBC BASIC, ZX Spectrum BASIC), there is however a memory-performance trade-off that needs to be addressed by the programmer for BASIC interpreters that only rely on floating-point computation routines (Applesoft BASIC, Commodore BASIC).

Regarding real numbers, there are different ways to represent them. Real numbers can be represented as fixed-point numbers. In fixed-point representation, a fixed binary point is used to separate the integer part and the fractional part of the number. Fixed-point representation however suffers from limited range and precision, and computations have to be *scaled* to ensure that values remain presentable and to prevent precision loss [Par99, p. 352]. The fixed-point representation is notably mentioned by John von Neumann in his seminal report and was used in the two versions of the *EDVAC – vN-EDVAC* and *M-EDVAC* [Neu93, ch. 8.1] [GH93, pp. 13, 15]. Real numbers can alternatively be represented as floating-point numbers. A floating-point number in base b consists of a sign, a significand[34] s and an exponent e, where the significand s is a unsigned fixed-point number and the exponent e is a signed integer. A floating-point number x is computed as follows [Par99, pp. 352–353]:

$$x = \pm s \times b^e$$

Which is expressed in the case of a binary representation:

$$x = \pm s \times 2^e$$

[34] Also known as *mantissa*.

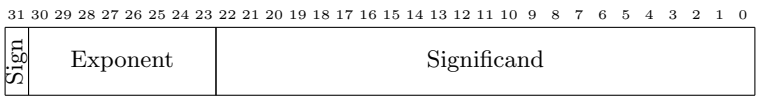

Figure 8: ANSI/IEEE 754-1985 single basic floating-point number format [The85, p. 11].

In order to maximize the precision, it is customary that the significand s is normalized and has the following property:

$$0.5 \leq s < 1$$

In this case, as the most significant bit is always set, it can be discarded. It is also customary to represent the signed exponent as an unsigned value by adding a *bias*[35]. The significand is usually stored in *big endian* byte order, from the most significant byte to the least significant byte, regardless of the microprocessor *endianness*.

Floating-point representation addresses limitations of fixed-point representation as it allows a wide range of values – from extremely large numbers to very small numbers [Par99, p. 351]. As there were many different implementations of floating-point arithmetic in the early days of computing, the *ANSI/IEEE Standard 754-1985* standard was published in 1985 by the IEEE (Institute of Electrical and Electronics Engineers) to ensure portability and reliability[36] in this matter [The85]. The standard specifies floating-point number formats, rounding rules, required operations (addition, subtraction, multiplication, division, remainder, square root and comparisons), conversion rules between different formats, and how special values[37] and exceptions[38] shall be handled. Regarding the floating-point number formats defined by the standard, they are organized in two groups – the *basic* group and the *extended* group – where each group has a *single* width format and a *double* width format [The85, ch. 3]. The *single basic* format is 32 bits long, with a sign bit, an 8-bit exponent with a +127 bias, and a 23-bit significand. The *double basic* format is 64 bits long, with a sign bit, an 11-bit exponent with a +1023 bias, and a 52-bit significand, allowing a larger range and a greater precision. Figure 8 illustrates the 32-bit single basic format. As the standard was developed in collaboration with microprocessor and computer manufacturers, it was already adopted by the industry before its formal approval and publication [Par99, p. 355] [The85, p. 18]. An example of early hardware implementation of the standard is the Motorola 68881 floating-point co-processor [Mot88, p. 4-33].

Regarding the different BASIC interpreters on 8-bit home computers, they all adopt a floating-point representation for real numbers, and floating-point number is usually the default variable type [App78b, pp. 17–18] [Com82b, p. 8] [Ams84b, ch. 4.7] [CAL84, p. 55]. There are, however, differences in the implementations as the representation of floating-point numbers was not yet standardized. A notable implementation

[35] The use of an unsigned value adjusted with a bias – instead of the standard two's complement representation – notably facilitates the zero detection [Par99, p. 353].

[36] Refer to *Floating-point Arithmetic in Applesoft BASIC* for examples of anomalies found in Applesoft BASIC [Tho85]. Such anomalies found in different software implementations of floating-point arithmetic have motivated the creation of a standard.

[37] Infinities ($+\infty$, $-\infty$), *NaN* (Not a Number) and signed zeros ($+0$, -0) [The85, ch. 6].

[38] *Invalid Operation*, *Division by Zero*, *Overflow*, *Underflow* and *Inexact* exceptions [The85, ch. 7].

is the one by Wozniak, in the context of the *Floating Point Package* [App78a, pp. 1, 94, 95]. In this implementation, a floating-point number is represented by 32 bits, where 8 bits are used for the exponent and 24 bits for the significand[39] [WA79, p. 111]. The 8-bit exponent is adjusted with a +128 bias, ranging from −128 (`$00`) to +127 (`$FF`). The 24-bit significand is normalized to retain maximum precision and stored in two's complement form, in big endian byte order. The 2 most significant bits of the high-order byte are used to store the significand sign and handle special cases (unnormalized significand, exponent equal to −128). As Applesoft BASIC and Commodore BASIC both derive from Microsoft 6502 BASIC, they share the same implementation of floating-point numbers [Ste08]. In this implementation, floating-point numbers are stored in 40 bits, with 8 bits for the exponent and 32 bits for the significand stored in big endian byte order [Tho85]. The exponent is adjusted with a +128 bias and ranges from −127 (`$01`) to +127 (`$FF`). If the exponent is equal to `$00` then the floating-point number is considered to be zero, regardless of the significand. The most significant bit of the 32-bit significand is a sign bit. The remaining 31 bits of the significand are used with an implicit high-order bit to maximize precision. The 32-bit significand gives a precision of 9 digits, values range from $\pm 2.938\,735\,88 \times 10^{-39}$ to $\pm 1.701\,411\,83 \times 10^{38}$. ZX Spectrum BASIC, Locomotive BASIC and BBC BASIC also implement a 40-bit floating number representation, with an 8-bit exponent adjusted with a +128 bias, and a 32-bit significand [VB85, pp. 169–170, 197] [JM86, pp.127–129] [Rus85, ch. 4]. While ZX Spectrum BASIC and BBC BASIC store first the 8-bit exponent, followed by the 32-bit significand in big endian byte order, Locomotive BASIC stores the bytes in the opposite order. For instance, 0.5 is represented `$80 $00 $00 $00 $00` in ZX Spectrum BASIC, while it is represented `$00 $00 $00 $00 $80` in Locomotive BASIC [LO83, p. 190] [JM86, p. 375].

To illustrate the representation of integers and floating-point numbers, we will present numerical constants stored in different BASIC interpreters that are frequently used for mathematical computation. Table 13 presents constants used by the ZX Spectrum BASIC, and illustrates the dual representation of integers and floating-point numbers in 40 bits [LO83, pp. 169–170, 190]. Table 14 presents floating-point constants used by Locomotive BASIC for trigonometric functions [JM86, p. 379]. Note that the significand is stored first in little endian byte order, and the exponent is stored last. Numerical constants can also be found in BBC BASIC, and in Microsoft 6502 BASIC and derivatives [Rus85, pp. 423, 424, 429, 430, 436, 437] [Ste08] [Ste15] [Ste20].

Type	Constant	Value
Integer	`$00 $00 $00 $00 $00`	0
Integer	`$00 $00 $01 $00 $00`	1
Floating-point	`$80 $00 $00 $00 $00`	0.5
Floating-point	`$81 $49 $0F $DA $A1`	Approximation of $\pi/2$
Integer	`$00 $00 $0A $00 $00`	10

Table 13: Table of mathematical constants in ZX Spectrum BASIC [LO83, p. 190].

[39] Or mantissa [WA79, p. 111]

Constant	Value
`$6E $83 $F9 $22 $7F`	Approximation of $1/\pi$
`$B6 $60 $0B $36 $79`	Approximation of $1/180$
`$13 $35 $FA $0E $7B`	Approximation of $\pi/180$
`$D3 $E0 $2E $65 $86`	Approximation of $180/\pi$

Table 14: Table of trigonometric constants in Locomotive BASIC [JM86, p. 190].

5.3 Arithmetic operations on integers

The addition of k-bit integers, where k is a multiple of 8, is straightforward as it can be achieved with a sequence of k 8-bit *add with carry* instructions. The addition is performed by adding the bytes of the two operands, from the least significant byte to the most significant byte. If the addition of two bytes overflows, then a carry is generated and will be added to the next addition. The MOS 6502 and the Z80 both propose an add with carry instruction (`ADC` [Zak78, pp. 98–99] [Zak80, pp. 190–193]). Listing in figure 9 presents an example of 16-bit big endian byte order addition on the MOS 6502, where the first 16-bit operand is stored at `ADR1-1` memory location, the second 16-bit operand at `ADR2-1` memory location, and the 16-bit result at `ADR3-1` memory location [Zak78, pp. 47–50]. Refer to [Zak80, pp. 99-105] for an implementation on the Z80.

```
CLC
CLD
LDA  ADR1
ADC  ADR2
STA  ADR3
LDA  ADR1-1
ADC  ADR2-1
STA  ADR3-1
```

Figure 9: Addition of 16-bit integers on the MOS 6502 [Zak78, pp. 47–50].

A similar method can be used for subtraction, based on a sequence of 8-bit *subtract with carry* instructions (`SBC` [Zak78, pp. 158-159] [Zak78, pp. 420-423]). Refer to [Zak78, pp. 50–51] and [Zak80, pp. 105–107] for examples of implementations.

The multiplication of two binary numbers is similar to the way one multiplies using pencil and paper. An early description of a binary multiplier was provided by von Neumann, with the requirement that the multiplier and multiplicand be fixed-point numbers between -1 and $+1$ [Neu93, ch. 9]: "Binary multiplication consists of this: For each digital position in the multiplier (going from left to right), the multiplicand is shifted by one position to the right, and then it is or is not added to the sum of partial products already formed, according to whether the multiplier digit under consideration is 1 or 0." [Neu93, ch. 7.7]. This multiplier, described by von Neumann, highlights the two basic operations required to carry out a multiplication: shifting and addition. A standard integer multiplication algorithm – known as the *shift-and-add* algorithm – is based on these two basic operations [Par99, ch. 9.1]. More formally, if we consider a base b, a multiplicand $a = a_{k-1}a_{k-2}...a_1a_0$, a multiplier $x = x_{k-1}x_{k-2}...x_1x_0$, and

a product $p = a \times x$, then p is equal to the sum of the partial products $x_i ab^i$. In the case of a binary multiplication, the $2k$-bit product p can be written as follows:

$$p = \sum_{i=0}^{k-1} x_i a 2^i \quad x_i \in \{0, 1\}$$

Thus, the binary multiplication is reduced to a sequence of shifting and adding operations. Two variants of the algorithm exist, the *left-shift* algorithm, and the *right-shift* algorithm. Both algorithms require k iterations of shifting and adding. However, it is recommended to use the right-shift algorithm, as it only requires k-bit wide additions, while the left-shift algorithm needs $2k$-bit wide additions [Par99, ch. 9.1].

Multiplication is such a fundamental operation that Zilog provides in the Z80 technical manual an example of implementation for unsigned integers [Zil77, pp. 68]. The routine takes as an input two 16-bit unsigned numbers stored respectively in *DE* and *HL* registers, and outputs the product in the 16-bit *HL* register – which is an important drawback as it is assumed that the final product can fit in 16 bits. A total of 16 iterations are performed, and the right-shift is carried out with the `SRL` instruction, as shown on figure 10. Refer also to Zaks for other examples of implementations [Zak80, pp. 113–133] [Zak78, pp. 56–73].

```
        LD   B,16
        LD   C,D
        LD   A,E
        EX   DE,HL
        LD   HL,0
MLOOP:  SRL  C
        RRA
        JR   NC,NOADD
        ADD  HL,DE
NOADD:  EX   DE,HL
        ADD  HL,HL
        EX   DE,HL
        DJNZ MLOOP
```

Figure 10: Multiplication of 16-bit unsigned integers on the Z80 [Zil77, p. 68].

Von Neumann also described a binary diviser based on the pencil and paper method[40] [Neu93, ch. 8.3]. This division algorithm is known today as the *shift-and-subtract* algorithm, since only shift and subtract operations are necessary for its implementation [Par99, ch. 13.1]. We consider the division equation:

$$z = (d \times q) + s \quad s < d$$

Where $z = z_{2k-1} z_{2k-2} ... z_1 z_0$ is the dividend, $d = d_{k-1} d_{k-2} ... d_1 d_0$ the divisor, $q = q_{k-1} q_{k-2} ... q_1 q_0$ the quotient, and $s = s_{k-1} s_{k-2} ... s_1 s_0$ the remainder. The binary division can be performed in k iterations of shifting and subtraction, where one bit of

[40] Or *long division* algorithm.

the quotient is computed at each iteration, from q_{k-1} to q_0. The recurrence relation is defined as such, where the final term $s^{(k)}$ is equal to $2^k s$:

$$s^{(0)} = z$$
$$s^{(j)} = 2s^{(j-1)} - q_{k-j}(2^k d)$$

We will now give an overview of the implementation of integer arithmetic in various BASIC interpreters. Note that it is customary for division and integer division operations to be distinct, since the former computes a real number quotient, while the latter computes an integer quotient and an integer remainder. Regarding Integer BASIC, it does not support floating-point arithmetic, and only implements integer arithmetic routines for the usual arithmetic operations – addition, subtraction, multiplication, integer division and modulo – and for exponentiation (Integer BASIC operators `+`, `-`, `*`, `/`, `MOD` and `^` [Ras78, pp. 25–26]). Microsoft 6502 BASIC and derivatives (Applesoft BASIC, Commodore BASIC) do not implement specific arithmetic routines for integers, as 16-bit integers are systematically converted to and from 40-bit floating-point numbers before and after an arithmetic operation, so that only floating-point routines are necessary (refer to `GIVAYF` and `AYINT` conversion routines [Ste15]). Locomotive BASIC has a specific integer package that contains routines dedicated to integer arithmetic operations [JM86, p. 134, pp. 396–p401]. These routines are used for the addition, subtraction, multiplication, integer division[41] and modulo operations (Locomotive BASIC operators `+`, `-`, `*`, `\` and `MOD` [Ams84b, ch. 4.3] [JM86, pp. 589–590]). BBC BASIC provides specific integer subroutines for addition, subtraction, multiplication, integer division[42] and modulo operations (BBC BASIC operators `+`, `-`, `*`, `DIV` and `MOD` [CAL84, ch. 26] [Rus85, pp. 344–347, pp. 358–359, pp. 360–p361, pp. 363–p368]). Multiplication is implemented with the shift-and-add algorithm (`$9D41` memory location), while the division is based on the shift-and-subtract algorithm (`$99BE` memory location). ZX Spectrum BASIC also executes dedicated sections of code, in the case of the addition, subtraction and multiplication of integers (ZX Spectrum BASIC operators `+`, `-` and `*` [VB85, ch. 7] [LO83, pp. 176–184]). The unsigned integer multiplication subroutine[43] implements a shift-and-add algorithm (`$30A9` memory location), as shown in figure 11.

An arithmetic operation on integers may produce a result that does not fit the integer format. In this case, BASIC interpreters that require an explicit declaration of integers usually display an error message to indicate that an overflow has occurred (`ERR` error message in Integer BASIC [Ras78, p. 27], `Overflow` error message in Locomotive BASIC [Ams84b, app. VIII], `Too big` error message in BBC BASIC [CAL84, p. 470]). ZX Spectrum BASIC adopts a different strategy as the type of the number is determined dynamically [Jam84, p. 44–45]. In the case of an integer arithmetic operation overflow, ZX Spectrum BASIC converts the 16-bit integers to 40-bit floating-point numbers, and then performs the floating-point arithmetic operation. This is illustrated in the main multiplication subroutine (`$30CA` memory location [LO83, pp. 180–184]). If the 16-bit unsigned integer multiplication subroutine sets the carry flag indicating an overflow (addition operations at `$30B1` and `$30B9` memory locations), then the code jumps to `$30EF` memory location to convert the integers to floating-point numbers,

[41] Locomotive BASIC uses `/` operator for division.

[42] BBC BASIC uses `/` operator for division.

[43] This unsigned integer multiplication subroutine is called by the main multiplication routine, see [LO83, pp. 180–184] for more details. Refer to section 5.2 for the representation of integers in ZX Spectrum BASIC.

```
$30A9   PUSH  BC
$30AA   LD    B,$10
$30AC   LD    A,H
$30AD   LD    C,L
$30AE   LD    HL,0
$30B1   ADD   HL,HL
$30B2   JR    C,$30BE
$30B4   RL    C
$30B6   RLA
$30B7   JR    NC,$30BC
$30B9   ADD   HL,DE
$30BA   JR    C,$30BE
$30BC   DJNZ  $30B1
$30BE   POP   BC
$30BF   RET
```

Figure 11: ZX Spectrum BASIC 16-bit unsigned integer multiplication subroutine [LO83, pp. 179–180].

and then perform the floating-point multiplication. The same strategy is used in the main addition subroutine where, in the case of an overflow, the code jumps to `$303C` memory location to convert the operands and perform a floating-point addition [LO83, pp. 176–179].

5.4 Arithmetic operations on floating-point numbers

Arithmetic on floating-point numbers differs from arithmetic on integers. In particular, the floating-point representation is naturally suited for multiplication and division operations, whereas addition is more difficult to implement [Par99, ch. 17.3]. The addition is typically performed in three steps. We consider the following expression:

$$\pm s \times b^e = (\pm s_1 \times b^{e_1}) + (\pm s_2 \times b^{e_2}) \quad e_1 \geq e_2$$

First, the two operands are aligned through shifting s_2 to the right by $e_1 - e_2$ digits. This step is known as *alignment shift* or *preshift*. The expression becomes:

$$\pm s \times b^e = (\pm s_1 \pm \frac{s_2}{b^{e_1 - e_2}}) \times b^{e_1} \quad e_1 \geq e_2$$

The addition is then performed. Finally, a normalization step known as *normalization shift* or *postshift* is carried out. If the operands' signs are alike, then a one digit normalization shift of the resulting significand s is enough. If the operands' signs are different, then the resulting significand s may need many normalization shifts as it may be very close to zero. Overflow and underflow may happen during the addition and normalization steps [The85, ch. 7.3, 7.4].

Floating-point multiplication is more straightforward as it is performed by multiplying the significands and adding the exponents, as shown by the following expression:

$$\pm s \times b^e = (\pm s_1 \times b^{e_1}) \times (\pm s_2 \times b^{e_2}) = \pm (s_1 \times s_2) \times b^{e_1 + e_2}$$

Normalization may be necessary as $s_1 \times s_2$ may be unnormalized. Overflow and underflow may also happen if the exponents' signs are alike, and overflow is possible during the normalization.

Similarly, floating-point division is carried out by dividing the significands and subtracting the exponents, as the following expression shows:

$$\pm s \times b^e = \frac{\pm s_1 \times b^{e_1}}{\pm s_2 \times b^{e_2}} = \pm \frac{s_1}{s_2} \times b^{e_1 - e_2}$$

Once again, normalization may be necessary, and overflow and underflow may happen under certain conditions. Note that arithmetic operations on floating-point numbers may require rounding of the result. Different rounding methods exist [The85, ch. 4] [Par99, ch. 17.4].

All the studied floating-point arithmetic implementations use the aforementioned methods to implement addition, subtraction, multiplication and division: *Floating Point Package* (`FADD`, `FSUB`, `FMULT` and `FDIV` routines [App78a, pp. 94, 95] [WA79, p. 109–117]), Applesoft BASIC and Commodore BASIC (`FADD`, `FSUB`, `FMULT` and `FDIV` routines [Ste15]), BBC BASIC (`$9C8B`, `$9CE1`, `$9D20` and `$A6AD` BBC BASIC II ROM routines [Rus85, ch. 4, ch. 9]), ZX Spectrum BASIC (`$3014`, `$300F`, `$30CA` and `$31AF` ROM routines [LO83]) and Locomotive BASIC (`$333F`, `$333B`, `$3415` and `$349E` Amstrad CPC 464 ROM routines [JM86, ch. 5.3, 6.1.1]).

As an illustration, we will present the addition and multiplication implementations by Wozniak in the *Floating Point Package* [App78a, pp. 94, 95] [WA79, p. 109–117]. The addition is implemented in the `FADD` routine [WA79, p. 112]. The two operands are stored respectively in the zero page floating-point accumulators `FP1` and `FP2`, where `FP1` consists of an 8-bit exponent `X1` and a 24-bit significand `M1`, and likewise `FP2` consists of an 8-bit exponent `X2` and a 24-bit significand `M2`. The operands are expected to be normalized, for maximum accuracy. The two exponents `X1` and `X2` are compared. If different, then the operands may be swapped (`SWAP` subroutine), as the alignment is only performed on `FP1`, and then `FP1` is aligned by shifting `M1` right and incrementing `X1` one or several times (`RTAR` subroutine). As the operands are aligned, the addition of the 24-bit significands `M1` and `M2` can be performed, and the sum is stored in `M1` (`ADD` subroutine). Finally, `FP1` is normalized if necessary (`NORM` subroutine). Table 15 illustrates the addition of +12 and −5 [WA79, p. 112].

Type	Zero page	Content	Value
Operand	`FP1`	`$83 $60 $00 $00`	+12
Operand	`FP2`	`$82 $B0 $00 $00`	−5
Sum	`FP1`	`$82 $70 $00 $00`	+7

Table 15: Addition of +12 and −5 in *Floating Point Package* [WA79, p. 112].

The multiplication is implemented in the `FMUL` routine [WA79, p. 113]. As previously, the operands are stored in `FP1` and `FP2`, and are expected to be normalized. First, `MD1` and `ABSWAP` subroutines are called. At this point, `X2` is stored in the accumulator. The two exponents `X1` and `X2` are then added with the `ADC X1` instruction. The `MD2` subroutine is then called to prepare the multiplication of the significands – this subroutine initializes the multiplication loop counter with `LDY #$17` instruction. A shift-and-add multiplication algorithm is then performed: the `RTLOG1` subroutine shifts the multiplier `M1`, and the `ADD` subroutine adds the multiplicand `M2` to the product. The final product in `FP1` is then normalized if necessary (`NORM` subroutine).

