History of Computing

Raúl Rojas

# Konrad Zuse's Early Computers

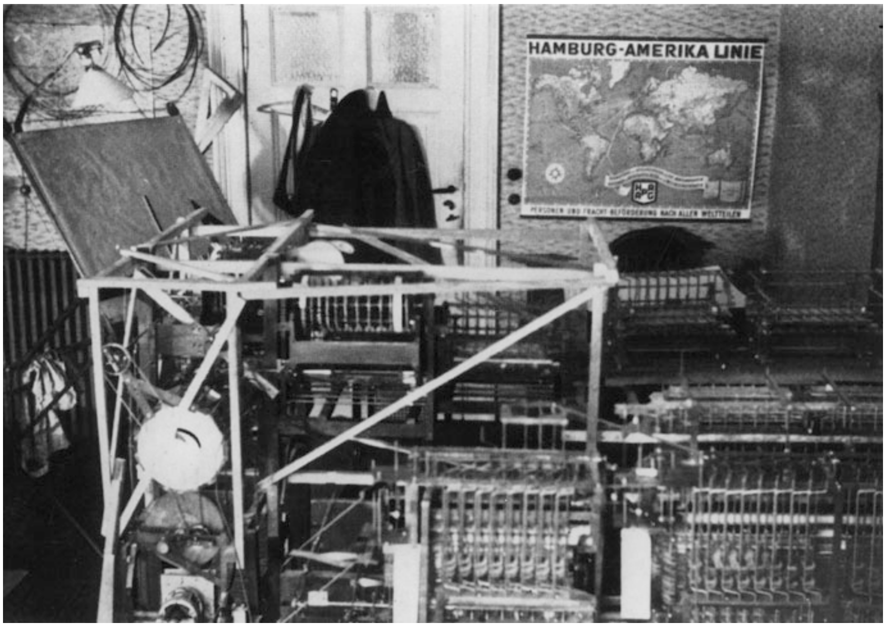
The Quest for the Computer in Germany

Springer

Raúl Rojas

# Konrad Zuse's Early Computers

The Quest for the Computer in Germany



Springer

Raúl Rojas
Nevada, USA

ISSN 2190-6831 ISSN 2190-684X (electronic)
History of Computing
ISBN 978-3-031-39875-9 ISBN 978-3-031-39876-6 (eBook)
https://doi.org/10.1007/978-3-031-39876-6

*Quando orientur controversiae, non magis disputatione opus erit inter duos philosophus, quam inter duos computistas. Sufficiet enim calamos in manus sumere sedereque ad abacos, et sibi mutuo (accito si placet amico) dicere: calculemus.*[1]

*Gottfried Wilhelm Leibniz*

[1] "If controversies were to arise, there would be no more need of disputation between two philosophers than between two calculators. For it would suffice for them to take their pencils in their hands and to sit down at the abacus, and say to each other (and if they so wish also to a friend called to help): Let us calculate."

# Foreword

This book describes the historical development of the architectures of the first computers built by the German inventor Konrad Zuse in the period 1936–1945. Although these machines are prominent in Germany, this is not the case in other countries. In many books on the history of the computer, Zuse's work receives only passing mention. However, as the various chapters in this volume show, the kind of computer architecture that Zuse developed is closer to modern computers than the architectures of the Harvard Mark I or the ENIAC, the two American machines most often celebrated as the world's first computers.

Over the years, I have published most of the material in this book as articles in academic journals, Internet sites, and conference proceedings. I started writing about Zuse's machines in 1994, so putting this book together meant reorganizing all the contributions in a coherent way. Some articles published in German have been translated for this volume. Each chapter contains references to the original publications. The advantage for the reader is that this collection brings together all stages of an amazing intellectual puzzle, the invention of the computer, no less, into a single volume.

For this book, I have chosen to keep each chapter as a stand-alone piece, so that they can be read in any order. Sufficient redundancy has been provided with explanations at the beginning of each chapter to ensure clarity of context. The reader can think of this book as a collection of essays that, after thirty years of research, are now closely interwoven.

To make the book easier to read, the preface has been written as a kind of "executive summary" containing the most important general information and chronology. It is intended for the super-busy reader. Then, for those who are just busy, the first chapter gives a comprehensive overview of the computers Zuse built from 1936 to 1945, that is, the Z1, Z2, Z3, and Z4, as well as other more specialized machines. Subsequent chapters deal with the architecture of each computer, culminating in the description of Plankalkül, the first proposal for a high-level programming language.

It is my sincere hope that the curious reader will venture out and peruse the entire book. Some chapters are easier to understand than others. For example, the

Z1 computer, the mechanical calculator, is more difficult to digest than the relay machines, the Z3 or Z4. After the overview in Chap. 1, the reader can skip to the chapters on the Z3 or Z4 if she or he prefers, and then return to the chapter on the Z1. Chapter 2 is easy to follow: it describes the historical circumstances for the development of electronic computers in the USA and Europe.

This book is for the curious and the adventurous. Students and practitioners of computer science should have no trouble following the material in all chapters. Readers from other disciplines can certainly get the main message, perhaps by adopting the nonlinear reading strategy mentioned above. Start with the concise summary that follows, and you will be well on your way to retracing the steps of a remarkable intellectual adventure.

Reno, USA
August 2023