5.5 Approximation of mathematical functions

Many mathematical functions can be defined by an infinite expansion of some form [Har+78, ch. 2]. The most familiar form of infinite expansions are series, based on a linear combination of some basis function. Taylor series are a well-known example of series expansions that will be described later on. Another form of infinite expansions are rational forms, such as continued fractions, that will be discussed as well. The numerical approximation of a function f can be achieved by computing a finite expansion. Several parameters regarding the approximation are to be considered [Har+78, ch. 1]. First, the choice of the mathematical function f. Some functions are more difficult to approximate than others, and as many functions are related to each other, it is possible to derive many functions from a restricted set of fundamental mathematical functions. For instance, trigonometric functions are all related to each other and can be derived from a single trigonometric function, e.g., sine function. Second, the domain of the function [Har+78, ch. 4.2]. The efficiency of the approximation can be greatly improved by reducing the domain, when possible. Domain reduction can be achieved by using the properties of the function such as periodicity and symmetry. For instance, if we consider the sine function, then the domain can be reduced to $[0, 2\pi]$ since it is a periodic function of period 2π. Furthermore, as sine is anti-symmetric about the point $x = \pi$ over the $[0, 2\pi]$ domain, then we can further reduce the domain to $[0, \pi]$. Again, as sine is symmetric about the line $x = \frac{\pi}{2}$ over the $[0, \pi]$ domain, then we can finally reduce the domain to $[0, \frac{\pi}{2}]$. Additions properties can also be used to reduce the approximation domain. If we consider the logarithm and exponential functions for a given base b, then we have the following relations: $log_b(xy) = log_b(x) + log_b(y)$, $b^{x+y} = b^x b^y$. Third, the precision of approximation. The precision depends on many factors, and notably the number of terms of the finite expansion. From a computational perspective, there is a trade-off between precision and cost [Har+78, ch. 4.3].

The polynomial form P_n of degree n is defined as [Har+78, ch. 4.5]:

$$P_n(x) = \sum_{k=0}^{n} a_k x^k$$

According to Taylor [Tay15] [AS64, ch. 3.6], any function f satisfying certain conditions may be approximated around a given point x_0 by a Taylor series expansion of degree n, where R_n is the remainder:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f^{(2)}(x_0)}{2!}(x - x_0)^2 + ... + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + R_n(x)$$

$$= \sum_{k=0}^{n} f^{(k)}(x_0)k!(x - x_0)^k + R_n(x)$$

A particular type of Taylor series expansions are Maclaurin series expansions, where the function f is approximated at $x_0 = 0$. For instance, the Maclaurin series expansion of degree n of the exponential function is defined as [Har+78, ch. 6.2.9] [AS64, ch. 4.2.1]:

$$e^x = \sum_{k=0}^{n} \frac{x^k}{k!} + R_n(x)$$

The approximation of a Taylor series expansion is very good in the close neighborhood of x_0 since it is calculated with the value and the derivatives of f at x_0. However,

the error grows quickly when x moves away from x_0. To approximate the function, the coefficients a_k of P_n should be computed so that the error is evenly distributed over the interval. For this purpose, the Remes[44] algorithm [Har+78, ch. 3.2, 3.6] is typically used to compute these coefficients as it minimizes the maximum absolute difference $|P_n(x) - f(x)|$ between the polynomial P_n and the function f over the considered interval. A well known reference that provides tables of polynomial coefficients computed by a variant of the Remes algorithm is *Computer Approximations* [Har+78, ch. 7]. An alternative error metric may be used, such as the maximum relative difference $|\frac{P_n(x)-f(x)}{f(x)}|$, the choice of the error metric depending on the function being approximated [Har+78, ch. 1.6].

From a computational point of view, the evaluation of $P_n(x)$ is usually performed with Horner's recursive method [Har+78, ch. 4.5] as it is considered efficient, i.e., a polynomial of degree n is evaluated with only n additions and n multiplications. We consider the polynomial $P_n(x)$ in the following form:

$$P_n(x) = \sum_{k=0}^{n} a_k x^k = [...(a_n x + a_{n-1})x + ... + a_1]x + a_0$$

Then we define the following recursive evaluation known as the *Horner's scheme*:

$$V_{n=} a_n$$

$$V_k = xV_{k+1} + a_k \quad k = n-1, n-2, ..., 0$$

Finally:

$$P_n(x) = V_0$$

An alternative approach to Taylor series expansion is Chebyshev series expansion, where a function f is approximated by a linear combination of Chebyshev polynomials of the first kind $T_k^*(x)$:

$$f(x) = P_n(x) + R_n(x) = \sum_{k=0}^{n} A_k T_k^*(x) + R_n(x)$$

Chebyshev polynomials of the first kind are obtained from the following recurrence relation [Cle54]:

$$T_0^*(x) = 1$$
$$T_1^*(x) = x$$
$$T_{k+2}^*(x) = 2T_{k+1}^*(x) - T_k^*(x)$$

Coefficients A_k for common mathematical functions can be found in *Polynomial approximations to elementary functions* [Cle54] and *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables* [AS64]. For instance, the exponential function can be approximated on interval $[0, 1]$ with a precision of 9 decimal places with the coefficients listed in table 16.

The evaluation of Chebyshev series is usually performed with Clenshaw's summation method. To this purpose, given a value x and a series of Chebyshev coefficients A_n, $\cdots$, A_0, a sequence of values B_n, $\cdots$, B_0 is constructed by applying the following recurrence relation [Cle55]:

[44] Or Remez.

k	$\mathbf{A_k}$
0	1.753 387 654
1	0.850 391 654
2	0.105 208 694
3	0.008 722 105
4	0.000 543 437
5	0.000 027 115
6	0.000 001 128
7	0.000 000 040
8	0.000 000 001

Table 16: Chebyshev coefficients for approximation of e^x on $[0, 1]$ [Cle54].

$$B_{n+2}(x) = 0$$
$$B_{n+1}(x) = 0$$
$$B_k(x) = A_k + (4x - 2)B_{k+1}(x) - B_{k+2}(x) \quad k = n, n-1, \cdots, 0$$

Then an approximation of $f(x)$ is:

$$P_n(x) = B_0(x) - (2x - 1)B_1(x)$$

Compared to Taylor series, a Chebyshev series expansion is likely to converge faster, and as such usually achieves a better precision for an equivalent computational cost [Har+78, ch. 3.3].

An expansion can also be represented in a rational form, such as a continued fraction[45]. In this case, the function $f(x)$ can be expressed as follows, where $C_n(x)$ is a truncated continued fraction [Har+78, ch. 2.2]:

$$f(x) = C_n(x) + R_n(x) = P_0(x) + \frac{P_1(x)}{Q_1(x)+}\frac{P_2(x)}{Q_2(x)+}\cdots\frac{P_n(x)}{Q_n(x)} + R_n(x)$$

In some situations, continued fractions converge more rapidly than polynomial expansions. For example, we consider the arctangent function expressed as a continued fraction:

$$arctan(x) = \frac{x}{1+}\frac{x^2}{3+}\frac{4x^2}{5+}\cdots\frac{k^2x^2}{(2k+1)+}\cdots \quad |x| < \infty$$

The rate of convergence of the continued fraction exceeds that of the Taylor series by a factor of $\sim 4^{-k}$[Har+78, p. 122]. The evaluation of a truncated continued fraction can be performed with different methods, and the choice of the method is guided by the respective cost of the required operations (addition, multiplication, division). [Har+78, ch. 4.6]. We consider the following truncated continued fraction $C_n(x)$:

$$C_n(x) = \frac{a_0}{b_0+}\frac{a_1}{b_1+}\cdots\frac{a_n}{b_n}$$

A first method is based on the following recurrence relation:

[45] We will use the compact notation to represent continued fractions [Har+78, ch. 2.2].

$$p_{-1} = 0 \quad p_0 = a_0$$
$$q_{-1} = 1 \quad q_0 = b_0$$
$$p_k = b_k p_{k-1} + a_k p_{k-2} \quad k = 1, \cdots, n$$
$$q_k = b_k q_{k-1} + a_k q_{k-2} \quad k = 1, \cdots, n$$
$$C_n(x) = \frac{p_n}{q_n}$$

This method requires $2n-1$ additions, $2n-1$ multiplications and only one division. A second method evaluates the continued fraction iteratively starting from the last term, and requires $n+1$ divisions and n additions. It is expressed as follows:

$$e_0 = b_n$$
$$e_k = b_{n-k} + \frac{a_{n-k+1}}{e_{k-1}} \quad k = 1, \cdots, n$$
$$C_n(x) = \frac{a_0}{e_n}$$

After this overview of approximation methods, we will now present the various implementations in the different 8-bit home computers BASIC interpreters. If these implementations share some similarities, they also differ in many details. The first observation is that all the BASIC implementations studied (Microsoft 6502 BASIC, BBC BASIC, ZX Spectrum BASIC and Locomotive BASIC) approximate a reduced set of mathematical functions, and deduce other functions from this reduced set[46]. It is also notable that this reduced set of functions is the same among the different implementations, and these functions are the exponential, logarithm, sine and arctangent functions. Indeed, trigonometric functions can be deduced from sine, inverse trigonometric functions from arctangent, and exponentiation functions from logarithm and exponential functions, as shown in table 17 [Har+78, ch. 6].

Function	**Relation**
Common logarithm	$log_{10}(x) = log_{10}(e) \times ln(x) \quad x > 0$
Cosine	$cos(x) = sin(\frac{\pi}{2} + x)$
Tangent	$tan(x) = \frac{sin(x)}{cos(x)} \quad x \neq (2k + \frac{1}{2})\pi$
Arcsine	$arcsin(x) = arctan(\frac{x}{\sqrt{1-x^2}}) \quad \lvert x \rvert \leq 1$
Arccosine	$arccos(x) = \frac{\pi}{2} - arcsin(x) \quad \lvert x \rvert \leq 1$
Exponentiation	$x^y = e^{y \times ln(x)} \quad x > 0$
Square root	$\sqrt{x} = x^{\frac{1}{2}} \quad x > 0$

Table 17: Relations between mathematical functions [Har+78, ch. 6].

Regarding Microsoft 6502 BASIC, the source code reveals that it can be compiled with *standard precision* (32-bit floating-point number) by default, or *additional precision* (40-bit floating-point number) if the `ADDPRC` compilation flag is set [Ste08] [Ste15]. Both Commodore BASIC and Applesoft BASIC are compiled with additional precision. The approximation of the exponential, logarithm, sine and arctangent functions is performed with polynomials, and polynomials are evaluated with Horner's

[46] With the exception of the square root function in BBC BASIC, as it will be shown further.

method [Wik20]. The implementation is very likely inspired by Hart et al., as the coefficients can be found – directly or indirectly – in *Computer Approximations*. Table 19 presents related tables of coefficients from *Computer Approximations*, with the table index, approximated function, range, precision[47] and degree of the polynomial. Tables 1044 and 4995 can be found in the source code of Microsoft BASIC 6502 at labels `EXPCON` and `ATNCON`. Other tables cannot be found directly in the source code, but the coefficients may have been derived. For instance, it is likely that the coefficients used to approximate the sine function at label `SINCON` have been computed by correcting the coefficients from table 3342 with a 4^k factor, as the approximated sine function is $sin(2\pi x)$, while table 3342 gives coefficients for the approximation of $sin(\frac{\pi}{2}x)$. Likewise, the coefficients used to approximate the logarithm base 2 function at label `LOGCN2` may have been derived from tables 2302 and 2662 that are associated respectively to logarithm base 10 and natural logarithm. Note that in the case of standard precision, coefficients are also related to *Computer Approximations* – tables 1043 and 4992 can be found in the source code at labels `EXPCON` and `ATNCON`. Other mathematical functions provided by Microsoft 6502 are based on this reduced set of functions (square root, exponentiation, tangent, cosine).

Function	Source code	Description
Exponential	`EXPCON`	Coefficients found in table 1044
Logarithm	`LOGCN2`	Coefficients derived from tables 2302 and 2662
Sine	`SINCON`	Coefficients derived from table 3342
Arctangent	`ATNCON`	Coefficients found in table 4995

Table 18: Relations between Microsoft 6502 BASIC (additional precision) and *Computer Approximations* [Har+78] [Wik20] [Ste15].

Table	Function	Range	Precision	Degree
1044	$2^x \approx P(x)$	$[0, 1]$	10.39	7
2302	$log_{10}(x) \approx zP(z^2) \quad z = \frac{x-1}{x+1}$	$[\frac{1}{\sqrt{2}}, \sqrt{2}]$	10.28	3
2662	$log_e(x) \approx P(z^2) \quad z = \frac{x-1}{x+1}$	$[\frac{1}{\sqrt{2}}, \sqrt{2}]$	9.92	3
3342	$sin(\frac{\pi}{2}x) \approx xP(x^2)$	$[0, 1]$	10.67	5
4995	$arctan(x) \approx xP(x^2)$	$[0, tan(\frac{\pi}{4})]$	> 9.43	11

Table 19: Tables of coefficients in *Computer Approximations* [Har+78, ch. 6].

With regard to Locomotive BASIC, the approximation of sine, arctangent and logarithm functions is performed with polynomials and Horner's method [JM86, pp. 374, 378–380]. Coefficients[48] for sine and arctangent functions can be found in *Computer approximations*, respectively in table 3342 and 4994 (see table 20). Approximation of

[47] There is an erratum in *Computer Approximations*, the table 4995 is not referenced correctly [Har+78, p. 129]. The precision of 9.43 corresponds to the table 4994 of degree 10. As the table 4995 is of degree 11, its precision is strictly superior to 9.43.

[48] These coefficients are not those of Taylor or Maclaurin series expansions as suggested by Janneck and Mossakowski [JM86, pp. 378–379], as they have been calculated with a variant of the Remes algorithm to yield a better approximation on the considered interval [Har+78, ch. 7].

the logarithm function is valid for the $[\frac{1}{\sqrt{2}}, \sqrt{2}]$ interval, and is based on the relation between the natural logarithm function and the inverse hyperbolic tangent function [JM86, p. 132] [Har+78, ch. 6.3.5]:

$$arctanh(x) = \frac{1}{2} \times ln(\frac{1+x}{1-x})$$

From which we can deduce:

$$ln(x) = 2 \times arctanh(y) \quad y = \frac{x-1}{x+1}$$

The approximation of the inverse hyperbolic tangent function is then performed with a polynomial[49]. Finally, the exponential function is approximated with a rational form, by dividing P_1 by $P_1 - P_2$, where P_1 and P_2 are two polynomials[50] [JM86, pp. 132–133, 374–375]. All other mathematical functions supported by Locomotive BASIC are deduced from these functions (common logarithm, cosine, tangent, exponentiation, square root).

Constant	Approximated value
`$1B $2D $1A $E6 $6E`	$-0.342\,879\,073 \times 10^{-5}$
`$F8 $FB $07 $28 $74`	$+0.160\,247\,028\,83 \times 10^{-3}$
`$01 $89 $68 $99 $79`	$-0.468\,165\,101\,634 \times 10^{-2}$
`$E1 $DF $35 $23 $7D`	$+0.796\,926\,012\,598\,8 \times 10^{-1}$
`$28 $E7 $5D $A5 $80`	$-0.645\,964\,095\,264\,46$
`$A2 $DA $0F $49 $81`	$+0.157\,079\,632\,676\,16 \times 10^{1}$

Table 20: Table 3342 in Locomotive BASIC (`$31EC-$3209` memory range) [JM86, p. 378] [Har+78, p. 238].

ZX Spectrum BASIC approximates sine, arctangent, natural logarithm and exponential functions with Chebyshev series expansions and Clenshaw's summation method, as thoroughly explained by Logan and O'Hara in *The Complete Spectrum ROM Disassembly* [LO83, pp. 198–199, 211–219]. Logan and O'Hara provide notably examples of BASIC programs that illustrate how Chebyshev polynomials are used, from the different approximation routines (sine, arctangent, natural logarithm and exponential) with their associated coefficients, to the evaluation of the polynomials [LO83, pp. 222–227]. The full expansion of Chebyshev polynomials is however $T_0^*(x)$, $2 \times T_1^*(x)$, $2 \times T_2^*(x)$, $\cdots$ instead of $T_0^*(x)$, $T_1^*(x)$, $T_2^*(x)$, $\cdots$, as shown in table 21 [LO83, pp. 198–199].

The coefficients for the approximation of sine and arctangent functions can be found indirectly[51] in *Polynomial approximations to elementary functions*, and also in *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables* that cites Clenshaw [LO83, pp. 223, 226] [AS64, ch. 4.3.104, 4.4.50] [Cle54]. If the first coefficient can be found directly, a correction by a factor of $\frac{1}{2}$ is applied starting from the second coefficient, as the Chebyshev polynomials are $T_0^*(x)$, $2 \times T_1^*(x)$, $2 \times T_2^*(x)$,

[49] Again, these coefficients are not those of Taylor or Maclaurin series expansions as suggested by Janneck and Mossakowski [JM86, p. 374]. For instance, 0.434 259 751 is not an approximate value of $2 \div ln(2) \div 7 \approx 0.412\,198\,583$.

[50] The study is however incomplete, as noted by [JM86].

[51] The last coefficient for arctangent is however missing in these references.

Original Chebyshev polynomials	ZX Spectrum BASIC Chebyshev polynomials
$T_0^*(x) = 1$	$T_0^*(x) = 1$
$T_1^*(x) = x$	$2 \times T_1^*(x) = 2x$
$T_2^*(x) = 2x^2 - 1$	$2 \times T_2^*(x) = 4x^2 - 2$
$T_3^*(x) = 4x^3 - 3x$	$2 \times T_3^*(x) = 8x^3 - 6x$
$T_4^*(x) = 8x^4 - 8x^2 + 1$	$2 \times T_4^*(x) = 16x^4 - 16x^2 + 2$
$T_5^*(x) = 16x^5 - 20x^3 + 5x$	$2 \times T_5^*(x) = 32x^5 - 40x^3 + 10x$

Table 21: Full expansion of Chebyshev polynomials in ZX Spectrum BASIC [LO83, pp. 198–199, 227].

$\cdots$ [LO83, pp. 198–199]. Table 22 presents the corrected coefficients used for the approximation of sine function. All other mathematical functions are deduced (cosine, tangent, arcsine, arccosine, square root, exponentiation) [LO83, p. 198].

Original coefficients	Corrected coefficients
+1.276 278 962	+1.276 278 962
−0.285 261 569	−0.142 630 785
+0.009 118 016	+0.004 559 008
−0.000 136 587	−0.000 068 294
+0.000 001 185	+0.000 000 592
−0.000 000 007	−0.000 000 003

Table 22: Corrected Chebyshev coefficients for the approximation of sine function in ZX Spectrum BASIC [LO83, p. 223] [AS64, ch. 4.3.104] [Cle54].

Regarding BBC BASIC, approximation of mathematical functions does not rely primarily on polynomials, but on continued fractions, and this method is used for natural logarithm, arctangent, sine and exponential functions [Rus85, pp. 423–426, 429–430, 436–440]. The evaluation of the continued fraction is performed iteratively, starting from the last term. All other mathematical functions are based on these functions (common logarithm, cosine, tangent, arccosine, arcsine, exponentiation), with the notable exception of the square root function [Rus85, pp. 419–421]. While other BASIC implementations infer the square root function from the exponentiation function (itself deduced from the logarithm and exponential functions as shown in table 17), BBC BASIC approximates the square root function iteratively with Heron's iterative method, thus allowing an important gain in performance. Heron's iterative method is expressed as follows [Har+78, ch. 6.1.7]:

$$y_{n+1} = \frac{1}{2}(y_n + \frac{x}{y_n})$$

Heron's iterative method is equivalent to Newton's iterative method applied to the square root function [Har+78, ch. 6.1.5]:

$$y_{n+1} = y_n - \frac{f(y_n)}{f'_n}$$

5.6 Performance and benchmarking

During the development of the arithmetic and mathematical features of BASIC interpreters, there has been a constant concern regarding the trade-off between performance and accuracy, as 8-bit systems have very limited computing power and memory space. These technical constraints have had a significant impact on the design and development of BASIC interpreters, and the authors have adopted distinct strategies – different representations of integers and floating-point numbers, dedicated arithmetic routines for integers, and various approximation methods. The approximation of mathematical functions is particularly revealing, as the authors attempt to maximize accuracy while limiting computation, by choosing sophisticated methods and optimal coefficients. The use of Heron's iterative method to approximate the square root function, instead of using the generic and costly exponentiation function, is another illustration of this concern regarding performance. This trade-off has been highlighted by the various benchmarks set up by the industry. Indeed, as the performance of BASIC implementations on 8-bit computers was a key selling point, it became necessary to set up standard tests to compare these implementations.

In June 1977, Tom Rugg and Phil Feldman introduced in *Kilobaud Magazine* [RF77a] a series of seven short tests known as the *Rugg/Feldman benchmark*. These tests – named *BM1* to *BM7* – evaluate various features of the BASIC programming language such as loops (*BM1*, *BM6*, *BM7*), jumps (*BM2* to *BM7*), arithmetic operations on integers (*BM2* to *BM7*), subroutine calls (*BM5*, *BM6*, *BM7*) and operations on arrays (*BM7*). The results of the benchmark showed that the Integer BASIC was significantly faster than all other BASIC implementations. The results sparked many comments, including a letter from Bill Gates arguing that – as only integer computation was evaluated – it was expected that Integer BASIC would be faster than other BASIC implementations that support by default floating-point numbers [RF77b]. This led John Coll to publish in *Personal Computer World* an extra test called *BM8* to evaluate floating-point support and transcendental functions [Col78b]. Updated results were published in November 1978 [Col78a]. These results confirmed the performance of Integer BASIC on the first seven tests. However, Integer BASIC was not able to perform *BM8* due to the lack of native support of floating-point numbers. The results also showed that Integer BASIC was significantly faster on tests *BM2* to *BM7* than Applesoft BASIC, its successor on the Apple II series of computers. As mentioned in section 5.3, the performance gap observed is related to the fact that Applesoft BASIC does not provide dedicated integer arithmetic routines.

In 1981, Jim Gilbreath published in *Byte* magazine a programming language performance benchmark based on the sieve of Eratosthenes algorithm [Gil81]. The implementation was provided in various programming languages such as Pascal, C, Fortran and BASIC. The test stresses memory references, structured control statements and I/O operations on a useful problem, i.e., the determination of prime numbers. The test does not involve floating-point arithmetic as only computation of integers is required. More complete results were published in 1983 [GG83]. Once again, the test highlights the superior performance of Integer BASIC over Applesoft BASIC for integer arithmetic [GG83, p. 294]. Another key takeaway is that the fastest implementations of the sieve of Eratosthenes are the one written in assembly language.

In 1983, David Ahl introduced in *Creative Computing magazine* a new benchmark for BASIC implementations [Ahl83]. The six lines long program evaluates loops, mathematical functions (square root, square) and the random number generator. Three metrics are measured: execution time, accuracy of floating-point support and the

quality of the random number generator. The main finding of the test is that the fastest implementations are not the most accurate. Interestingly, the measure of the accuracy reveals BASIC interpreters that share a common floating-point implementation. Indeed, the accuracies computed for the VIC-20, C64, Apple II Plus and Apple IIe are identical[52], which is not surprising as Commodore BASIC and Applesoft BASIC both derive from the Microsoft 6502 BASIC. The benchmark was updated in 1984 and more complete tests were published[Ahl84].

5.7 Conclusion

Our study of the implementation of arithmetic and mathematical features in BASIC interpreters has highlighted several points. First, the complexity of the algorithms involved, especially with regard to the approximation of mathematical functions. This complexity is a major challenge for programmers, especially in the context of a fast-moving industry with short delivery deadlines. Second, the variety of choices made, which results in very different implementations. This diversity of implementations eventually reflects a lack of consensus among the different programmers on the best methods to adopt. This has led the software and hardware industries to adapt. First, the *ANSI/IEEE Standard 754-1985* standard was published in 1985, to ensure the quality of floating-point implementations [The85]. Then, hardware manufacturers have gradually introduced advanced mathematical features into microprocessors, on one hand to facilitate the work of programmers, and on the other hand for performance and reliability reasons. For instance, if we consider the Motorola 68000 family, the multiplication and division of signed and unsigned integers were introduced in the Motorola 68000 microprocessor (`MULU`, `MULS`, `DIVU` and `DIVS` instructions [Mot88, p. 3-3]), and the first implementation of the *ANSI/IEEE Standard 754-1985* standard appeared in the Motorola 68881 floating-point coprocessor [Mot88, p. 4-33] (refer to [Mot88, p. 4-41] for the list of monadic and dyadic operations supported).

Our study also showed that, despite the differences, the different BASIC implementations yield a significant numerical accuracy. There are however situations where the programmer will favor performance over accuracy, and thus cannot use the provided routines that are too costly. It is therefore up to the programmer to find an alternative. In this context, where performance prevails over accuracy, we will study in the next section techniques based on precomputed numeric tables.

6 Precomputed tables

6.1 Overview

Mathematical tables have played an essential role in the development of science, and have been used in many fields of application: arithmetic, geometry, physics, astronomy, navigation and geography, to name a few. These tables were still widely used in the twentieth century, until the advent of electronic calculators and computers made their usage obsolete. Due to the variety of tables and the multiplicity of sources, efforts have been made to catalogue and document these tables. The *Report of the Committee on Mathematical Tables* published in 1873 is a notable example of an early comprehensive catalogue on this subject, with an in-depth study and historical development of 25 distinct types of mathematical tables [Adv73]. The report, however,

[52] Measured accuracy is 0.001 041 423 5.

does not provide the tables. The *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables* published in 1964 was conceived as a compendium of mathematical tables for hand computation [AS64]. As such, it provides numerical tables for different mathematical functions with an accuracy of at least 5 significant figures, and tabular intervals have been chosen to ensure an accuracy of 4 or 5 significant figures for linear interpolation [AS64, p. IX]. Different interpolation methods are also described, such as Lagrange interpolating polynomial and Aitken's iteration method [AS64, ch. 25.2]. We describe the Lagrange interpolating polynomial, as it is a widely used interpolation method.

Given a set of $n+1$ pairs $\{(x_0, y_0 = f(x_0)), \cdots, (x_n, y_n = f(x_n))\}$, the Lagrange interpolating polynomial $P_n(x)$ of degree n that approximates $f(x)$ is expressed as follows:

$$P_n(x) = \sum_{i=0}^{n} y_i l_i(x)$$

$$l_i(x) = \frac{(x - x_0) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n)}{(x_i - x_0) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_n)}$$

Lagrange interpolating polynomials $P_1(x)$ and $P_2(x)$ correspond respectively to linear and quadratic interpolation, and are defined as follows:

$$P_1(x) = y_0 \frac{(x - x_1)}{(x_0 - x_1)} + y_1 \frac{(x - x_0)}{(x_1 - x_0)}$$

$$P_2(x) = y_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + y_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + y_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

Mathematical tables are discussed by von Neumann in *First Draft of a Report on the EDVAC* [Neu93, ch. 10.3, 10.4]. Von Neumann argues that the use of mathematical tables with interpolation is an efficient alternative to series expansions. However, as linear and quadratic interpolation do not provide sufficient accuracy, it is necessary to use at least cubic interpolation to achieve satisfactory accuracy. Von Neumann also mentions that multiplication should not be performed with mathematical tables, as interpolation requires multiplication.

In *Computer Approximations*, Hart et al. identify a case where the use of mathematical tables is relevant, and this is "when the need for speed of evaluation outweighs considerations of storage and high precision" [Har+78, ch. 2.7]. Hart et al. add that for some functions it is possible to reduce storage and increase accuracy at the cost of additional computations by using *radix tables*. These reduced size tables can be combined to produce the desired function value. For instance, the exponential function e^{x+y} can be expressed as $e^x e^y$, which allows to reduce the range of the mathematical table used to approximate the exponential function. Another advantage of using tables is that it can be used for the desired function and its inverse. Thus, a table storing values of the exponential function may be also used to approximate the logarithm function.

In this section, we will present how precomputed tables are used to improve performance at runtime. Firstly, we will introduce methods based on single entry mathematical tables to perform fast multiplication and division (logarithm and antilogarithm tables, quarter-square table), in lieu of traditional algorithms (shift-and-add, shift-and-subtract). Secondly, we will present a specific case of precomputed table known as *screen memory addresses table*, a well known optimization technique in the context of real-time graphics.

6.2 Multiplication and division with single entry tables

Multiplication is such a common operation that many methods have been proposed [Adv73, pp. 15–20]. The traditional table-based method requires a double entry multiplication table, where the two arguments are the multiplicand and the multiplier. If the result of the multiplication can be given at once, the room needed to store the double entry table is an important issue, as the growth is quadratic. Similarly, division methods based on double entry tables have been proposed. To address this issue, alternative methods based on single entry tables have been introduced, such as multiplication and division with logarithm and antilogarithm tables, and multiplication with a quarter-square table. If the calculation is a little more complex, the linear growth of the single entry table allows to store many more values. These two methods are presented below.

6.2.1 Logarithm and antilogarithm tables

The logarithm function was introduced in 1614 by John Napier as a means to simplify long, tedious and error-prone multiplications and divisions of large numbers necessary in fields such as astronomy, engineering and navigation [Nap14]. To achieve this, Napier had to create a function that would establish a correspondence between an arithmetic progression and a geometric progression. Working closely with Napier, Henry Briggs published the first table of decimal logarithms[53], giving the logarithms of the numbers from 1 to 1000 with a precision of 14 digits [Bri17] [Adv73, p. 49] [Roe11]. Subsequently, tables with increasing scope and different precision were published [Bri24] [Vla28].

The logarithm function for a given base b where $b > 0$ and $b \neq 1$ and for $x > 0$ is defined as the inverse function of b^x, and b^x is the antilogarithm of x in base b:

$$x = log_b(b^x)$$
$$antilog_b(x) = b^x$$

The following properties regarding multiplication and division can be derived from the definition of the logarithm function:

$$log_b(x \times y) = log_b(x) + log_b(y)$$
$$log_b(x \div y) = log_b(x) - log_b(y)$$

Leading to:

$$x \times y = antilog_b(log_b(x) + log_b(y))$$
$$x \div y = antilog_b(log_b(x) - log_b(y))$$

Thus, a multiplication requires two logarithm table lookups for the multiplicand and the multiplier, an addition, and then an antilogarithm table lookup to find the product. Similarly, a division requires three table lookups and a subtraction.

The use of logarithm and antilogarithm tables is notably mentioned by von Neumann, to perform multiplication, division and square root calculations [Neu93, ch. 10.3]. However, von Neumann does not recommend it for the multiplication – as table-based methods should use interpolation to yield satisfactory accuracy, and interpolation requires multiplication.

[53] Also called Briggsian logarithms.

This technique is illustrated in Elite, a seminal video game written by Ian Bell and David Braben, and released initially on the BBC Micro model B and Acorn Electron. While the *standard versions* of Elite (BBC Micro model B, Acorn Electron) use classic shift-and-add and shift-and-subtract algorithms (as there is not enough memory to store the tables), the *advanced versions* (BBC Master, BBC Micro model B with a 6502 Second Processor) take advantage of the extra memory available on these systems to store logarithm and antilogarithm tables to perform multiplication and division, and thus speed up computation [Mox20a] [Mox20b] [Mox20c] [Mox20d]. Performance at runtime is the main concern, and as the question of accuracy is secondary, interpolation is not necessary, and as such multiplication can be implemented with logarithm and antilogarithm tables. Table 23 lists the routines that can be found in the source code of the different versions of Elite on Acorn computers.

Method	**Routines**
Table-based multiplication	`FMLTU`
Table-based division	`LL28`
Shift-and-add multiplication	`MULT1`, `MU11`
Shift-and-subtract division	`TIS1`, `TIS2`, `DVID4`

Table 23: Multiplication and division routines in Elite.

The logarithm and antilogarithm tables are generated at runtime in BBC BASIC. The 16-bit logarithm table is split in two 256 bytes tables `log` and `logL` that store respectively the high-order bytes and low-order bytes of a logarithmic growth, while the 8-bit antilogarithm table `ANTILOG` is a 256 bytes table that contains an exponential growth [Mox20a]. Figure 12 illustrates how the 16-bit logarithm table is generated at runtime. According to Braben, the use of logarithm and antilogarithm tables has a dramatic impact on performance [Bra11, 26:02–27:26].

```
.log
SKIP 1
FOR I%, 1, 255
B% = INT(&2000*LOG(I%)/LOG(2)+0.5)
EQUB B% DIV 256
NEXT
.logL
SKIP 1
FOR I%, 1, 255
B% = INT(&2000*LOG(I%)/LOG(2)+0.5)
EQUB B% MOD 256
NEXT
```

Figure 12: Generation of the 16-bit logarithm table in advanced versions of Elite [Mox20a].

6.2.2 Quarter-square table

The quarter-square multiplication is a method that requires only one table of single entry. This method relies on the following identity, where q is the quarter-square function [Adv73, pp. 21–25]:

$$x \times y = q(x+y) - q(x-y) \quad q(w) = \frac{w^2}{4}$$

The first[54] known mention of a multiplication method of x by y that uses squares of $x+y$ and $x-y$ is by Hiob Ludolf in 1690 in *Tetragonometria tabularia* [Lud90, ch. V] [Adv73, pp. 22, 27] [Roe13a]. Pierre-Simon Laplace also mentions the method in 1809 [Lap09]. In 1817, Antoine Voisin and Johann Anton Philipp Bürger published independently the first known tables of quarter-squares [Roe13b]. Both Voisin and Bürger provide only the truncated quarter-squares as it is not necessary to provide the decimal part in the context of multiplication [Roe13b, p. 3] [Adv73, pp. 21–22]. Indeed, if x and y are both odd or both even, then $x+y$ and $x-y$ are both even, and $(x+y)^2$and $(x-y)^2$ are divisible by 4 as the square of an even number can be written $(2n)^2 = 4n^2$. If x is even and y is odd or x is odd and y is even, then $x+y$ and $x-y$ are both odd, and $(x+y)^2$ and $(x-y)^2$ are both odd as the square of an odd number can be written $(2n+1)^2 = 4n^2+4n+1$. We can then deduce that both decimal parts ($+\frac{1}{4}$ and $-\frac{1}{4}$) are not necessary to perform the multiplication:

$$q(x+y) - q(x-y) = \frac{(2m+1)^2}{4} - \frac{(2p+1)^2}{4}$$
$$= m^2 + m + \frac{1}{4} - p^2 - p - \frac{1}{4} = m^2 + m - p^2 - p$$

The quarter-square multiplication can be rewritten:

$$x \times y = \lfloor q(x+y) \rfloor - \lfloor q(x-y) \rfloor$$

From a computational point of view, the quarter-square multiplication requires an addition and a subtraction of the multiplicand and multiplier, two table lookups and a final subtraction. If in terms of complexity this method is comparable to the one based on logarithm and antilogarithm tables, it however yields a higher precision [Adv73, p. 22].

A first known implementation of the quarter-square multiplication for the MOS 6502 is proposed by Charles Putney in 1986 [Put86]. The routine takes as an input two 8-bit unsigned numbers and returns a 16-bit unsigned product. It uses a table of 512 16-bit truncated quarter-squares, and the table is split in two 512 bytes tables (`SQH`, `SQL`) that respectively store the high-order and low-order bytes of the 16-bit truncated quarter-squares. Both tables are aligned to a 256-byte page boundary to avoid page boundary crossing penalty cycles.

Faster implementations exist on the MOS 6502, such as the one proposed by Jackasser* presented in figure 13 [Jac15]. As previously, the routine takes as an input two 8-bit unsigned numbers (`T1`, `T2`) and returns a 16-bit unsigned product (`PRODUCT`).

[54] Several sources [OR00][Der06, p25] attribute this method to the Babylonians as Babylonians were familiar with tables of squares. There is however no documents to support this assertion, and this is merely speculation. It is more likely that multiplication that could not be done mentally was performed with an instrument [Høy02] [Pro08, pp. 178–179] [C. Proust, personal communication, June 30, 2021].

It however leverages additional tricks to reduce the number of instructions, and in particular limit the number of arithmetic operations required (addition, subtraction) by combining indexed addressing and self-modifying code[55]. Two 16-bit tables are used: the table of 16-bit truncated quarter-squares $\lfloor \frac{w^2}{4} \rfloor$, and an extra table of 16-bit $\lfloor \frac{(w-255)^2}{4} \rfloor$ values, with $0 \leq w \leq 511$. This extra table allows to compute $\lfloor q(x-y) \rfloor$ without the need to perform a subtraction. Both tables are split in two high-order bytes and low-order bytes tables – `SQUARE1_HI` and `SQUARE1_LO` for the truncated quarter-squares table, `SQUARE2_HI` and `SQUARE2_LO` for the extra table. As previously, these tables are aligned to a 256-byte page boundary to prevent page boundary crossing penalty cycles. The page alignment also allows a self-modifying code trick: the low-order byte of the absolute addresses at `SM1+1` and `SM3+1` is updated with `T1`, and the low-order byte of the absolute addresses at `SM2+1` and `SM4+1` is updated with `T1`'s one complement[56]. Operations on the tables are then performed by leveraging *Absolute, X* addressing mode with `T2` as an index. In the end, only two 8-bit subtractions are required (see `SM2` and `SM4`), whereas theoretically the algorithm requires four 8-bit subtractions and two 8-bit additions. Refer to Graham* for examples of runtime table generators [Gra16].

```
        LDA  T1
        STA  SM1+1
        STA  SM3+1
        EOR  #$FF
        STA  SM2+1
        STA  SM4+1
        LDX  T2
        SEC
SM1:    LDA  SQUARE1_LO,X
SM2:    SBC  SQUARE2_LO,X
        STA  PRODUCT+0
SM3:    LDA  SQUARE1_LO,X
SM4:    SBC  SQUARE2_HI,X
        STA  PRODUCT+1
```

Figure 13: Quarter-square multiplication [Jac15].

6.3 Screen memory addresses tables

The general principle of precomputing values is not limited to arithmetic and mathematical functions, it is indeed a classic optimization technique that trades space for time [Ben82, pp. 40–43]. This technique is notably used in real-time bitmap graphics, to determine the screen memory address associated with a point of (x, y) integer coordinates, where the screen is considered as a plane with $(0, 0)$ being the upper left corner. As the screen memory layout of bitmap graphic modes on many 8-bit systems is not linear, it can be quite costly to compute the screen memory address associated

[55] Self-modifying code will be discussed in section 7.4. Note that this technique requires the code to be executed from RAM.

[56] `T1`'s one complement is required for the extra table (`SQUARE2_HI`, `SQUARE2_LO`).

to a given point, which for instance can prevent to achieve fast and smooth animation in video games on these systems.

A first illustration is the Amstrad CPC, based on the Motorola 6845 CRT[57] controller that was originally designed for character-based displays [GOR86, p. 1.1] [Mot84c]. The Amstrad CPC has a non-linear screen memory layout that reflects the original purpose of the CRT controller, i.e., display characters. The standard width of the screen memory is 80 bytes, and the standard height is 200 pixel lines – or 25 character rows, as there are 8 lines per character row [GOR86, p. 6.4]. The format of a screen memory byte depends on the graphic mode selected – *Mode 0*, *Mode 1* or *Mode 2*. In Mode 1, as there are 320 pixels per line, a screen memory byte stores 4 pixels, and thus there are 2 bits per pixel. By default, the screen location starts at `$C000`. Whereas the screen memory address is linear along the x axis, this is not the case for the y axis, as shown in table 24. The screen memory is indeed divided into eight blocks of 2 KB, with one block for each of the 8 lines of the character rows: block 0 running from `$C000` to `$C7FF` stores pixels for the first line of the character rows, block 1 running from `$C800` to `$CFFF` stores pixels for the second line of the character rows, and so on. This particular memory layout explains the *venetian blind* effect observed by Amstrad CPC users during the loading of many games from tape[58] as writing data linearly in the screen memory address starting at `$C000` will fill consecutively line 0 (`$C000-$C04F`), line 8 (`$C050-$C09F`), line 16 (`$C0A0-$C0EF`), and so on.

Line	Address range
0	$C000-$C04F
1	$C800-$C84F
...	...
8	$C050-$C09F
9	$C850-$C89F
...	...
199	$FF80-$FFCF

Table 24: Amstrad CPC screen memory layout.

The Amstrad CPC ROM provides a routine (`SCR DOT POSITION`) that computes the screen memory address of a pixel of coordinates x and y [GOR86, p. 15.99]. However, the routine is very costly as it is 56 instructions long [JM86, pp. 254–255]. To minimize the cost of finding the start memory address of a line, Francis Pierot proposed in 1986 to use a lookup table where the 16-bit start memory addresses of each line are stored sequentially [Pie86, pp. 130, 236–237]. In 2001, Zik* proposed a faster implementation where the 16-bit lookup table is split in two high-order byte and low-order byte lookup tables, allowing to find the start memory address of a line in only five instructions, as shown in figure 14 [Zik01]. Both tables are aligned to a 256-byte page boundary and are stored sequentially, i.e., the low-order byte table starts at `TABLE` label and the high-order byte table starts at `TABLE+256`. The routine expects the line number in L register and returns the memory address in HL register:

Another example of non-linear memory layout is the high-resolution (*Hi-Res*) graphic mode of the Apple II [App78a, pp. 19, 21] [Sta82, ch. 4]. This graphic mode takes its data from two separate 8 KB blocks called *picture buffers*: the primary page

[57] Cathode-ray tube.

[58] Batman, Dragon Ninja and Head Over Heels to name a few.

```
LD H,TABLE/256
LD A,(HL)
INC H
LD H,(HL)
LD L,A
```

Figure 14: Line start memory address lookup on Amstrad CPC [Zik01].

picture buffer runs from `$2000` to `$3FFF`, followed by the secondary page picture buffer from `$4000` to `$5FFF`. The purpose of these two buffers is to achieve flicker-free graphics with the page flipping technique. This graphic mode can display 192 lines, with 40 bytes per line. A byte stores 7 pixels, where the most significant bit selects the color of the byte, and the remaining bits are an on/off bit for each pixel, hence a total of 280 pixels per line. If the memory layout along the x axis is linear, this is not the case along the y axis, as shown in table 25. If we write data linearly to the `$2000` primary page, then the following lines will be filled: line 0 (`$2000-$2027`), line 64 (`$2028-$204F`), line 128 (`$2050-$2077`), line 8 (`$2080-$20A7`), line 72 (`$20A8-$20CF`), line 136 (`$20D0-$20F7`), and so on.

Line	Address range
0	$2000-$2027
1	$2400-$2427
...	...
8	$2080-$20A7
9	$2480-$24A7
...	...
64	$2028-$204F
65	$2428-$244F
...	...
128	$2050-$2077
...	...
191	$3FD0-$3FF7

Table 25: Apple II Hi-Res graphic mode primary page memory layout [App78a, pp. 19, 21] [Sta82, ch. 4].

Different implementations to compute the start memory address of a line have been proposed. A classic implementation (`HPOSN`) is provided by Wozniak in the Applesoft ROM [App78c, p. 83]. Later on, Jeffrey Stanton introduced a much faster version based on two 192 bytes long lookup tables (`YVERTH`, `YVERTL`) that store respectively the high-order byte and low-order byte of each memory address [Sta82, pp. 111, 137–139]. As usual, both tables are aligned to a 256-byte page boundary. The classic game Prince of Persia, developed by Jordan Mechner and released originally on the Apple II in 1989, uses these tables – the tables are however named differently (`YHI`, `YLO`). Figure 15 presents the content of `YHI` table found in Prince of Persia original source code, where the high-order byte of each line starting memory address can be easily identified – `$20` for line 0, `$24` for line 1, and so on [Mec89, HRTABLES.S].

```
hex 2024282C3034383C2024282C3034383C
hex 2125292D3135393D2125292D3135393D
hex 22262A2E32363A3E22262A2E32363A3E
hex 23272B2F33373B3F23272B2F33373B3F

hex 2024282C3034383C2024282C3034383C
hex 2125292D3135393D2125292D3135393D
hex 22262A2E32363A3E22262A2E32363A3E
hex 23272B2F33373B3F23272B2F33373B3F

hex 2024282C3034383C2024282C3034383C
hex 2125292D3135393D2125292D3135393D
hex 22262A2E32363A3E22262A2E32363A3E
hex 23272B2F33373B3F23272B2F33373B3F
```

Figure 15: `YHI` lookup table in Prince of Persia on Apple II [Mec89, HRTABLES.S].

The same problem has been reported on other systems, and lookup table-based solutions have been proposed. All graphics modes[59] of the BBC Micro model B have a non-linear memory layout [BDH83, pp. 462-477]. Like the Amstrad CPC, the BBC Micro model B uses the Motorola 6845, intended for character display [Mot84c]. Ian O. Angell and Brian J. Jones propose in *Advanced Graphics with the BBC Model B Microcomputer* to use a lookup table that contains the high-order byte of the start memory address of each of the 32 text rows, the table being generated at runtime in BBC BASIC [AJ83a, pp. 257, 260].

The ZX Spectrum *display file* is also non-linear [Jam84, pp. 76–79]. Again, Angell and Jones propose to use lookup tables to address this issue [AJ83b, pp. 212–214]. Two tables are generated at runtime in ZX Spectrum BASIC: one for the high-order byte and one for the low-order byte of the 16-bit address of each display file line. Manic Miner and Jet Set Willy, two iconic ZX Spectrum games developed by Matthew Smith, implement a single 16-bit lookup table. In Manic Miner, a back memory buffer (`$6000-$6FFF` memory range) is used to draw all the graphic elements. Graphic elements drawing routines (`$8944`, `$8DF8`, `$8E75`, `$8EF1`, `$9135` and `$927F` memory locations) use this table (`$8300-$83FF` memory range) to draw into the back memory buffer [Dym20b]. When all the graphic elements are drawn, the content of this back memory buffer is then copied to the display file (`$4000-$4FFF` memory range) with an `LDIR` instruction (`$87AB` memory location). Figure 16 shows a sample of this lookup table, after the disassembly of the game by Richard Dymond with SkoolKit. The analysis of Jet Set Willy leads to similar conclusions [Dym20a].

Regarding the C64, the high-resolution (*HiRes*) and multi-colour bitmap modes have a non-linear memory layout. A.P. Stephenson and D.J. Stephenson propose in *Advanced machine code programming for the Commodore 64* routines to compute the memory address of a given pixel [SS84, ch. 7]. These routines are however quite costly. TWW* proposes a method based on four lookup tables for the high-resolution bitmap mode, and for the specific case where $0 \leq x \leq 255$. Two 8-bit tables store

[59] Except Mode 7.

```
$8300 DEFW $6000
$8302 DEFW $6100
...
$8310 DEFW $6020
$8312 DEFW $6120
...
$8320 DEFW $6040
$8322 DEFW $6140
...
$83FC DEFW $6EE0
$83FE DEFW $6FE0
```

Figure 16: Lookup table in Manic Miner on ZX Spectrum [Dym20b].

the high-order byte and low-order byte of the 16-bit address of line y, an 8-bit table is used for x and another 8-bit table for the associated bitmask [TWW16]. Several methods are proposed to generate the tables. In the first method, tables are generated at compilation time with Kick Assembler [Nie] script language, as shown in figure 17. This simple method results however in a significantly larger compiled file[60]. In the second method, a 54 bytes long machine code routine generates the tables at runtime, and thus eliminates the need to store the tables in the compiled file. Several options are also discussed for the generic case where $0 \leq x \leq 319$.

```
.align $100
BitMask:
.fill 256,pow(2,7-i&7)
X_Table:
.fill 256,floor(i/8)*8
Y_Table_Hi:
.fill 200,>GFX_MEM+[320*floor(i/8)]+[i&7]
.align $100
Y_Table_Lo:
.fill 200,<GFX_MEM+[320*floor(i/8)]+[i&7]
```

Figure 17: Generation of lookup tables for the HiRes bitmap mode on C64 with Kick Assembler [TWW16].

6.4 Conclusion

The use of precomputed tables is a major optimization technique on 8-bit computers, and a classic example of time-space trade-off. To this end, careful tuning must be done to improve performance at runtime while still fitting into the system's limited memory, as illustrated by the differences between the standard and advanced versions of Elite on Acorn computers (see section 6.2.1). Significant performance gains can also

[60] A file can be compressed though, as discussed in section 7.3.

be achieved by performing low-level optimizations – techniques that will be discussed in the next section.

7 Low-level optimization techniques

7.1 Overview

When 8-bit microprocessors were introduced, the manufacturer's official documentation was an essential knowledge base to start with. The programmer could also refer to specialized magazines or books available, to learn how to program, or to deepen his knowledge on certain subjects, e.g, program optimization. Both written by Zaks, *Programming the 6502* and *Programming the Z80* are classic books, considered by many programmers as important educational texts to learn assembly language programming on these microprocessors, as well as indispensable reference guides [Zak78] [Zak80]. Other books were dedicated to more specific matters. *Writing efficient programs*, written by Jon Louis Bentley and first published in 1982, is a reference book on program optimization widely used by programmers on many different systems [Ben82]. *Compilers: Principles, Techniques, and Tools*, written by Aho et al. and published in 1986, is a classic computer science book on the topic of compilation, and provides very valuable information to the assembly language programmer [ASU86]. Programming techniques on 8-bit microprocessors have however evolved over time, even after the end of the commercial life of 8-bit systems. Indeed, the knowledge of microprocessors has improved, and some undocumented features have been discovered. Computer enthusiasts and hobbyists communities – such as the demoscene – also contributed to the emergence – and generalization to some extent – of low-level optimization techniques. In this section, we will study such techniques. We will first give an overview of optimization techniques as provided by classic sources (manufacturer's documentation [MOS76b] [Zil76], *Writing efficient programs* and *Compilers: Principles, Techniques, and Tools* books [Ben82] [ASU86]). We will then present three low-level programming techniques[61] used to maximize runtime performance on 8-bit microprocessors. It is not uncommon for these techniques to be used together, as the examples studied will show. The first technique is *loop unrolling*, and proves to be an important – if not the main – source of performance improvement, and relies on trading space for time. Loop unrolling can be performed statically – before the compilation of the program – and this technique is well documented in classic sources [Ben82, pp. 56–59]. Loop unrolling can also be done at runtime, and in this case the code is generated during the execution of the program. The second technique is *self-modifying code*, where the program modifies its own flow of instructions during its execution. The third technique relies on *undocumented instructions*. This technique is specific to each microprocessor, and consists in exploiting the side effects of unofficial instructions. Apart from static loop unrolling, all these techniques – for the purpose of optimization – seem to have emerged over time.

[61] We present three low-level optimization techniques, but there exist many other programming *tricks* to speed up execution at runtime. Readers looking for a more comprehensive list of these optimization tricks on the MOS 6502 and Z80 may refer to other sources such as [Bit22] and [Wil20].

7.2 Classic sources

The manufacturer's documentation provides guidance to the programmer in the context of performance. On the MOS 6502, good programming practices include leveraging the zero page to store frequently accessed data, and using index registers and indexed addressing modes to process blocks of data sequentially [MOS76b, pp. 5–6]. On the Z80, it is recommended to use general purpose registers when possible as registers accesses are significantly faster than memory accesses, relative addressing to jump to nearby locations, index registers and indexed addressing mode to process blocks of data sequentially, and block instructions when relevant [Zil76, pp. 20–22, 26–28, 35–37]. The technical documentation published by the manufacturers also gives very useful information for each instruction of the instruction set: the number of cycles required to execute the instruction, the number of bytes of the instruction, and for certain instructions specific conditions where the instruction requires one or several extra cycles for its execution [MOS76b, p. 6] [Zil76, pp. 43–54]. Several instructions of the MOS 6502 (`ADC`, `AND`, `CMP`, `EOR`, `LDA`, `LDX`, `LDY`, `SBC`) require one extra cycle if a page boundary is crossed, and branch instructions of the MOS 6502 (`BCC`, `BCS`, `BEQ`, `BMI`, `BNE`, `BPL`, `BVC`, `BVS`) require one extra cycle if a branch occurs to the same page, and two extra cycles if a branch occurs to another page. On the Z80, `JR cc,e` instructions require an extra M cycle if the condition is true, while `CALL cc,e` and `RET cc` instructions require two extra M cycles if the condition is true [Zil76, pp. 53–54]. Therefore, good programming practices also include a careful arrangement of the memory, and organizing the source code control flow to limit branches.

Additionally, *Writing efficient programs* presents programming best practices with performance in mind [Ben82]. Bentley emphasizes the fact that performance relies above all on the proper choice of data structure and algorithms, and that optimization at the source code level shall be addressed last [Ben82, pp. 31–38]. The trade-off between time and space is also discussed. Techniques that trade space for time (data structure augmentation, storage of precomputed results, caching, lazy evaluation) and time for space (packing, code interpretation) are introduced [Ben82, pp. 39–49]. Optimization techniques at the source code level are also presented, and are divided in four sets of rules: *loop rules*, *logic rules*, *procedure rules* and *expression rules* [Ben82, pp. 51–85]. As programs spend most of their time in loops, it is necessary to eliminate unnecessary computation within the loops. Examples of loops rules are: eliminating repeated computation (*Code motion out of loops* rule), reducing the number of tests (*Combining tests* rule), reducing the cost of indexing (*Loop unrolling* rule) and sharing the cost of loop overhead (*Loop fusion* rule) [Ben82, pp. 51–66]. Logic rules relate to reducing the cost of logical and mathematical computations. For instance, algebraic identities can be exploited (*Exploit algebraic identities* rule) to perform *strength reduction*, and a logical or mathematical function over a small finite domain may be replaced by a lookup table[62] (*Precompute logical function* rule) [Ben82, pp. 66–75]. Procedure rules aim at optimizing the underlying structure of the program organized into procedures. For example, the inline expansion of a procedure increases performance by eliminating the cost overhead of the procedure call, to the expense of space (*Collapsing procedure hierarchies* rule). Rules to reduce the cost of recursive procedures (*Transformations on recursive procedures* rule) and exploit multiprocessor architecture (*Parallelism*) are also discussed [Ben82, pp. 75–81]. Finally, expression rules aim at optimizing the evaluation of expressions. For instance, constants should be declared

[62] As discussed in section 6.

and initialized to allow *constant folding*[63] at compile time (*Compile-time initialization* rule), and common subexpressions should be identified and eliminated[64] (*Common subexpression elimination* rule) [Ben82, pp. 81–85].

The question of code performance has also been studied in the field of compilation, i.e., the translation from a high-level programming language source code into machine code before the execution of a program. Conceptually, a compiler operates in phases, where each phase transforms one representation of the high-level source code into another representation. Typical successive phases of the compilation process are lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and code generation [ASU86, pp. 10–15]. After the analysis phases, the compiler generates an intermediate representation of the source code. An example of this intermediate representation is the *three-address code* representation, where each instruction has at most three operands [ASU86, pp. 464–473]. This abstract representation is suitable for code optimization and code generation. During the code optimization phase, the compiler attempts to improve the intermediate source code, as illustrated in table 26. The goal is to speed up the program or reduce its memory footprint, or ideally both. The code optimization typically consists of three steps: the control-flow analysis, the data-flow analysis and the source code transformations. Many source code transformations may be applied such as common subexpression elimination, code motion, induction variable-elimination and strength reduction. In particular, loops are very good candidates for code optimization [ASU86, pp. 585–598].

Original sequence	**Optimized sequence**	**Transformations**
`c := a + b` `d := a + b`	`c := a + b` `d := c`	Common subexpression elimination
`x := x * 2`	`x := x + x`	Strength reduction

`a := x ** 2`
`b := 3`
`c := x`
`d := c * c`
`e := b * 2`
`f := a + d`
`g := e * f`
	`a := x * x`
	`f := a + a`
	`g := 6 * f`
		Strength reduction, copy propagation, constant folding, common subexpression elimination and dead code elimination [KA, p29–41]

Table 26: Examples of *three-address code* intermediate code optimization [ASU86].

Finally, during the code generation phase, the compiler produces absolute machine code, relocatable machine code or assembly language source code for the target microprocessor. The code generation phase depends on the target microprocessor, i.e., the instruction set, the different addressing modes and registers available. The knowledge of the costs associated to each instruction and addressing mode is also an important factor to generate quality code [ASU86, pp. 513–521]. As the generated code may

[63] Constant folding is performed by compilers on the intermediate code during the code optimization phase [ASU86, p. 595]

[64] Common subexpression elimination is a classic transformation performed by compilers on the intermediate code during the code optimization phase [ASU86, pp. 592–594]

be far from being optimal, a final optimization specific to the target microprocessor and known as *peephole optimization* is usually performed [ASU86, pp. 554–557]. The peephole is a small moving window on the generated code that is used to perform local transformations. Examples of transformations are redundant instructions elimination, control-flow optimization, strength reduction or the use of specific machine idioms. It is recommended to perform several successive passes on the code as the transformations can spawn new optimization opportunities. There is however no guarantee that the final generated code is optimal.

7.3 Loop unrolling

Programs spend most of their time in loops, and as such loops are the most promising sources of performance improvement in programs [Ben82, p. 51] [AU77, p. 441]. In the case of short loops, a large cost is related to modifying the loop indices, and this cost can be reduced by *unrolling the loop*, i.e., replicating the body of the loop and adjusting the number of loop iterations [Ben82, pp. 56-59] [AU77, pp. 471-472]. This optimization is a classic time-space trade-off. The fact that programs spend most of their time in loops – and more generally in small sections of code, a phenomenon known as *locality of reference* or *locality* – had a significant impact on the evolution of the design of microprocessors. A good illustration is the Motorola 68000 family. The Motorola 68010 released in 1982 is the successor of the Motorola 68000, the first microprocessor of the Motorola 68000 family. The Motorola 68010 supports a *loop mode* that allows faster execution of *tight loops* [Mot88, p. 3.62] [MM83]. More specifically, the Motorola 68010 recognizes a *tight loop sequence* when the destination of the branch instruction falls within the length of the microprocessor's instruction pipeline, which corresponds to a maximal displacement value of -4. When the tight loop is detected, the Motorola 68010 alters its normal execution activity to take advantage of its internal instruction pipeline, thus eliminating unnecessary instruction fetching from memory and reducing by almost 50% the tight loop execution time. An illustration of the loop mode is the code that initializes an array of *N+1* 32-bit elements, as shown in figure 18. On a Motorola 68010, the cost of a loop iteration is 14 cycles, while on the Motorola 68000 the cost is 30 cycles.

```
        LEA  ARRAY,A0
        MOVE.W  #N,D0
LOOP:   CLR.L  (A0)+
        DBRA  D0,LOOP
```

Figure 18: Example of tight loop (Motorola 68010) [MM83].

Its successor, the Motorola 68020, drops the loop mode of the Motorola 68010 and instead proposes a 256-byte on-chip instruction cache placed between the microprocessor and the main memory, and organized in 64 32-bit entries [Mot88, p. 3.86]. The performance gain is achieved in two ways. First, the cache is accessed in 2 cycles, whereas a minimum of 3 cycles is required to access the main memory. Secondly, the cache allows instruction fetches and operand accesses to proceed in parallel. Its successor, the Motorola 68030, proposes a 256-byte on-chip instruction cache and an additional 256-byte on-chip data cache [Mot88, p. 3.108]. The Motorola 68030

also provides support for *burst filling*[65] of the instruction and data caches, improving throughput furthermore. Whenever a *cache miss* occurs, burst filling allows to fill four 32-bit cache entries in only 1 cycle. It should be noted that the generalization of small single or multiple level memory caches in modern microprocessors has reintroduced the cost of code size in the time-space trade-off [KEH93, ch. 3.1].

On the MOS 6502 and Z80, the issue of locality was partially addressed. As loop sequences are considered critical, both microprocessors support index registers and indexed addressing modes. The MOS 6502 also supports a zero page addressing mode, to speed up access to frequently used data, while the Z80 proposes block transfer and block search instructions[66]. However, significant performance gains can still be achieved by unrolling loops. As an illustration, we consider the MOS 6502 source code presented in figure 19 that copies the 256 bytes of page 2 (`$0200-$02FF` range) to page 3 (`$0300-$03FF` range). The initialization of Y costs 2 cycles and 2 bytes. The three instructions that copy one byte and then increment Y cost 11 cycles and 7 bytes in memory (4 cycles and 3 bytes for `LDA $0200,Y`, 5 cycles and 3 bytes for `STA $0300,Y`, 2 cycles and 1 byte for `INY`). The conditional branch instruction `BNE LOOP` costs 3 cycles if the branch is taken, 2 cycles otherwise. As the branch is taken 255 times, and not taken at the last iteration of the loop, there is a cost of 767 cycles just for the conditional branch instruction. The whole sequence that copies the 256 bytes takes a total of 3585 cycles for a 11 bytes memory footprint. The relative cost associated to the loop (i.e., the conditional branch instruction) is 0.213. Figure 20 presents the source code with an unrolling factor of 2, where 2 bytes are copied within one iteration of the loop. Table 27 shows how the loop overhead can be dramatically reduced by unrolling the code. An unrolling factor of 16 leads to a loop overhead of 47 cycles, reducing the relative cost to 0.016. The memory footprint however increases to 116 bytes.

```
        LDY #$00
LOOP:   LDA $0200,Y
        STA $0300,Y
        INY
        BNE LOOP
```

Figure 19: Copy 256 bytes from page 2 to page 3.

Unrolling factor	**Loop cost (cycles)**	**Total cost (cycles)**	**Loop relative cost**	**Memory (bytes)**
1	767	3585	0.213	11
2	383	3201	0.119	18
4	191	3009	0.063	32
8	95	2913	0.032	60
16	47	2865	0.016	116

Table 27: Loop unrolling time-space trade-off.

[65] Burst filling requires DRAM chips with burst capability.
[66] Such as `LDIR`, see figure 7.

```
        LDY #$00
LOOP:   LDA $0200,Y
        STA $0300,Y
        INY
        LDA $0200,Y
        STA $0300,Y
        INY
        BNE LOOP
```

Figure 20: Copy 256 bytes from page 2 to page 3 (Unrolling factor is 2).

A good candidate for loop unrolling is the processing of an array in memory, and if the array is small enough, it may be even possible to fully unroll the code without impacting significantly the memory footprint. For instance, the BBC BASIC supports 40-bit floating-point numbers (see section 5.2), and operations on floating-point numbers are performed via two *floating-point accumulators* stored in the zero page in memory ranges `$2E-$35` and `$3B-$42` [Rus85, ch. 4] [Bir84, pp. 251–253]. The floating-point accumulator is a 8-byte structure optimized to perform arithmetic computation on 40-bit floating-point numbers, and it contains a sign byte, an overflow/underflow byte, an exponent byte, a 4-byte significand and a significand *rounding byte* [Bir84, pp. 56-57]. The routine presented in figure 21 can be found at `$A178` memory location in the BBC BASIC II ROM and is an example of fully unrolled code [Bir84, p. 390]. The routine adds the 4-byte significand and the significand rounding byte of the second floating-point accumulator (`$3E-$42` range) to the 4-byte significand and the significand rounding byte of the first floating-point accumulator (`$31-$35`). Many examples of loop unrolling related to the processing of the significand can be found in the BBC BASIC ROM (`$A21E`, `$A24D` and `$A2BE` routines, to name a few [Bir84, pp. 393–396]).

In some cases, good opportunities for loop unrolling are missed due to memory constraints. A good example is the shift-and-add multiplication routine `MULT1` found in the Elite source code for Acorn computers that multiplies two 8-bit integers. While the standard versions of Elite (BBC Micro model B, Acorn Electron) do not unroll the `MUL4` loop as there is not enough memory available, the advanced versions (BBC Master, BBC Micro model B with a 6502 Second Processor) unroll completely the loop by replicating seven times the sequence of four instructions presented in figure 22 [Mox20d]. The unrolled loop takes an extra 39 bytes of memory but allows a significant performance gain. In the worst case scenario, i.e., the branch `BCC P%+4` is never taken, the unrolled loop takes 84 cycles, whereas the original loop takes 118 cycles.

In the case of a large number of loop iterations, the loop should not be unrolled completely to keep a good time-space trade-off. An example of partial loop unrolling can be found in the BBC OS 1.20 ROM with the routine that *clears* the VDU (Visual Display Unit) RAM in the `$3000-$7FFF` memory range [BDH83, pp. 462-477] [Har10]. As show in figure 23, the routine starts at `$CBE5`, where the X register is set to zero, and several OS and VDU variables are initialized (*Paged mode counter* at `$0269`, *VDU text cursor X position* at `$0318` and *VDU text cursor Y position* at `$0319`). The code then branches to `$CC02` via an indirect jump that uses the 16-bit address stored in the *VDU jump vector* variable at `$035D`. The unrolled loop starts at `$CC02` where 80 `STA` instructions in *absolute, X* addressing mode are stored sequentially. The index

```
$A178   LDA $35
$A17A   ADC $42
$A17C   STA $35
$A17E   LDA $34
$A180   ADC $41
$A182   STA $34
$A184   LDA $33
$A186   ADC $40
$A188   STA $33
$A18A   LDA $32
$A18C   ADC $3F
$A18E   STA $32
$A190   LDA $31
$A192   ADC $3E
$A194   STA $31
$A196   RTS
```

Figure 21: Addition of the 4-byte significand and significand rounding byte of the floating-point accumulators (`$A178` routine, BBC BASIC II ROM) [Bir84, p. 390].

```
BCC P%+4
ADC T1
ROR A
ROR P
```

Figure 22: Sequence of instructions unrolled in the advanced versions of Elite [Mox20d].

register is incremented at `$CCF2`, and the code branches to `$CD65` after 256 iterations of the unrolled loop are performed.

In the specific case where the programmer wants to initialize a contiguous block of memory with a constant value, it is possible to use the stack for additional performance gain, both in terms of speed and memory. A *push* is indeed a good choice as the stack pointer is automatically updated at no extra cost. On the MOS 6502, `PHA` stores the 8-bit content of the accumulator A at the memory location pointed by the stack pointer S, and then decrements S (3 cycles, 1 byte) [Zak78, p. 148]. On the Z80, the `PUSH qq` instruction decrements the stack pointer SP, stores the high-order byte of the register pair qq (BC, DE, HL or AF) to the memory address pointed by SP, decrements SP again, and stores the low-order byte of the register pair to the memory location pointed by SP (3 M cycles, 11 T cycles, 1 byte) [Zak80, pp. 379–380]. Several precautions need to be taken though before manipulating the stack. First, as interrupts may occur at any time and as interrupt handling relies on the stack, it is necessary to be careful [Zak78, pp. 228–237] [Zak80, pp. 496–510]. Both the MOS 6502 and Z80 offer the possibility to disable/enable maskable interrupts with specific instructions – `SEI/CLI` on the MOS 6502, and `DI/EI` on the Z80 [Zak80, pp. 131, 177] [Zak80, pp. 244, 247].

```
$CBE5   LDX #$00
$CBE7   STX $0269
$CBEA   STX $0318
$CBED   STX $0319
$CBF0   JMP ($035D)
...
$CC02   STA $3000,X
$CC05   STA $3100,X
...
$CCEC   STA $7E00,X
$CCEF   STA $7F00,X
$CCF2   INX
$CCF3   BEQ $CD65
$CCF5   JMP ($035D)
...
```

Figure 23: Clear VDU RAM routine (`$CBE5` memory location, BBC OS 1.20 ROM) [Har10].

Secondly, as the stack pointer will be used, it is necessary to save it beforehand, then restore it afterwards. Finally, as the stack pointer will move down, it needs to be initialized correctly. On the MOS 6502, the stack pointer S has to point to the last memory location of the block, as S is decremented after writing to memory. On the Z80, the stack pointer SP has to point one byte beyond the end of the block, as SP is decremented before writing to memory. An illustration of this technique is the Ant Attack Trainer written by Paul Grenfell that patches the classic ZX Spectrum game Ant Attack[67] to improve the performance of the game, as shown in figure 24 [Gre17]. The routine patched is the code that clears the rendering back buffer [Gre17, 17:17]. While the original routine relies on loop unrolling with an average of 11 T cycles per byte written within the loop (7 T cycles for `LD (HL),C`, 4 T cycles for `INC L`), the use of the stack allows 5.5 T cycles per byte written (11 T cycles for `PUSH HL`). Furthermore, the smaller memory footprint of `PUSH HL` instruction (1 byte) allows to store 64 instructions, instead of 32 sequences of instructions (1 byte for `LD (HL),C`, 1 byte for `INC L`) in the original code – the unrolling factor is multiplied by two. Note that the stack pointer is saved and restored by using self-modifying code – the second operand of `LD SP,1234` instruction is modified to store the stack pointer. This technique will be presented in section 7.4.

On the Z80, combining loop unrolling and the stack is the fastest way to copy a block of memory. `POP qq` can read 2 bytes in 10 T cycles, while `PUSH qq` can write 2 bytes in 11 T cycles, which means that theoretically a byte can be copied in 10.5 T cycles with AF, BC, DE and HL registers, if we put aside some overhead that will be explained later [Zak80, pp. 373–374, 379–380]. This is faster than dedicated block transfer instructions `LDD` and `LDI` that can move one byte from memory location pointed by HL to memory location pointed by DE, and update BC, DE and HL, in 16 T cycles [Zak80, pp. 348–349, 352–353]. The stack is used in two steps. First, a sequence of bytes is *popped* from the source memory block to the Z80 registers.

[67] Ant Attack on ZX Spectrum was written by Sandy White.

```
$8300  LD   ($834F),SP
$8304  LD   SP,$B000
$8307  LD   B,$1D
$8309  LD   HL,0
$830C  PUSH HL
$830D  PUSH HL
...
$834A  PUSH HL
$834B  PUSH HL
$834C  DJNZ $830C
$834E  LD   SP,1234
$8351  RET
```

Figure 24: Optimization of clear buffer routine (Ant Attack Trainer) [Gre17].

AF, *BC*, *DE* and *HL* registers may be used, as well as index registers *IX* and *IY*. Stack operations on index registers are however slower, as `POP IX` and `POP IY` cost 14 T cycles, and `PUSH IX` and `PUSH IY` 15 T cycles [Zak80, pp. 375–378, 381–384]. Alternate register *BC'*, *DE'* and *HL'* may also be used, thanks to the `EXX` instruction that can swap the registers in 4 T cycles[Zak80, p. 256]. Secondly, the sequence of bytes is *pushed* from the registers to the destination memory block. As explained in the previous paragraph, the stack pointer *SP* has to point one byte beyond the end of the destination block. The registers will be pushed in the opposite order. The source code presented in figure 25 illustrates how to use the stack to copy a large block of memory [Gre18, 24:18] [Gre]. First, maskable interrupts are disabled with `DI` instruction, and the stack pointer is saved with the self-modifying code technique mentioned previously [Zak80, p. 244]. Then follows a sequence of instructions that copies 16 bytes from `SRC` memory location to `DEST` memory location. There is a total of 192 sequences, i.e., a total of 3072 bytes are copied. The stack pointer is restored, and maskable interrupts are re-enabled with `EI` [Zak80, p. 247]. With this method, a byte can be copied in 12.75 T cycles. We however assume that some of the parameters are fixed – source and destination memory locations, number of bytes to copy is a multiple of 16, code is fully unrolled. There are variants of this technique, and maximum performance can be achieved by copying a sequence of 14 bytes with *AF*, *BC*, *DE*, *HL*, *BC'*, *DE'* and *HL'* registers, leading to 12.5 T cycles per byte written.

While loop unrolling that results in a short sequence of instructions may be typed in directly by the programmer, this is not the case of a long sequence of instructions that should be generated, to facilitate source code implementation and maintenance, and also to reduce the risk of type in error. For this purpose, most assemblers provide built-in features to generate source code easily. For instance, Pasmo Z80 cross assembler supports macros, where a macro can be defined with the `MACRO` directive, and the macro can be expanded and repeated with the `REPT n,v` directive, where `n` is the number of repetitions, and `v` is an optional loop variable that can be used by the macro during its expansion [Alb] [Gre]. Another example is Kick Assembler, an assembler for the MOS 6502 that supports a script language allowing complex code generation [Nie], as illustrated in figure 26.

The generation of large blocks of source code in the context of static compilation, i.e., before the execution of the program, has however severe drawbacks. First, it

```
        DI
        LD   (REST+1),SP
        LD   SP,SRC
        POP  AF
        POP  BC
        POP  DE
        POP  HL
        EXX
        POP  BC
        POP  HL
        POP  DE
        POP  IX
        LD   SP,DEST+16
        PUSH IX
        PUSH DE
        PUSH HL
        PUSH BC
        EXX
        PUSH HL
        PUSH DE
        PUSH BC
        PUSH AF
        ...
REST:   LD   SP,1234
        EI
```

Figure 25: Copy a large block of memory with the stack [Gre].

dramatically increases the size of the program stored on the media (tape, floppy disk). Secondly, it also impacts the loading time of the program, which can be critical on tape and on particularly slow floppy disk drives – such as the Commodore 1541 floppy disk drive [Wsz83]. These drawbacks may be partially addressed by using lossless data compression algorithms to reduce the size of the program – it is indeed likely that unrolled loops will compress well due to their repetitive structure. In this case, it is essential to choose a lossless data compression algorithm that favors an efficient decompression in a environment with limited CPU speed and memory. While the programmer can decide to implement the compression algorithm, it is common practice to use existing tools known as *cruncher*s or *packers*. A popular cruncher for the MOS 6502 is Exomizer [Lin]. Exomizer is more specifically a *cross-cruncher*, as it runs on a modern system to compress a MOS 6502-based program and produce a self-decompressing program for the target computer. Exomizer favors high compression ratio and efficient decompression. As a consequence, the compression may be very slow – even on a modern system. There are however situations where unrolled loops may not compress well. For instance, operands that are a 8-bit constant or a 16-bit constant may vary while the loop is unrolled, and as a consequence render compression inefficient. An alternative to compression of unrolled loops is to generate code at runtime with a *code generator*. As explained in section 7.4, the MOS 6502 and Z80 have a von Neumann architecture that allows data to be considered as code, and vice

```
.var blitterBuffer=$3000
.var charset=$3800
.for (x=0;x<16;x++) {
 .for(var y=0;y<128;y++) {
  if (var y=0) LDA blitterBuffer+x*128+y
  else         EOR blitterBuffer+x*128+y
  STA charset+x*128+y
 }
}
```

Figure 26: *Blitter fill* routine with Kick Assembler [Nie, p. 24].

versa. Code generation at runtime is a technique widely used in the C64 demoscene, and more specifically in the context of *speedcode*, which is the term coined by the C64 demoscene community to designate *loop unrolling* [Cru15] [Dre11]. Demos are indeed very good candidates for code generation, due to their deterministic nature. For this alternative to be worthwhile, it is however necessary that the size of the code generator is smaller than the size of the generated code, and that the cost of code generation at runtime is negligible. Using a code generator offers additional possibilities. It is for instance possible to generate variants of the generated code by specializing the code generator at runtime.

An illustration of runtime code generation is the 4 KB *intro*[68] on C64 programmed by Golara* [Gol19a]. The fact that the program has to fit in just 4 KB makes it a good candidate for code generation to minimize its size. The intro generates speedcode routines at runtime to perform *raster bars* and *FPP*-based (*Flexible Pixel Positioning*) visual effects. The main speedcode routine is named `fpp_speedcode` and resides in the `$A000-$B280` memory range [Gol19a, fppspeedcode.asm]. It is generated with two code templates, respectively placed between `fppcode1s`/`fppcode1e` and `fppcode2s`/`fppcode2e` labels. Figure 27 presents the first code template. This code template writes data to the VIC-II chip registers (`$D011`, `$D016`, `$D017`) and to six of the eight sprite pointers (`$5BFA-$5BFF` memory range[69]) [Com82b, pp. 131–145, 391–393]. Similarly, the second code template updates the VIC-II chip registers (`$D016`, `$D017`, `$D021`) and the sprite pointers. Each code template is copied alternatively, with a total number of 128 copies, to achieve the desired visual effects by updating the VIC-II registers and sprites pointers at each scanline. Furthermore, each copy of the code template in the main speedcode routine is specialized at runtime, and code specialization is performed at each frame by two auxiliary speedcode routines (`copyshadowfpp`, `scrollraster_speedcode`). A sample[70] of the main speedcode routine at runtime is presented in figure 28, illustrating the specialization of the code (e.g., `LDA #$00` instruction of the first code template is transformed into `LDA #$1E` at byte `$A006`).

Loop unrolling techniques at runtime are also used on Z80-based computers, such as the Amstrad CPC. Pict* describes a simple code generator based on a code template and the `LDIR` instruction, presented in figure 29 [Pic93a] [Zak80, pp. 354–355]. The code template placed between `ORIG` and `DEST` labels is *propagated* 46 times by the `LDIR`

[68] An intro is a size-restricted demo. Typical intros are 4 KB or 64 KB [Reu10].
[69] `fppscreen` is equal to `$5800`.
[70] This sample was captured with VICE Monitor [VIC].

```
fppcode1s:
    STA $D011
    STX $D017
    LDA #$00
    STA fppscreen+$3F8+2
    STA fppscreen+$3F8+3
    STA fppscreen+$3F8+4
    STA fppscreen+$3F8+5
    STA fppscreen+$3F8+6
    STY $D017
    STY $D016
    STA fppscreen+$3F8+7
    STX $D016
    LDA #$00
fppcode1e:
```

Figure 27: `fppcode1s`/`fppcode1e` code template [Gol19a].

instruction[71]. As the stack is used by the generated code to fetch data from the `SINTAB` table, maskable interrupts are disabled and the stack pointer is saved beforehand. The stack pointer is restored and maskable interrupts are re-enabled afterwards. Pict* also describes a more complex code generator that speeds up the support of sprites, at the cost of memory [Pic93b]. As the Amstrad CPC hardware does not support sprites, it is left to the programmer to implement sprites, and there are many available techniques with different trade-offs. The presented technique is specific to *Mode 1* screen mode (320x200 pixels, 4 colors), where a byte stores 4 pixels (2 bits per pixel), and as such it is necessary to perform costly computation (bit shifting, masking) to produce the final byte stored in screen memory from the sprite data [Ams84b, ch. 5.2]. The code generator (`GENER` label) produces four routines (`SPROUT` label), as there are four different situations of 2-bit shift for a given sprite pixel, and the sprite data are directly embedded into the code to maximize throughput, as shown in figure 30.

7.4 Self-modifying code

Self-modifying code, i.e., code that can modify its own instructions at runtime, is a well known *trick* that dates back to the early days of computing history. The use of self-modifying code was justified by the extreme technical constraints of early computers. It was deprecated over time as a programming practice as it led to cryptic source code, difficult maintenance and debugging. It is however still used nowadays, in particular to achieve code obfuscation and make evaluation of programs difficult. Self-modifying code is a particular case of runtime code generation, and sometimes both concepts may be confused. Indeed, Cai et al. define self-modifying code as 'any program that loads, generates, or mutates code at runtime' [CSV07, p. 1]. We will first discuss the technical requirements that allow to modify code at runtime. Then, we will present

[71] An alternative to unrolling the loop at runtime would be to unroll the loop before compilation. However, Pict estimates that a better file compression can be achieved with a memory area filled with the same value (zero) [Pic93a].

```
$A000   STA $D011
$A003   STX $D017
$A006   LDA #$1E
$A008   STA $5BFA
$A00B   STA $5BFB
$A00E   STA $5BFC
$A011   STA $5BFD
$A014   STA $5BFE
$A017   STY $D017
$A01A   STY $D016
$A01D   STA $5BFF
$A020   STX $D016
$A023   LDA #$00
$A025   STA $D021
$A028   STX $D017
$A02B   LDA #$1F
$A02D   STA $5BFA
$A030   STA $5BFB
$A033   STA $5BFC
$A036   STA $5BFD
$A039   STA $5BFE
$A03C   STY $D017
$A03F   STY $D016
$A042   STA $5BFF
$A045   STX $D016
$A048   LDA #$1C
...
$B280   RTS
```

Figure 28: First two iterations of the main speedcode routine (`$A000-$B280` memory range). A copy of the first template starts at `$A000`, and a copy of the second template at `$A025`. Bytes `$A007`, `$A024` and `$A02C` are specialized at each frame by auxiliary speedcode routines [Gol19a].

the historical development of this technique, as well as different use cases. Finally, we will study examples on the MOS 6502 and the Z80.

Not all computers are able to modify code at runtime. Indeed, only computers that follow a von Neumann architecture offer this possibility natively. The von Neumann architecture, also known as the Princeton architecture, is a computer architecture that has been described by von Neumann and others in the *First Draft of a Report on the EDVAC* published in 1945 [Neu93]. In this architecture, the computer consists of an arithmetic unit (*Central Arithmetical part* or *CA*), a control unit (*Central Control part* or *CC*), a memory *M* to store both instructions and data, external storage (*Outside recording medium of the device* or *R*), and input and output mechanisms (respectively *I* and *O*) [Neu93, ch. 2.0]. There is no distinction between instructions and data in memory, and a common bus is used to exchange instructions and data between the memory and the arithmetic and control units. The fact that a shared memory and

```
        LD      HL,ORIG
        LD      DE,DEST
        LD      BC,(DEST-ORIG)*46
        LDIR
        DI
        LD      (SAVSP+1),SP
        LD      SP,SINTAB
        LD      B,6
SCR:    POP     HL
ORIG:   POP     DE
        LD      A,(DE)
        LD      (HL),A
        POP     HL
        LD      A,(HL)
        LD      (DE),A
DEST:   DEFS    (DEST-ORIG)*46,0
        POP     DE
        LD      A,(DE)
        LD      (HL),A
        DEC     B
        JP      NZ,SCR
SAVSP:  LD      SP,0
        EI
```

Figure 29: Code generator based on a code template and the `LDIR` instruction [Pic93a].

shared bus are used indistinctly for instructions and data simplifies the architecture but leads to performance concerns as instructions and data cannot be processed in parallel. This is known as the *von Neumann bottleneck* [Bac78]. Examples of microprocessors with a von Neumann architecture include the MOS 6502, the Z80 and the Motorola 68000. Computer architectures such as Harvard architecture do not allow self-modifying code. In the Harvard architecture, there are separate instructions and data memories, and separate instructions and data buses, addressing the von Neumann bottleneck as instructions and data may be processed in parallel. The Mark-I designed by Howard Aiken at Harvard and built by IBM engineers is the first Harvard architecture computer [HP17, ch. M.2]. Variants of the Harvard architecture exist, and there are usually known as *modified Harvard architecture*. A widespread type of *modified Harvard architecture* are processors with a shared main memory to store instructions and data, but with separate instruction and data caches to increase performance. The use of instruction and data caches is a well known technique to increase CPU throughput, as most programs spend a lot of time in short portions of code – such as loops [Mot88, pp. 3.95, 3.113]. These processors may allow code modification at runtime under certain conditions. Examples of the modified Harvard architecture are the Motorola 68020 and Motorola 68030 [Mot84b] [Mot89]. The Motorola 68020 has a 256-byte on-chip instruction cache, and the Motorola 68030 has a 256-byte on-chip instruction cache and an additional 256-byte data cache [Mot84b, ch. 7] [Mot89, ch. 6]. Instruction and data caches can be controlled in *supervisor mode* by the programmer

```
POKE:    LD     A,(DE)
         CP     $FF
         JP     Z,NXTBYTE
         OR     A
         JP     NZ,MASKBYTE
         LD     A,(HL)
         LD     (IY+0),$36
         INC    IY
         LD     (IY+0),A
         INC    IY
NXTBYTE:LD     (IY+0),$23
         INC    IY
         JP     CODED
```

Figure 30: Extract from the code generator. Registers *DE*, *HL* and *IY* point respectively on sprite mask, sprite data and generated code. `$36` and `$23` are opcodes of instructions `LD (HL),n` and `INC HL` [Zak80, pp. 265-266, 301–302] [Pic93b].

with the *Cache Control Register* (*CACR*) which provides cache clearing, cache entry[72] clearing and cache activation/deactivation facilities. Self-modifying code may be achieved in different ways: by deactivating the instruction cache beforehand as shown in figure 31, or by clearing the content of the instruction cache after the code modification and before the code execution. Note that the use of self-modifying code may cause compatibility issues within the same microprocessors family. For instance, self-modifying code techniques used on the Motorola 68000 may not work on Motorola 68020 and Motorola 68030 because of the instruction cache [Mor86, p. 166].

```
MOVEQ  #0,D0
MOVE.C D0,CACR
```

Figure 31: Disable instruction cache (Motorola 68020, 68030).

From an historical point of view, the IBM SSEC (Selective Sequence Electronic Calculator) released in 1948 is the first known computer that supports self-modifying code [Bas+81, p. 365]. The related patent entitled *Selective Sequence Electronic Calculator* gives detailed examples of the use of code modification at runtime [Ham+49, pp. 271–288]. For instance, code modification is used for the iterative approximation of a square root by the Newton formula with the required accuracy. Later on, self-modifying code was also used to handle subroutines. An early example is the CDC 6600 mainframe computer designed by Seymour Cray and Jim Thornton, and released in 1964 by Control Data Corporation. In the CDC 6600, procedure linkage was implemented with "self-modification", although it was considered as potentially unsafe [KEH93, p. 1] [Tho70, p. 114]. In the first volume of its comprehensive monograph entitled *The Art*

[72] Operations on a specific entry of the cache require the additional use of the *Cache Address Register* (*CAAR*) to specify the index of the entry.

of Computer Programming, whose first edition was published in 1968, Donald Knuth describes an hypothetical computer called *MIX* that leverages self-modifying code to support subroutines [Knu68, ch. 1.3.1, 1.4.1]. *Subroutine linkage* is achieved by using the jump address register J and the `STJ` instruction to modify the `JMP` instruction that returns to the main program. Knuth however considered the use of self-modifying code to support subroutines obsolete in later editions [Knu97, p. 187]. In 1984, Fred Cohen defined in its influential paper entitled *Computer Viruses – Theory and Experiments* a computer virus as "a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself" [Coh87]. In 1990, the first virus presenting mutation capabilities to bypass virus scanners was discovered [Szo05, ch. 7.5.1]. Polymorphic and metamorphic viruses, i.e., viruses that can mutate, rely on a mutation engine that can perform various code transformations while preserving the exact function of the code. Code transformations include instruction equivalence, instruction sequence equivalence, instruction reordering, register renaming, data reordering, junk code insertion, run-time code generation, inlining and outlining [Ayc06, ch. 3]. These code transformations techniques rely in part on the capability of modifying code at runtime. In the field of software protection, self-modifying code is one of the key technique used to limit illegal copying of software. It is particularly useful to obfuscate protection code, by making reverse-engineering and disassembly more difficult [Mor83, ch. 6]. Self-modifying code is also a good technique to build tamper resistant software, i.e., a system immune from observation and modification [Auc96]. The use of self-modifying code for code obfuscation is still popular, whether for a legitimate purpose (software and intellectual property protection), or for a criminal goal (such as the development of polymorphic and metamorphic viruses). As a consequence, the study of *von Neumann programs* – programs with self-modifying capabilities – is still an active domain of research. Recent works discuss for instance the cost of attacking a von Neumann program by defining a taxonomy of self-modifying code techniques and categorising the adversary's capabilities [MKP11], and the formal evaluation of a von Neumann program – either by constructing a formal Hoare-style logic framework with strong support for self-modifying code [CSV07], or by extending a PushDown System (PSD) with self-modifying rules to perform reachability analysis [TY17].

We will now present different examples of self-modifying code technique on the MOS 6502 and Z80. The use of self-modifying code in the context of software protection has been documented on MOS 6502-based systems. George Morrison describes in *Atari Software Protection Techniques* published in 1983 several use cases of self-modifying code. First, self-modifying code may be used to conceal protection code. In the example provided, the system call `JSR $E453` used by the protection code to check for bad sectors on the disk is generated dynamically [Mor83, pp. 40–43]. As such, the memory location that stores an "innocent" `STA $0000` instruction is replaced at runtime by the byte sequence `$20 $53 $E4` corresponding to the expected system call. The instructions writing these bytes may be mixed in other routines, stored in different places, and the written bytes may also be generated in convoluted ways – by adding or subtracting numbers – making reverse engineering even more difficult. Secondly, self-modifying code can be used to prevent copy of cartridges on disk or tape [Mor83, pp. 56–58]. Since a program loaded from a disk or a tape is stored in RAM, an efficient copy protection technique for cartridges is to add self-modifying code that will render the program useless by writing bytes at specific memory locations. On the cartridge, the self-modifying code has no effect, as the cartridge is a ROM (Read Only Memory) chip, and the cartridge disables the RAM sharing the same address space. As previously, the self-modifying code may also be concealed, distributed in different

parts of the memory, to make reverse engineering more challenging.

An illustration of self-modifying code to make disassembly and reverse engineering of programs more difficult is Pac-Man on the Apple II published by Datasoft, Inc. in 1984. The program includes in the 256-byte boot block of the floppy disk a copy protection mechanism leveraging self-modifying code and undocumented instructions [Ayc16, ch. 8.1] [4am15b]. Self-modifying code is used to change at runtime a decryption key used to decrypt 152 bytes located in the `$0801-$0898` address range. The code modification is performed at address `$085F`, where the value `$CA` is stored at address `$089D`, transforming the `EOR #$AA` instruction into `EOR #$CA`, as shown in figure 32.

```
$085D   $A9 $CA      LDA #$CA
$085F   $8D $9D $08  STA $089D
$0862   $B0 $31      BCS $0895
...
$0895   $8A          TXA
$0896   $48          PHA
$0897   $A0 $98      LDY #$98
$0899   $B9 $00 $08  LDA $0800,Y
$089C   $49 $AA      EOR #$AA
$089E   $99 $00 $07  STA $0700,Y
$08A1   $88          DEY
$08A2   $D0 $F5      BNE $0899
```

Figure 32: Decryption key update with self-modifying code (Pac-Man on Apple II, Datasoft, Inc.) [Ayc16, ch. 8.1] [4am15b].

Another notable use case of self-modifying code is code specialization. Prince of Persia on the Apple II uses this technique to specialize the graphic engine code at runtime, as shown in figure 33. In this example, the `OPACITY` zero page variable [Mec89, EQ.S, 372] is used as an index to get from the `OPCODE` table the opcode to use for code specialization [Mec89, HRTABLES.S, 321–326]. This code specialization at runtime is an example of both space and time optimization as it reduces the size of the source code and also eliminates the cost of comparison in loops [Ayc16, pp. 183–185]. Refer also to figure 28 for another example of code specialization on the MOS 6502, in the context of a speedcode routine.

```
LDX OPACITY
LDA OPCODE,X
STA :80
STA :81
```

Figure 33: Code specialization with self-modifying code (Prince of Persia on Apple II, Broderbund Software, Inc.) [Mec89, HIRES.S, 711–714].

Regarding the Z80, examples of self-modifying code can be found in section 7.3. In figures 24, 25 and 29, the technique is used to save and restore the stack pointer, by modifying the second operand of a `LD SP,nn` instruction [Zak80, pp. 293–294]. The technique is also used to generate code at runtime, as shown in figures 29 and 30.

The use of self-modifying code as a programming practice had a very limited impact on the development tools. An example worth mentioning is Kick Assembler for the MOS 6502 that provides features to ease the implementation and maintenance of self-modifying code. For instance, it is possible to put labels in front of operands [Nie, ch. 3.4]. Kick Assembler also provide constants for the different opcodes [Nie, ch. 14.3]. The constants follow a naming convention, where the prefix is the mnemonic and the suffix – if relevant – is the addressing mode. For example, `LDA_IMM` corresponds to the `LDA` instruction in *Immediate* addressing mode (opcode `$A9`), while `LDA_ZP` is the `LDA` instruction in *Zero page* addressing mode (opcode `$A5`).

7.5 Undocumented instructions

Undocumented instructions refer to instructions that are not officially documented by the microprocessor's designer or manufacturer. *Undocumented instructions* are also called *illegal instructions*, *undefined instructions*, *unintended instructions*, *extra instructions*, or alternatively *undocumented opcodes*, *illegal opcodes*, *undefined opcodes*, *unintended opcodes*, *extra opcodes* [Sim+85, p. 72] [She83] [Nit85, ch. 8.2] [Gro20]. A notable early example is the Motorola 6800 for which undocumented instructions were reported [Whe77]. The MOS 6502 and the Z80 are also well known examples of microprocessors with undocumented instructions. For instance, the MOS 6502 microprocessor instruction set consists of 56 different types of instructions, where each type of instruction supports at least one of the 13 addressing modes available, for a total of 151 documented instructions [MOS76b] [She83]. Since an instruction is encoded in an 8-bit opcode, there are 105 opcodes that are not documented. Two types of undocumented instructions are commonly found. The first type consists of undocumented instructions that do nothing during a certain amount of clock cycles. These instructions are usually named `NOP`, similarly to the official `NOP` (*No Operation*) instruction [Gro20] [Zak78, p. 145]. The second type is undocumented instructions that halt the CPU and therefore stop the execution of the program. These instructions may be called `HCF` (*Halt and Catch Fire*) on the Motorola 6800, `JAM`, `KIL` or `HLT` on the MOS 6502 [Whe77] [Gro20, p. 41]. Undocumented instructions on 8-bit microprocessors have been largely studied, as there were many empty slots in the opcode table. As we will show later, undocumented instructions are used in the context of software protection, as it makes reverse engineering much harder. There are also helpful for optimization, whether it be space optimization or time optimization. Their use is however risky as there is no guarantee that their behavior will remain the same in future versions of the microprocessor [Whe77] [Nit85, p. 21] [Sim+85, p. 77] [Com82a, p. 9]. An example is the WDC 65C02 microprocessor, an evolution of the MOS 6502 introduced by Western Design Center and that supports additional instructions. It is likely that many programs using MOS 6502 undocumented instructions will not work correctly on the WDC 65C02 [Wag83]. To prevent unexpected consequences of undocumented instructions, microprocessors have evolved to behave in a well defined way whenever an undocumented instruction is encountered. An early example is the Motorola 68000 where undocumented instructions are gathered in two groups of 16-bit opcodes, the first group beginning with the `%1010` bit pattern, and the second group with the `%1111` bit pattern [Sta83]. Any attempt to execute an undocumented instruction leads to the execution of a trap routine[73] for corrective action. Nowadays, all modern micro-

[73] On the Motorola 68000 family, the illegal instruction trap can also be called directly with the `ILLEGAL` instruction [Mot92, p. 4.107].

processors generate theoretically an *Invalid opcode* or *Invalid instruction* exception. There are however several recent examples of undocumented instructions that are not handled properly – this is a critical security concern as undocumented instructions can lead to a Denial-of-Service (DoS) [Dom17]. As a consequence, the detection of undocumented instructions is still an active domain of research. The study of undocumented instructions is also important for the development of emulators – and more specifically for the emulation of 8-bit systems, such as the ones based on the MOS 6502 and derivatives, or on the Z80. As we will show later, many software on these systems use undocumented instructions, and an accurate emulation requires a good understanding of their behaviour [Gro20, p. II] [You05, p. 5].

7.5.1 Study of undocumented instructions

Several methods have been proposed to study undocumented instructions. We will take as an example the `$87` opcode on the MOS 6502. This instruction is now well documented [Gro20, pp. 15–17]: it performs a logical *AND* of *A* and *X* registers, and stores the result in the zero page location defined in the one byte operand. The instruction is usually named `SAX`, but other names exist – `ANDX`, `AXS` and `AAX` [She83] [Sim+85] [Gro20, pp. 15–17].

```
PHP
PHA
STX $FE
AND $FE
STA $FE
PLA
PLP
```

Figure 34: Sequence of documented instructions equivalent to the `$87 $FE` undocumented instruction on the MOS 6502 [Gro20, p. 15].

The first obvious method is to execute the opcode and see what happens. Simstad et al. propose to study the behavior of the `$87` opcode on the MOS 6510 with the help of a monitor program [Sim+85, pp. 74–76]. For this purpose, the sequence of bytes `$87 $FB $00 $00` is written at `$1000` memory location and then executed with the monitor program. The proposed test sequence assumes[74] that the `$87` opcode is followed by a one byte or a two bytes operand, and that the low-order byte of the operand is `$FB`. The `$00` opcode corresponds to the `BRK` instruction that will give the control back to the monitor program [Zak78, p. 111]. After execution of the test sequence, the control is given back to the monitor program, and the program counter contains `$1002`, which means that the `$87` opcode is followed by a one byte operand. From this point, different hypotheses regarding the nature of the `$FB` operand may be taken and verified. The operand cannot be an 8-bit displacement (*Relative* addressing mode), otherwise the program counter would have a different value[75]. The operand may be

[74] The sequence of bytes `$87 $00` can be used to test the hypothesis of an opcode that is not followed by any operand.

[75] It is recommended to fill the memory around the test sequence 127 bytes forward and 128 bytes backward with `BRK` instructions to ensure that the monitor program takes the control back in case a branch happens.

an 8-bit data (*Immediate* addressing mode) or an 8-bit zero page address (*(Indirect, X)*, *(Indirect), Y*, *Zero page*, *Zero page, X* and *Zero page, Y* addressing modes[76]). Hypotheses regarding indexed addressing modes may be verified by initializing X and Y with different values and looking at the content of the memory after the execution of the test sequence, hypotheses regarding the use of the accumulator may be also verified by setting a specific value in the accumulator beforehand, and so forth. This method allows to document the behavior of the opcode, however it has two drawbacks. First, a minor drawback is that it usually takes time to assess the different hypotheses. Secondly, a major drawback is that the conclusion is only valid for the very specific conditions of the test, i.e., the content of the memory and registers before the execution of the test sequence.

To address the first drawback, a proposed method is to look at the format of the opcode, and make hypotheses regarding its behavior by looking at opcodes with a similar format. We can indeed suppose that the encoding of the opcodes follows a certain logic. This is the case for the MOS 6502 instruction set, where for example `STA` follows the `%100bbb01` binary layout, and `STX` follows the `%100bb110` binary layout, where `b` sequences take certain values, depending on the addressing mode [Zak78, pp. 162–164]. Regarding the `$87` opcode, Joel C. Shepherd takes the hypothesis that it is an instruction that stores a combination of A and X in zero page addressing mode, since the binary representation is similar to `STA` and `STX` instructions in zero page addressing mode, as shown in table 28 [She83]. The hypothesis needs however to be confirmed by running a test sequence [Sim+85, p. 77].

Instruction	Addressing mode	Opcode	Binary
`STA`	*Zero page*	`$85`	`%10000101`
`STX`	*Zero page*	`$86`	`%10000110`
`???`	*Zero page*	`$87`	`%10000111`

Table 28: Hypothesis regarding `$87` opcode [She83].

Regarding the second drawback, it is necessary to determine for each instruction its scope of validity. In this case, automated tests may be used to determine the behavior of the instruction with a given input parameter space. An example of such test suite[77] is the one developed by Wolfgang Lorenz for the MOS 6502 and derivatives [Gro20, p. I]. Following these automated tests, undocumented instructions may be classified into different groups. For instance, Groepaz* classifies the MOS 6510[78] instructions into three distinct groups [Gro20]. The first group are *stable* instructions [Gro20, pp. 7–41]. An instruction is stable if it behaves in a consistent manner and may be used by the programmer without any particular precaution. The `$87` opcode is an example of stable instruction. There are 94 stable instructions on the MOS 6510, including 24 `NOP` and 12 `JAM`. The second group are *partially unstable* instructions [Gro20, pp. 42–53]. In this case, certain precautions must be taken, but the instruction always behaves in a predictable manner. There are eight partially unstable instructions on the MOS 6510,

[76] Refer to table 4 for the list of addressing modes on the MOS 6502.

[77] These test suites also play a critical role in the context of software emulation. For instance, the VICE emulator which emulates Commodore computers based on the MOS 6502 and MOS 6510 implements the Wolfgang Lorenz test suite to ensure an accurate emulation of these microprocessors [Gro20, p. 99].

[78] If most findings should apply for the MOS 6500 family, there may be however slight differences, especially regarding unstable instructions [Gro20, p. II].

including three NOP instructions. An example of precaution that must be taken for instructions in an indexed addressing mode ($93, $9B, $9C, $9E and $9F) is to ensure that the index register is in a range that will prevent a page boundary to be crossed. The third group are *highly unstable* instructions. In this case, the instruction may behave in a unpredictable manner due to analogue side effects such as time or chip temperature, and it may only be used with severe restrictions. There are two highly unstable instructions on the MOS 6510 ($8B and $AB) [79]. Refer to table 29 for the list of partially unstable and highly unstable instructions on the MOS 6510.

Instruction	Addressing mode	Opcode
SHA	*(Indirect), Y*	$93
	Absolute, Y	$9F
SHY	*Absolute, X*	$9C
SHX	*Absolute, Y*	$9E
TAS	*Absolute, Y*	$9B
LAS	*Absolute, Y*	$BB
NOP	*Immediate*	$82
	Immediate	$C2
	Immediate	$E2
LAX	*Immediate*	$AB
ANE	*Immediate*	$8B

Table 29: Partially unstable and highly unstable instructions on the MOS 6510 [Gro20, pp. 3–4].

Another approach to understand the behavior of the undocumented instructions is to study the architecture of the microprocessor at a transistor level [Sim+85, p. 77]. If this method was not possible during the commercial life of microprocessors due to the fact that this information is proprietary and therefore not accessible to the public, recent works have proven this approach to give very valuable insight regarding the execution of instructions – whether these instructions are documented or not – at a transistor level. A good illustration is the Visual 6502 project, a logic simulator of the MOS 6502 [JSS10]. Starting from high resolution photographs of the different physical layers of the MOS 6502, James et al. derived a polygonal representation of each physical layer of the chip. Geometric intersection of polygons allows to form transistors and build the connectivity, resulting in an accurate simulation of the MOS 6502, without any prior knowledge of the specification and the instruction set of the microprocessor. The vectorization process is however prone to a few errors. The limited complexity of the MOS 6502 (3510 transistors) allowed to correct these errors by hand (8 errors for over 20 000 components modeled). If Visual 6502 allows a very accurate simulation of the MOS 6502, there are however some limitations for highly unstable instructions. As the simulation is entirely digital, analogue side effects are not reproduced.

The study of undocumented instructions on 8-bit microprocessors is facilitated by the simplicity of the instruction set, as well as the reduced size of the opcodes

[79] Despite their high instability, these instructions have been found in existing software on the C64. Ocean/Imagine tape loader, Mastertronic variant of the *Burner* tape loader and Turrican 3 use $8B, while Wizball uses $AB [Gro20, pp. 54–60].

and operands. On the MOS 6502, an instruction has a 1 byte opcode and up to a 2 bytes operand, which limits drastically the search space. However, on modern microprocessors, the search space is too large to enumerate all the possible instructions. In *Breaking the x86 ISA*, Christopher Domas describes how to perform an exhaustive test of documented and undocumented instructions on a x86 microprocessor [Dom17]. This requires to leverage the x86 microprocessor single-step mode, by setting the x86 trap flag before the execution of an instruction, and then catching the single-step interrupt. As an instruction may be up to 15 bytes long, a method named *tunneling* is used to reduce the search space, and requires to determine the actual length of a 15-byte long candidate instruction. To achieve this, a *page fault analysis* technique is used. Two consecutive pages in memory are configured, the first with *read*, *write* and *execute* permissions, the second with *read* and *write* permissions only. The candidate 15-byte instruction is then placed in memory, with the first byte at the end of the first page, and the remaining bytes at the start of the second. The instruction is then executed. If the instruction decoder requires a byte that is stored in the second page, then a *page fault* error occurs. The candidate instruction is incrementally moved across page boundaries till no page fault occurs. Eventually, the entire instruction resides in the executable page, and the actual length of the instruction can be deduced. This method reduces the search space from $\sim 1.3 \times 10^{36}$ to $\sim 10^9$, allowing to test the x86 instruction set in a reasonable amount of time. Scans performed on dozens of x86 microprocessors allowed to discover many undocumented instructions, bugs in assemblers and disassemblers, flaws in enterprise hypervisors, as well as critical x86 hardware bugs.

7.5.2 Undocumented instructions on the Z80

In the previous section, we presented different methods to study undocumented instructions, with a particular attention to the case of the MOS 6502 and derivatives. We focus now on the Z80, where the instruction set is more complex. Indeed, while opcodes on the MOS 6502 are strictly 8-bit, opcodes on the Z80 may be 8-bit or 16-bit [Zil76, ch. 5.0]. There are exactly 252 documented instructions with a 8-bit opcode, and the 8-bit values left (`$CB`, `$DD`, `$ED`, `$FD`) are the most significant byte of the 16-bit opcodes. This is in these 16-bit opcodes that undocumented instructions can be found. Sean Young classifies undocumented instructions on the Z80 in six distinct groups presented in table 30 [You05, ch. 3].

Many of these undocumented instructions are related to documented instructions and follow the same encoding logic. For instance, there are eight undocumented instructions prefixed by `$CB`, with opcodes ranging from `$CB30` to `$CB37`, listed in table 31. Young named these instructions `SLL` (Shift Logical Left), and they have the same behavior as the eight documented instructions `SLA` (Shift Left Arithmetic) with a `$CB` prefix, except that `SLL` sets bit 0 while `SLA` resets it. If the experimentation is essential to document correctly the instructions, a prior analysis of the documentation is also very useful to make relevant assumptions. If we look at these opcodes, they belong in theory to the rotate and shift group, and they are particularly close to the shift instructions `SLA`, `SRA` and `SRL`, as presented in table 32. The assumption that the instruction performs a shift operation is relevant. Likewise, with the help of table 33, relevant assumptions regarding the 3 least significant bits of these undocumented instructions can be made, all of which are confirmed experimentally[80].

[80] A well known test suite for the Z80 is the Zexdoc/Zexall instruction exerciser developed

Prefix	Description
$CB	8 undocumented instructions named SLL
$DD	46 undocumented instructions where *IX* is used instead of *HL* Opcodes prefixed by $DDCB are excluded
$FD	46 undocumented instructions where *IY* is used instead of *HL* Opcodes prefixed by $FDCB are excluded
$ED	20 undocumented instructions Mostly duplicates of documented instructions
$DDCB	225 undocumented instructions in *(IX+d) Indexed* addressing mode Result is stored, if any, in one of the seven main 8-bit registers
$FDCB	225 undocumented instructions in *(IY+d) Indexed* addressing mode Result is stored, if any, in one of the seven main 8-bit registers

Table 30: Groups of undocumented instructions on the Z80 as proposed by Young [You05, ch. 3].

Opcode	Instruction
$CB30	SLL B
$CB31	SLL C
$CB32	SLL D
$CB33	SLL E
$CB34	SLL H
$CB35	SLL L
$CB36	SLL (HL)
$CB37	SLL A

Table 31: Undocumented instructions of the $CB group [You05, p. 11].

Similar assumptions can be made for other undocumented instructions. Regarding instructions prefixed by $DD and $FD, the documentation shows a direct relation in the opcode format between instructions that use *HL* register and instructions that use *IX* or *IY* registers. For instance, LD A,(HL) is encoded $7E, while LD A,(IX+d) and LD A,(IY+d) are encoded $DD7E and $FD7E, followed by the offset value [Zil76, p. 25]. Likewise, INC HL opcode is $23, while INC IX and INC IY are respectively encoded $DD23 and $FD23 [Zil76, p. 31]. A careful study of the documentation shows that this relation is consistent, for all the addressing modes concerned. Experiments confirm that many undocumented instructions follow the same pattern. For instance, the instruction INC H is encoded $24, and the undocumented opcodes $DD24 and $FD24 correspond respectively to INC IXh and INC IYh, where IXh and IYh designate the most significant byte of *IX* and *IY*.

Undocumented instructions prefixed by $DDCB and $FDCB are of particular interest as they allow to perform two operations at once, which is both a space and time optimization. According to the documentation, the instructions prefixed by $DDCB and $FDCB belong to the rotate and shift group, and the bit manipulation group, respectively in the context of *(IX+d)* and *(IY+d) Indexed* addressing modes [Zil76,

by Franck Cringle for the YAZE Z80 Emulator [You05, p25].

Encoding	Instruction
%11001011 %00100...	SLA
%11001011 %00101...	SRA
%11001011 %00110...	SLL
%11001011 %00111...	SRL

Table 32: Encoding of shift instructions for *Register* addressing and *Register indirect* addressing in the rotate and shift group [Zil76, pp. 31–33, 50].

Encoding	Source and destination
%.....000	B
%.....001	C
%.....010	D
%.....011	E
%.....100	H
%.....101	L
%.....110	(HL)
%.....111	A

Table 33: Encoding of source and destination for *Register* addressing and *Register indirect* addressing in the rotate and shift group [Zil76, pp. 31-33, 50].

pp. 31–33]. A careful study of the associated opcodes shows that the 3 least significant bits of the fourth byte are always set to %110. Changing these bits with the encoding associated to one of the main 8-bit registers (see table 33) leads the instruction to store the result, if any[81], in the designated register. For instance, the undocumented instruction RLC (IX+1),B (opcode $DDCB0100) is equivalent to the sequence of three documented instructions presented in figure 35.

```
LD B,(IX+1)
RLC B
LD (IX+1),B
```

Figure 35: Sequence of documented instructions equivalent to the $DDCB0100 undocumented instruction on the Z80 [You05, p. 13].

While many undocumented instructions on the Z80 have been documented by Young, there are however still gaps in the 16-bit opcode list [You05, ch. 9]. It is likely that these instructions are considered useless, and therefore not documented.

[81] The exceptions are the BIT instructions that test a bit and only affect the *Z* flag [You05, p14].

7.5.3 Use of undocumented instructions

Undocumented instructions have been widely used in the context of software protection, to make reverse engineering of the code much harder. In a standard monitor or disassembler, these instructions appear as `???` or similar[82], as they are not recognized [Sim+85, pp. 73–74]. Besides, as the size of the instruction is unknown, the monitor or disassembler usually assumes that it is 1 byte long, and this may shift the disassembly of the following instructions. As no official documentation mentions these instructions, it is left to the reverse engineer to document these and study their behavior. There are many examples of the use of undocumented instructions to harden software protection and prevent software privacy. One example is Mr.Do! on the Apple II, where the undocumented instruction `$74` is used, as shown in figure 36 [4am15a, pp. 8–9]. The built in Apple monitor displays `???` as the instruction is unknown, followed by a `JMP $1CB0` instruction, as it is assumed that `$74` is a 1 byte long anomaly.

```
$0801 $74          ???
$0802 $4C $B0 $1C JMP $1CB0
```

Figure 36: Mr.Do! disassembly on Apple II monitor [4am15a, pp. 8–9].

`$74` corresponds to a `NOP` instruction in *Zero page, X* addressing mode [Gro20, pp. 38–39]. The opcode is followed by a 1 byte operand[83], which is the location in the zero page. The instruction reads a byte from the zero page, but the byte is not stored anywhere. If the monitor had the ability to recognize undocumented opcodes, it would have displayed the code presented in figure 37. Other games on the Apple II use the same technique, such as Pac-Man (Datasoft, Inc.) [4am15b, pp. 9–10] [Ayc16, p. 169].

```
$0801 $74 $4C      NOP $4C,X
$0803 $B0 $1C      BCS $0821
```

Figure 37: Mr.Do! actual code [4am15a, p8-9].

In the context of software protection, undocumented instructions may also be used to crash the system as a way to punish the pirate. This technique has been used in games published by Epyx on the C64 (Death Sword V1 and V2, Rad Warrior, Spiderbot). In these games, if the protection check fails, then a `JAM` instruction (`$02`) is executed to send the computer in an infinite loop [KJP90, pp. 114–120]. When a `JAM` instruction[84] is executed, the byte following the instruction is fetched, the data and address bus are both set to `$FF`, and the program execution stops [Gro20, p. 41]. It is then necessary to reset the microprocessor.

[82] Devpac, a popular disassembler for Z80-based computers (Amstrad CPC, ZX Spectrum), displays `NOP*` when an undocumented instruction is encountered. The undocumented instruction is assumed to be a `NOP`, and the `*` is indicative of an unexpected situation [Ams84a, p. Mon.2.7] [HiS87, p. Mon.11].

[83] As the instruction is 2 bytes long, it is also known as `DOP`, a double `NOP`.

[84] There are 12 different opcodes for the `JAM` instruction: `$02`, `$12`, `$22`, `$32`, `$42`, `$52`, `$62`, `$72`, `$92`, `$B2`, `$D2` and `$F2`.

Undocumented instructions may be also very useful for both space and time optimization. On the MOS 6500 family, many undocumented instructions are a combination of two documented instructions, and as such are able to perform two operations at once. There may be however some restrictions, such as on the addressing modes, or on the way the flags are affected. Table 34 presents the undocumented instructions that are a combination of two documented instructions with the same addressing mode.

Undocumented instruction	**Combination of documented instructions**
SLO	ASL then ORA
RLA	ROL then AND
SRE	LSR then EOR
RRA	ROR then ADC
SAX	Store *A AND X* (STA, STX)
LAX	Load *A* and *X* (LDA, LDX)
DCP	DEC then CMP
ISC	INC then SBC

Table 34: Combinations of documented instructions with the same addressing mode [Gro20, p. 5].

Let's consider the LAX instruction that loads both the accumulator and the *X* register with the contents of a memory location [Gro20, pp. 18–19]. Loading the accumulator and the *X* register with the same value is useful if the original value is manipulated, and then needs to be restored. Instead of loading it from the memory, it can be transferred from the *X* register to the accumulator with a TXA instruction, which saves both space and time. An illustration of this technique can be found in Turrican 3 on the C64 programmed by AEG*. As shown in figure 38, *A* and *X* are first initialized with the byte stored at $DC00 memory location. A bitwise AND is then performed with the accumulator, followed by a conditional branch to $4EC6 memory location. Finally, the original value is restored in the accumulator with a TXA instruction. The use of LAX $DC00 (4 cycles, 3 bytes) and TXA (2 cycles, 1 byte) instead of LDA $DC00 (to load the original value, 4 cycles, 3 bytes) and LDA $DC00 again (to restore the original value, 4 cycles, 3 bytes) allows to save 2 cycles and 2 bytes.

```
$4EB7    LAX $DC00
$4EBA    AND #$04
$4EBC    BEQ $4EC6
$4EBE    TXA
```

Figure 38: Use of LAX in Turrican 3 version 1.1. Analysis performed with VICE Monitor.

Another useful undocumented instruction is the $CB opcode, known as AXS or SBX, depending on the assembler [Gro20, pp. 31–34]. The AXS instruction performs a logical *AND* of the accumulator and the *X* register, subtracts an immediate value, and then stores the results in *X*, leaving the accumulator intact. This instruction is useful for space and time optimization in different use cases, such as subtract a fixed value from

X, decrement nibbles in X independently, or apply a mask to X. `AXS` may also be combined with `LAX` [Gol19a, main.asm] [Gol19b, part2.asm].

The fact that undocumented instructions were gradually being documented had an impact on the development tools, such as assemblers, disassemblers and monitors. An illustration is the evolution of the programming tools for the MOS 6500 family. In early assemblers, it was necessary for the programmer to write the bytes – 8-bit opcode, followed by an 8-bit or 16-bit operand if necessary – of the undocumented instruction directly in the source code. Similarly, the use of a disassembler or monitor required an external documentation to identify the undocumented instructions. As undocumented instructions are now well documented, modern tools usually support them. An example is Kick Assembler, an assembler that supports different instructions sets, including a subset of the undocumented instructions. The programmer can specify the instruction set to use with the `.cpu` directive, followed by an instruction set identifier. The default instruction set `_6502` includes official instructions and a subset of the undocumented instructions, whereas the instruction set `_6502NoIllegals` only includes the official instructions [Nie, ch. 3.1, A.3.2]. The support and naming of undocumented instructions differs from one assembler to another, as shown by Groepaz* on five of the most popular assemblers on the C64 (Kick Assembler, ACME, ca65, dasm and 64tass) [Gro20, p. 89]. Some assemblers may not support certain undocumented instructions as these instructions are not considered useful. For instance, the `$02` opcode which locks the CPU is not supported in Kick Assembler and dasm. It is however supported by others. Some assemblers may also not support certain undocumented instructions as they are redundant with existing instructions. For example, as the `$1A` opcode is strictly identical to the official `NOP` instruction (`$EA`), it is not supported by any of the assemblers. Regarding the naming, as there is no standard naming convention, we observe some differences. For instance, the naming varies for `$CB` opcode, the instruction being named `AXS` (Kick Assembler, ca65, 64tass) or `SBX` (ACME, dasm, 64tass). In another example, ACME assembler supports `DOP` and `TOP` instructions for undocumented `NOP` which are respectively 2 and 3 bytes long, while the others use a generic `NOP` for both cases. For undocumented opcodes that are not supported by an assembler because they are considered useless or redundant, it is necessary to use the classic technique: write directly the bytes in the source code. Kick Assembler offers a `.byte` directive that can be used for this purpose. Another example of modern tool that supports the undocumented instructions is the VICE Monitor [VIC, /src/monitor/asm6502.c]. Note that once again the naming is not consistent. For instance, the undocumented `NOP` instructions are labeled `NOOP`, to probably avoid confusion with the official `NOP` instruction (`$EA`). Table 35 highlights the differences regarding the naming and support of undocumented instructions in popular C64 development tools.

8 Conclusion and perspectives

Despite the thousands of software that have been released on 8-bit systems during their commercial lifetime, only a handful of original source codes have been preserved. Even if period documentation regarding programming on 8-bit microprocessors is abundant and easily accessible – whether it be books or specialized magazines that have been digitized and are publicly available on the Internet – the low availability of original source codes limits our understanding of programming practices on 8-bit microprocessors.

The software in their primary digital form, i.e., an executable binary file with possibly one or more data files, are however largely available. In particular, software

Opcode	Kick Assembler	ACME	ca65	dasm	64tass	VICE Monitor
$87	SAX	SAX	SAX	SAX	SAX	SAX
$0B	ANC	ANC	ANC	ANC	ANC	ANC
$2B	ANC2	-	-	-	-	ANC
$CB	AXS	SBX	AXS	SBX	AXS SBX	SBX
$02	-	JAM	JAM	-	JAM	JAM
$04	NOP	NOP DOP	NOP	NOP	NOP	NOOP
$0C	NOP	NOP TOP	NOP	NOP	NOP	NOOP
$1A	-	-	-	-	-	NOOP

Table 35: Examples of undocumented instructions naming in different C64 tools [Nie, A.3.2. Illegal 6502 Mnemonics] [Gro20, p. 89] [VIC, /src/monitor/asm6502.c].

piracy has had a critical impact on the widespread distribution and availability of software during the commercial lifetime of 8-bit systems. Software piracy also favored the diffusion on alternative storage media, thus preventing the disappearance of software due to the original storage media deterioration. The popularity of 8-bit system emulators on modern computers has largely perpetuated the availability of a large catalogue of 8-bit software to the public[85]. If the executable binary file can be disassembled, it is however a long and tedious work to reconstruct and comment the source code. In particular, the use of techniques in the software such as encryption, compression, self-modification and undocumented instructions can make the task of reverse engineering a daunting challenge. These difficulties are illustrated in the reverse engineering of Mr.Do! and Pac-Man on Apple II presented in sections 7.4 and 7.5 [4am15b][4am15a]. It is however possible to fully reconstruct a source code, but it can take years, as shown by the change logs of the reverse engineering of Manic Miner and Jet Set Willy on the ZX Spectrum [Dym20b] [Dym20a].

An alternative is to perform an analysis of the software at runtime. It can be performed manually with a monitor program, and this technique was helpful to discover certain techniques presented previously. For instance, we used the VICE Monitor to highlight the use of speedcode and undocumented instructions on the C64 (see figures 28 and 38). The advantage of runtime analysis in the context of our study is that it can reveal some of the techniques (e.g., loop unrolling) without requiring a complete reconstruction of the source code. If runtime analysis is ultimately the best technique, it takes however some time when done manually.

The different methods used in our study show some limits, and in particular do not allow a large scale study. If we were able to identify and highlight some techniques, we however did not quantify and study over time the use of these techniques. To achieve this, it would be interesting to instrument and automate runtime analysis

[85]Software is usually available as tape or disk images. For instance, `T64` and `D64` are popular tape and disk image formats on C64 emulators such as VICE [VIC].

with modern tools on a large collection of software[86]. For instance, an analysis of the memory at runtime would allow the detection of relevant data structures such as known trigonometric, logarithm, quarter-square or screen memory addresses tables (see sections 6.2.1, 6.2.2 and 6.3). Furthermore, by analysing the sequential flow of instructions, we can detect the use of undocumented instructions, self-modifying code as well as the execution of unrolled loops (see sections 7.3, 7.4 and 7.5). This large scale study would have the direct benefit of allowing us to quantify over time the use of these techniques. Another possibility would be to extract memory content at runtime, and identify common data and code sections with data mining techniques [Ber06]. This could be used in the context of authorship attribution – where the author can be a programmer or more generally a software development company – as programmers and software development companies reuse some of their assets in their different projects [RZM11]. The reverse engineering of several games on the ZX Spectrum developed by Ultimate Play The Game (Atic Atac, Cookie, Jetpac, Lunar Jetman, Pssst and Tranz Am) shows that some routines are being reused across games. For instance, Atic Atac and Lunar Jetman use the same decryption routine [Mad]. By generalizing this process, it would allow a better understanding of the evolution of programming techniques – whether from a qualitative or quantitative point of view – on 8-bit systems.

Acknowledgement

I would like to thank Linus Åkesson (Lft*), Arnaud Storq (NoRecess*) and Stéphane Sikora (Siko*) for reviewing this publication with care and providing insightful comments.

License

[86] VICE emulator supports a binary monitor that can be controlled over a dedicated connection, which theoretically allows an instrumented and automated runtime analysis [VIC].

9 Bibliography

References

[4am15a] 4am. *Mr. Do (4am crack)*. 2015.

[4am15b] 4am. *Pac-Man (Datasoft) (4am crack)*. 2015.

[AS64] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Applied mathematics series. U.S. Government Printing Office, 1964. URL: https://books.google.com/books?id=EMIWTKQXSZUC.

[Act93] Amstrad Action. "Show offs!" In: *Amstrad Action* (89 Mar. 1993), pp. 30–33.

[Adv73] British Association for the Advancement of Science. Committee on mathematical tables. *Report of the Committee on Mathematical Tables*. Taylor & Francis, 1873. URL: https://books.google.com/books?id=bXw3AAAAMAAJ.

[Ahl83] D. H. Ahl. "Benchmark Comparison Test". In: *Creative Computing* 9.11 (Nov. 1983), pp. 259–260.

[Ahl84] D. H. Ahl. "Creative Computing Benchmark". In: *Creative Computing* 10.3 (Mar. 1984), p. 6.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Inc., 1986. ISBN: 0-201-10088-6.

[AU77] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley Publishing Company, Inc., 1977. ISBN: 0-201-00022-9.

[Alb] J. Albo. *Pasmo, ensamblador cruzado Z80 portable / portable Z80 cross assembler*. URL: https://pasmo.speccy.org/ (visited on 07/08/2023).

[Ams84a] Amsoft. *Amsoft Devpac*. Amsoft, 1984.

[Ams84b] Amstrad Consumer Electronics, PLC. *Amstrad CPC 464 User Instructions*. Amstrad Consumer Electronics, PLC, 1984.

[AJ83a] I. O. Angell and B. J. Jones. *Advanced Graphics with the BBC Model B Microcomputer*. The Macmillan Press Ltd, 1983. ISBN: 978-1-349-06766-4.

[AJ83b] I. O. Angell and B. J. Jones. *Advanced Graphics with the Sinclair ZX Spectrum*. The Macmillan Press Ltd, 1983. ISBN: 0333350502.

[App78a] Apple Computer, Inc. *Apple II Reference Manual*. Apple Computer, Inc., Jan. 1978.

[App79] Apple Computer, Inc. *Apple II Reference Manual*. Apple Computer, Inc., 1979.

[App76] Apple Computer, Inc. “Apple Introduces the First Low Cost Microcomputer System with a Video Terminal and 8K Bytes of RAM on a single PC Card.” In: *Interface Age* (Oct. 1976), p. 11.

[App78b] Apple Computer, Inc. *Applesoft II Reference Manual*. Apple Computer, Inc., Aug. 1978.

[App77] Apple Computer, Inc. “Introducing Apple II.” In: *BYTE* 2.6 (June 1977), pp. 14–15.

[App78c] Apple Computer, Inc. *Programmer's Aid #1*. Apple Computer, Inc., 1978.

[Arn85] Arnor Ltd. *MAXAM Z80 Development System*. Arnor Ltd., 1985.

[Ata83a] Atari, Inc. *1200 XL home computer field service manual*. Atari, Inc., Feb. 1983.

[Ata83b] Atari, Inc. *2600/2600A VCS domestic (M/N) field service manual*. Atari, Inc., Jan. 1983.

[Ata83c] Atari, Inc. *600 XL home computer field service manual*. Atari, Inc., Oct. 1983.

[Ata83d] Atari, Inc. *Atari home computers, the next generation*. 1983.

[Ata83e] Atari, Inc. *ATARI Microsoft BASIC II Reference Manual*. Atari, Inc., 1983.

[Ata81] Atari, Inc. *Atari Personal Computer Product Catalog*. 1981.

[Auc96] D. Aucsmith. “Tamper resistant software: an implementation”. In: *Information Hiding*. Ed. by R. Anderson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 317–333. ISBN: 978-3-540-49589-5.

[Ayc06] J. Aycock. *Computer Viruses and Malware*. Vol. 22. Advances in Information Security. Springer, 2006. ISBN: 978-0-387-30236-2. DOI: 10.1007/0-387-34188-9. URL: https://doi.org/10.1007/0-387-34188-9.

[Ayc16] J. Aycock. *Retrogame Archeology*. Springer, Jan. 2016. ISBN: 978-3-319-30002-3. DOI: 10.1007/978-3-319-30004-7.

[Bac78] J. Backus. “Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs”. In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 613–641. ISSN: 0001-0782. DOI: 10.1145/359576.359579. URL: https://doi.org/10.1145/359576.359579.

[Bas+81] C. J. Bashe et al. “The Architecture of IBM's Early Computers”. In: *IBM J. Res. Dev.* 25.5 (Sept. 1981), pp. 363–376.

[Ben82] J. L. Bentley. *Writing Efficient Programs*. Prentice-Hall, Inc., 1982. ISBN: 0139702512.

[Ber06] P. Berkhin. "A Survey of Clustering Data Mining Techniques". In: *Grouping Multidimensional Data: Recent Advances in Clustering*. Ed. by J. Kogan, C. Nicholas, and M. Teboulle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 25–71. ISBN: 978-3-540-28349-2. DOI: 10.1007/3-540-28349-8_2. URL: https://doi.org/10.1007/3-540-28349-8_2.

[Bir84] I. Birnbaum. *Assembly Language Programming for the BBC Microcomputer*. Macmillan microcomputer books. Macmillan, 1984. ISBN: 9780333370964. URL: https://books.google.com/books?id=mychAAAACAAJ.

[Bit22] Bitbreaker. *Advanced optimizing*. 2022. URL: https://codebase64.org/doku.php?id=base:advanced_optimizing (visited on 07/08/2023).

[Bra11] D. Braben. *Classic Game Postmortem - ELITE*. 2011. URL: https://www.gdcvault.com/play/1014628/Classic-Game-Postmortem (visited on 07/08/2023).

[BDH83] A. C. Bray, A. C. Dickens, and M. A. Holmes. *The Advanced User Guide for the BBC Microcomputer*. Cambridge Microcomputer Centre, 1983. ISBN: 9780946827008. URL: https://books.google.com/books?id=owyONAAACAAJ.

[Bri24] H. Briggs. *Arithmetica logarithmica*. 1624.

[Bri17] H. Briggs. *Logarithmorum Chilias Prima*. 1617.

[BYT77] BYTE. "Commodore's New PET Computer". In: *BYTE* 2.10 (Oct. 1977), p. 50.

[CSV07] H. Cai, Z. Shao, and A. Vaynberg. "Certified Self-Modifying Code". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 66–77. ISBN: 9781595936332. DOI: 10.1145/1250734.1250743. URL: https://doi.org/10.1145/1250734.1250743.

[Cam05] B. Camper. "Homebrew and the social construction of gaming : community, creativity, and legal context of amateur Game Boy Advance development". MA thesis. 2005.

[Cle55] C. W. Clenshaw. "A note on the summation of Chebyshev series". In: *Mathematics of Computation* 9 (1955), pp. 118–120.

[Cle54] C. W. Clenshaw. "Polynomial approximations to elementary functions". In: *Mathematics of Computation* 8 (1954), pp. 143–147.

[Coh87] F. Cohen. "Computer viruses: Theory and experiments". In: *Computers & Security* 6.1 (1987), pp. 22–35. ISSN: 0167-4048. DOI: https://doi.org/10.1016/0167-4048(87)90122-2.

[Col78a] J. Coll. "Benchmarks again and connections". In: *Personal Computer World* 1.7 (Nov. 1978), p. 52.

[Col78b] J. Coll. “Direct addressing: where to get your personal computer”. In: *Personal Computer World* 1.1 (1978), pp. 55–58.

[CAL84] J. Coll, D. Allen, and Acorn Computers Limited. *BBC Microcomputer System User Guide*. British Broadcasting Corporation, 1984. ISBN: 9780563211716. URL: https://books.google.com/books?id=aYgxOwAACAAJ.

[Com85] Commodore Business Machines, Inc. *6500 microprocessors*. Commodore Business Machines, Inc., Nov. 1985.

[Com82a] Commodore Business Machines, Inc. *6510 microprocessor with I/O*. Commodore Business Machines, Inc., Nov. 1982.

[Com82b] Commodore Business Machines, Inc. *Commodore 64 Programmer's Reference Guide*. SAMS, 1982. ISBN: 0672220563.

[Com82c] Commodore Business Machines, Inc. *Commodore 64 User's Guide*. 1982.

[Con20] K. Connell. *Abug Talks - Kieran Connell talks about Coding the BEEB-NICCC demo on the BBC Micro*. 2020. URL: https://www.youtube.com/watch?v=hbfLIeGrkNI (visited on 07/08/2023).

[Cru15] Cruzer. *Speedcode a.k.a. Loop Unrolling*. 2015. URL: https://codebase64.org/doku.php?id=base:speedcode (visited on 07/08/2023).

[Cus75] R. H. Cushman. In: *EDN* (Sept. 1975), pp. 36–42.

[Der06] J. Derbyshire. *Unknown Quantity: A Real and Imaginary History of Algebra*. The National Academies Press, 2006. ISBN: 978-0-309-09657-7.

[Dig76a] Microcomputer Digest. “MOS Technology drops 6501”. In: *Microcomputer Digest* 2.11 (May 1976), p. 4.

[Dig75] Microcomputer Digest. “Motorola sues MOS Technology”. In: *Microcomputer Digest* 2.6 (Dec. 1975), p. 11.

[Dig76b] Microcomputer Digest. “The "super 8080" microcomputer”. In: *Microcomputer Digest* 3.1 (July 1976), pp. 1–2.

[Dis04] P. Diskin. *Nintendo Entertainment System Documentation*. Tech. rep. Aug. 2004.

[Dom17] C. Domas. “Breaking the x86 ISA”. In: 2017.

[Dre11] Ninja/The Dreams. *28c3: Behind the scenes of a C64 demo*. 2011. URL: https://www.youtube.com/watch?v=po9IY5Kf0Mo (visited on 07/08/2023).

[Dym20a] R. Dymond. *The complete Jet Set Willy RAM disassembly*. June 2020. URL: https://skoolkit.ca/disassemblies/jet_set_willy/hex/index.html (visited on 07/08/2023).

[Dym20b] R. Dymond. *The complete Manic Miner RAM disassembly*. July 2020. URL: https://skoolkit.ca/disassemblies/manic_miner/hex/index.html (visited on 07/08/2023).

[Fyl75] D. Fylstra. "Son of Motorola (or the $20 CPU Chip)". In: *BYTE* 0.3 (Nov. 1975), pp. 56–62.

[Gil81] J. Gilbreath. "A High-Level Language Benchmark". In: *BYTE* 6.9 (Sept. 1981), pp. 180–198.

[GG83] J. Gilbreath and G. Gilbreath. "Eratosthenes Revisited". In: *BYTE* 8.1 (Jan. 1983), pp. 283–326.

[GOR86] B. Godden, P. Overell, and D. Radisic. *Soft968 - CPC 464/664/6128 Firmware*. Amstrad Consumer Electronics, PLC, 1986.

[GH93] M. D. Godfrey and D. F. Hendry. "The Computer as von Neumann Planned It". In: *IEEE Annals of the History of Computing* 15.1 (Jan. 1993), pp. 11–21. ISSN: 1058-6180. DOI: 10.1109/85.194088. URL: https://doi.org/10.1109/85.194088.

[Gol19a] Golara. 2019. URL: https://gitlab.com/pseregiet/desire-4k-for-moonshine/-/tree/master (visited on 07/08/2023).

[Gol19b] Golara. 2019. URL: https://gitlab.com/pseregiet/desire-fastline-c64-intro/-/tree/master (visited on 07/08/2023).

[Gra16] Graham. *Table generator for square table based multiplications*. 2016. URL: https://codebase64.org/doku.php?id=base:table_generator_routine_for_fast_8_bit_mul_table (visited on 07/08/2023).

[Gre] P. Grenfell. URL: https://bitbucket.org/evilpaul/zx101/src/master/source/modules/twister.asm (visited on 07/08/2023).

[Gre17] P. Grenfell. *ARTtech Seminar: Paul Grenfell - Hacking Retro Games - When Is A Trainer Like A Space Rocket?* 2017. URL: https://www.youtube.com/watch?v=XdS_XvTOQJY (visited on 07/08/2023).

[Gre18] P. Grenfell. *ARTtech seminars - Paul Grenfell: Retro demo scene coding - ZX Spectrum 101*. 2018. URL: https://www.youtube.com/watch?v=M6Zi8xLCbMs (visited on 07/08/2023).

[Gro20] Groepaz. *NMOS 6510 Unintended Opcodes*. Dec. 2020.

[Hag18] J. Hague. *Halcyon Days: Interviews with Classic Computer and Video Game Programmers*. Nov. 2018. URL: https://dadgum.com/halcyon/ (visited on 07/08/2023).

[Ham+49] F. E. Hamilton et al. "Selective Sequence Electronic Calculator". 2,636,672. Jan. 1949.

[Har10] J. G. Harston. *BBC OS 1.20 Disassembly*. 2010. URL: https://mdfs.net/Docs/Comp/BBC/OS1-20/ (visited on 07/08/2023).

[Har+78] J. F. Hart et al. *Computer Approximations*. SIAM series in applied mathematics. R. E. Krieger Publishing Company, 1978. ISBN: 9780882756424. URL: https://books.google.com/books?id=7vVQAAAAMAAJ.

[Has76] B. Hashizume. "Microprocesor Update: Zilog Z80". In: *BYTE* (12 Aug. 1976), pp. 34–38.

[HP17] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.

[HiS87] HiSoft. *HiSoft Devpac*. HiSoft, 1987.

[Høy02] J. Høyrup. "A Note on Old Babylonian Computational Techniques". In: *Historia Mathematica* 29.2 (2002), pp. 193–198.

[Int75a] Intel Corporation. *intel 8080 Assembly Language Programming Manual*. Intel Corporation, Sept. 1975.

[Int75b] Intel Corporation. *intel 8080 Microcomputer Systems User's Manual*. Intel Corporation, Sept. 1975.

[Jac15] Jackasser. *Seriously fast multiplication (8-bit and 16-bit)*. 2015. URL: https://codebase64.org/doku.php?id=base:seriously_fast_multiplication (visited on 07/08/2023).

[JSS10] G. James, B. Silverman, and B. Silverman. "Visualizing a Classic CPU in Action: The 6502". In: *ACM SIGGRAPH 2010 Talks*. SIGGRAPH '10. Los Angeles, California: Association for Computing Machinery, 2010. ISBN: 9781450303941. DOI: 10.1145/1837026.1837061. URL: https://doi.org/10.1145/1837026.1837061.

[Jam84] M. James. *An Expert Guide to the Spectrum*. Granada Publishing Ltd, 1984. ISBN: 0246122781.

[JM86] J. W. Janneck and T. Mossakowski. *ROM-Listing CPC 464, 664, 6128: ausführl. dokumentiertes Listing aller Betriebssystem- u. BASIC-Routinen ; detaillierte Hintergrundinformationen zu Speicheraufteilung, Z 80, VideoRAM, Schaltplänen ; die Unterschiede zum CPC 664/6128-BASIC-Betriebssystem werden gesondert dokumentiert*. Markt-&-Technik-Verlag, 1986. ISBN: 9783890901343. URL: https://books.google.com/books?id=SRSqPQAACAAJ.

[Jon85] D. Jones. "Hidden Extras". In: *Your Spectrum* (18 Sept. 1985), pp. 51–52.

[KJP90] K.J.P.B. *The Kracker Jax Revealed Trilogy*. 1990.

[KEH93] D. Keppel, S. J. Eggers, and R. R. Henry. *A case for runtime code generation*. Tech. rep. 1993.

[Kew84] G. Kewney. "Amstrad CPC464". In: *Personal Computer World* 7.5 (May 1984), pp. 170–176.

[KA] F. Kjolstad and A. Aiken. *CS143 Lecture 14 Intermediate Code & Local Optimizations*. URL: https://web.stanford.edu/class/cs143/lectures/lecture14.pdf (visited on 07/08/2023).

[Knu68] D. E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. First. Addison-Wesley, 1968.

[Knu97] D. E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Third. Addison-Wesley, 1997.

[Lap09] P. S. Laplace. "Mémoire sur divers points d'analyse". In: *Journal de l'École Polytechnique, XVème Cahier, Tome VIII* (1809).

[Lin] M. Lind. URL: https://bitbucket.org/magli143/exomizer/wiki/Home (visited on 07/08/2023).

[LO83] I. Logan and F. O'Hara. *The Complete Spectrum ROM Disassembly*. Melbourne House, 1983. ISBN: 9780867591170. URL: https://books.google.com/books?id=xLbcAAAACAAJ.

[Lud90] J. H. Ludolf. *Tetragonometria tabularia, qua per tabulas quadratorum, a radice 1 usque ad 100000 simplici additionis, subtractionis & dimidiationis beneficio, numeri figurati quilibet tum plani polygonii tum solidi & cossici inveniri atque radices eorum extrahi possunt: cum novis & variis usibus arithmetico-geometricis*. typis Groschianis, 1690. URL: https://books.google.com/books?id=7zzp3POzxRsC.

[MM83] D. MacGregor and B. Moyer. "Built-in tight-loop mode raises µP's performance". In: *Electronic Design* 31.22 (1983), pp. 225–231.

[Mad] P. Maddern. URL: https://pobtastic.github.io/ultimate/ (visited on 07/08/2023).

[Man83] S. Mann. "Electron". In: *Personal Computer World* 6.10 (Oct. 1983), pp. 160–167.

[MKP11] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel. "A Taxonomy of Self-Modifying Code for Obfuscation". In: *Computers and Security* 30.8 (Nov. 2011), pp. 679–691. ISSN: 0167-4048. DOI: 10.1016/j.cose.2011.08.007. URL: https://doi.org/10.1016/j.cose.2011.08.007.

[Mec89] J. Mechner. *Prince-of-Persia-Apple-II*. 1989. URL: https://github.com/jmechner/Prince-of-Persia-Apple-II/ (visited on 07/08/2023).

[Mor82] C. Morgan. "NCC Report". In: *BYTE* 7.9 (Sept. 1982), pp. 58–61.

[Mor77] C. Morgan. "The TRS-80: Radio Shack's New Entry into the Personal Computer Market". In: *BYTE* 2.11 (Nov. 1977), p. 46.

[Mor83] G. Morrison. *Atari Software Protection Techniques*. Alpha Systems, 1983.

[Mor86] M. Morton. "Tricks and traps". In: *BYTE* 11.9 (Sept. 1986), pp. 163–172.

[MOS76a] MOS Technology, Inc. *MCS6500 Microcomputer Family Hardware Manual, Second Edition*. MOS Technology, Inc., Jan. 1976.

[MOS76b] MOS Technology, Inc. *MCS6500 Microprocessors*. MOS Technology, Inc., May 1976.

[MOS75a] MOS Technology, Inc. *MCS6501 - MCS6505 Microprocessors*. MOS Technology, Inc., Aug. 1975.

[MOS75b] MOS Technology, Inc. "MOS 6502 The Second of a Low Cost High Performance Microprocessor Family". In: *Computer* 8.9 (1975), pp. 38–39. DOI: 10.1109/C-M.1975.219074.

[Mot88] Motorola, Inc. *M68000 Family Reference*. Motorola, Inc., 1988.

[Mot84a] Motorola, Inc. *MC6800*. Motorola, Inc., 1984.

[Mot84b] Motorola, Inc. *MC68020 32-Bit Microprocessor User's Manual*. Motorola, Inc., 1984.

[Mot89] Motorola, Inc. *MC68030 Enhanced 32-Bit Microprocessor User's Manual Second Edition*. Motorola, Inc., 1989.

[Mot84c] Motorola, Inc. *MC6845*. Motorola, Inc., 1984.

[Mot92] Motorola, Inc. *Motorola M68000 Family Programmer's Reference Manual*. Motorola, Inc., 1992.

[Mox20a] M. Moxon. *Multiplication and division using logarithms*. 2020. URL: https://www.bbcelite.com/deep_dives/multiplication_and_division_using_logarithms.html (visited on 07/08/2023).

[Mox20b] M. Moxon. *Shift-and-add multiplication*. 2020. URL: https://www.bbcelite.com/deep_dives/shift-and-add_multiplication.html (visited on 07/08/2023).

[Mox20c] M. Moxon. *Shift-and-subtract division*. 2020. URL: https://www.bbcelite.com/deep_dives/shift-and-subtract_division.html (visited on 07/08/2023).

[Mox20d] M. Moxon. *Version analysis of MULT1*. 2020. URL: https://www.bbcelite.com/compare/main/subroutine/mult1.html (visited on 07/08/2023).

[Mus07a] Computer History Museum. *Oral History Panel on the Development and Promotion of the Intel 8080 Microprocessor*. Apr. 2007. URL: https://archive.computerhistory.org/resources/access/text/2013/05/102658123-05-01-acc.pdf (visited on 07/08/2023).

[Mus07b] Computer History Museum. *Zilog Oral History Panel on the Founding of the Company and the Development of the Z80 Microprocessor*. Apr. 2007. URL: http://archive.computerhistory.org/resources/text/Oral_History/Zilog_Z80/102658073.05.01.pdf (visited on 07/08/2023).

[Nap14] J. Napier. *Mirifici logarithmorum canonis descriptio*. A. Hart, 1614. URL: https://books.google.com/books?id=J19dAAAAcAAJ.

[Neu93] J. von Neumann. “First Draft of a Report on the EDVAC”. In: *IEEE Annals of the History of Computing* 15.4 (Oct. 1993), pp. 27–75. ISSN: 1058-6180. DOI: 10.1109/85.238389. URL: https://doi.org/10.1109/85.238389.

[Nie] M. Nielsen. *Kick Assembler Reference Manual*. URL: http://www.theweb.dk/KickAssembler/KickAssembler.pdf (visited on 07/08/2023).

[Nit85] W. Nitschke. *Advanced Z80 machine code programming*. Interface Publications, 1985.

[OR00] J. J. O’Connor and E. F. Robertson. *An overview of Babylonian mathematics*. Dec. 2000. URL: https://mathshistory.st-andrews.ac.uk/HistTopics/Babylonian_mathematics/ (visited on 07/08/2023).

[Par99] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Inc., 1999. ISBN: 0195125835.

[Pic92] Pict. “Logon System”. In: *Amstrad Cent Pour Cent* (45 Nov. 1992), pp. 34–36.

[Pic93a] Pict. “Logon System”. In: *Amstrad Cent Pour Cent* (46 Jan. 1993), pp. 46–47.

[Pic93b] Pict. “Logon System”. In: *Amstrad Cent Pour Cent* (47 Apr. 1993), pp. 18–21.

[Pie86] F. Pierot. *Graphisme en Assembleur sur Amstrad CPC*. Éditions du P.S.I., 1986. ISBN: 2865953408.

[Pro08] C. Proust. “Quantifier et calculer : usages des nombres à Nippur”. In: *Revue d’histoire des mathématiques* 14.2 (2008), pp. 143–209.

[Put86] C. Putney. “Fastest 6502 Multiplication Yet”. In: *Apple Assembly Line* 6.6 (Mar. 1986).

[Ras78] J. Raskin. *Apple II Basic Programming Manual*. Apple Computer, Inc., 1978.

[Reu10] M. Reunanen. “Computer Demos - What Makes Them Tick?” In: (2010).

[Roe11] D. Roegel. *A reconstruction of Briggs’ Logarithmorum chilias prima (1617)*. Tech. rep. 2011. URL: https://hal.inria.fr/inria-00543935/en.

[Roe13a] D. Roegel. *A reconstruction of Ludolfs’s Tetragonometria tabularia (1690)*. Tech. rep. 2013. URL: https://hal.inria.fr/hal-00880845.

[Roe13b] D. Roegel. *A reconstruction of Voisin’s table of quarter-squares (1817)*. Tech. rep. 2013. URL: https://hal.inria.fr/hal-00812834.

[RZM11] N. Rosenblum, X. Zhu, and B. Miller. “Who Wrote This Code? Identifying the Authors of Program Binaries”. In: Sept. 2011, pp. 172–189. ISBN: 978-3-642-23821-5. DOI: 10.1007/978-3-642-23822-2_10.

[RF77a] T. Rugg and P. Feldman. “BASIC Timing Comparison ...information for speed freaks”. In: *Kilobaud Microcomputing* (June 1977), pp. 66–77.

[RF77b] T. Rugg and P. Feldman. “BASIC Timing Comparison ...revisited and updated”. In: *Kilobaud Microcomputing* (Oct. 1977), pp. 20–25.

[Rus85] J. Ruston. *The BBC Micro Compendium*. Interface, 1985. ISBN: 9780907563334. URL: `https://books.google.com/books?id=nyU8NAAACAAJ`.

[Seg92] Sega Enterprises, Ltd. *Sega Game Gear color portable video game system maintenance manual Europe*. Sega Enterprises, Ltd., Aug. 1992.

[Seg] Sega Enterprises, Ltd. *Sega Master System Service Manual*. Sega Enterprises, Ltd.

[Seg94] Sega Enterprises, Ltd. *Sega Service Manual Genesis, Mega Drive PAL, Mega CD/Sega CD*. Sega Enterprises, Ltd., Apr. 1994.

[She83] J. C. Shepherd. “Extra Instructions”. In: *Compute!* 5.41 (Oct. 1983), pp. 261–264.

[Sim+85] T. N. Simstad et al. *Program Protection Manual For the C-64 Volume II*. CSM Software, Inc., 1985.

[Sin83] I. Sinclair. *Introducing Spectrum Machine Code*. Granada Publishing Ltd, 1983. ISBN: 0-246-12082-7.

[Sta82] J. Stanton. *Apple Graphics & Arcade Game Design*. The Book Co., 1982.

[SP84] J. Stanton and D. Pinal. *Atari Graphics and Arcade Game Design*. Arrays Inc., 1984. ISBN: 0912003057.

[Sta83] T. W. Starnes. “Design Philosophy Behind Motorola’s MC68000, Part 3: Advanced instructions”. In: *BYTE* 8.6 (June 1983), pp. 339–349.

[Ste20] M. Steil. *BASIC & KERNAL ROM Disassembly*. 2020. URL: `https://www.pagetable.com/c64ref/c64disasm/` (visited on 07/08/2023).

[Ste08] M. Steil. *Create your own Version of Microsoft BASIC for 6502*. 2008. URL: `https://www.pagetable.com/?p=46` (visited on 07/08/2023).

[Ste15] M. Steil. *Microsoft BASIC for 6502 Original Source Code [1978]*. 2015. URL: `https://www.pagetable.com/?p=774` (visited on 07/08/2023).

[SS84] A. P. Stephenson and D. J. Stephenson. *Advanced machine code programming for the Commodore 64*. Granada Publishing Ltd, 1984. ISBN: 0246124423.

[SW14] K. Stuart and D. Wall. *Sega Mega Drive/Genesis: Collected Works*. Read-Only Memory Ltd, 2014. ISBN: 978-0-9575768-1-0.

[Szo05] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005. ISBN: 0321304543.

[Tay15] B. Taylor. *Methodus incrementorum directa et inversa*. 1715. URL: https://books.google.com/books?id=oA5tpwAACAAJ.

[Teb82] D. Tebbutt. "Sinclair ZX Spectrum". In: *Personal Computer World* 5.6 (June 1982), pp. 118–124.

[The85] The Institute of Electrical and Electronics Engineers, Inc. "IEEE Standard for Binary Floating-Point Arithmetic". In: *ANSI/IEEE Std 754-1985* (Aug. 1985), pp. 1–20. DOI: 10.1109/IEEESTD.1985.82928.

[Tho85] J. W. Thomas. "Floating-point Arithmetic in Applesoft BASIC". In: *Call-A.P.P.L.E.* (July 1985).

[Tho70] J. E. Thornton. *Design of a Computer: The Control Data 6600*. Courant Institute of Mathematical Sciences New York University series. Scott, F., 1970. URL: https://books.google.com/books?id=eWhTAAAAMAAJ (visited on 07/08/2023).

[TY17] T. Touili and X. Ye. "Reachability Analysis of Self Modifying Code". In: *2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*. 2017, pp. 120–127. DOI: 10.1109/ICECCS.2017.19.

[TWW16] TWW. *Plotting Pixels in HiRes Bitmap*. 2016. URL: https://codebase64.org/doku.php?id=base:dots_and_plots (visited on 07/08/2023).

[VIC] VICE Team. *VICE*. URL: https://sourceforge.net/p/vice-emu/ (visited on 07/08/2023).

[VB85] S. Vickers and R. Bradbeer. *ZX Spectrum Basic Programming*. 1985. URL: https://books.google.com/books?id=YqmYtAEACAAJ.

[Vla28] A. Vlacq. *Arithmetica logarithmica*. 1628.

[Wag83] R. Wagner. "Assembly Lines". In: *Softalk* 3.10 (June 1983), pp. 199–201.

[Web84] M. Webb. "Processor progress". In: *A&B Computing* 1.8 (Aug. 1984), pp. 42–46.

[Whe77] G. Wheeler. "Undocumented M6800 Instructions". In: *BYTE* 2.12 (Dec. 1977), pp. 46–47.

[WC78] G. Wideman and M. Czerwinski. "Inside the Commodore PET". In: *Electronic Today* 2.2 (Feb. 1978), pp. 10–16.

[Wik20] C64 Wiki. *POLY1*. 2020. URL: https://www.c64-wiki.com/wiki/POLY1 (visited on 07/08/2023).

[Wil20] B. Wilson. *Z80 Optimization*. 2020. URL: `https://wikiti.brandonw.net/index.php?title=Z80_Optimization` (visited on 07/08/2023).

[Wor85] P. Worlock. "Commodore 128". In: *Personal Computer World* 8.7 (July 1985), pp. 136–141.

[Woz77] S. Wozniak. "The Apple-II". In: *BYTE* 2.5 (May 1977), pp. 34–43.

[WA79] S. Wozniak and Apple Computer, Inc. *The Wozpak][and other assorted goodies*. Apple Puget Sound Program Library Exchange, 1979.

[Wsz83] S. Wszola. "Commodore 64". In: *BYTE* 8.7 (July 1983), pp. 232–246.

[You05] S. Young. *The Undocumented Z80 Documented*. Tech. rep. Sept. 2005.

[Zak78] R. Zaks. *Programming the 6502*. Sybex, 1978.

[Zak80] R. Zaks. *Programming the Z80 second edition*. Sybex, 1980.

[Zik01] Zik. "Assembleur: Coding". In: *Quasar CPC* 19 (2001), pp. 16–17.

[Zil84] Zilog, Inc. *Z80 Family Product Specifications Handbook*. Zilog, Inc., 1984.

[Zil76] Zilog, Inc. *Z80-CPU Technical Manual*. Zilog, Inc., 1976.

[Zil77] Zilog, Inc. *Z80-CPU Z80A-CPU Technical Manual*. Zilog, Inc., 1977.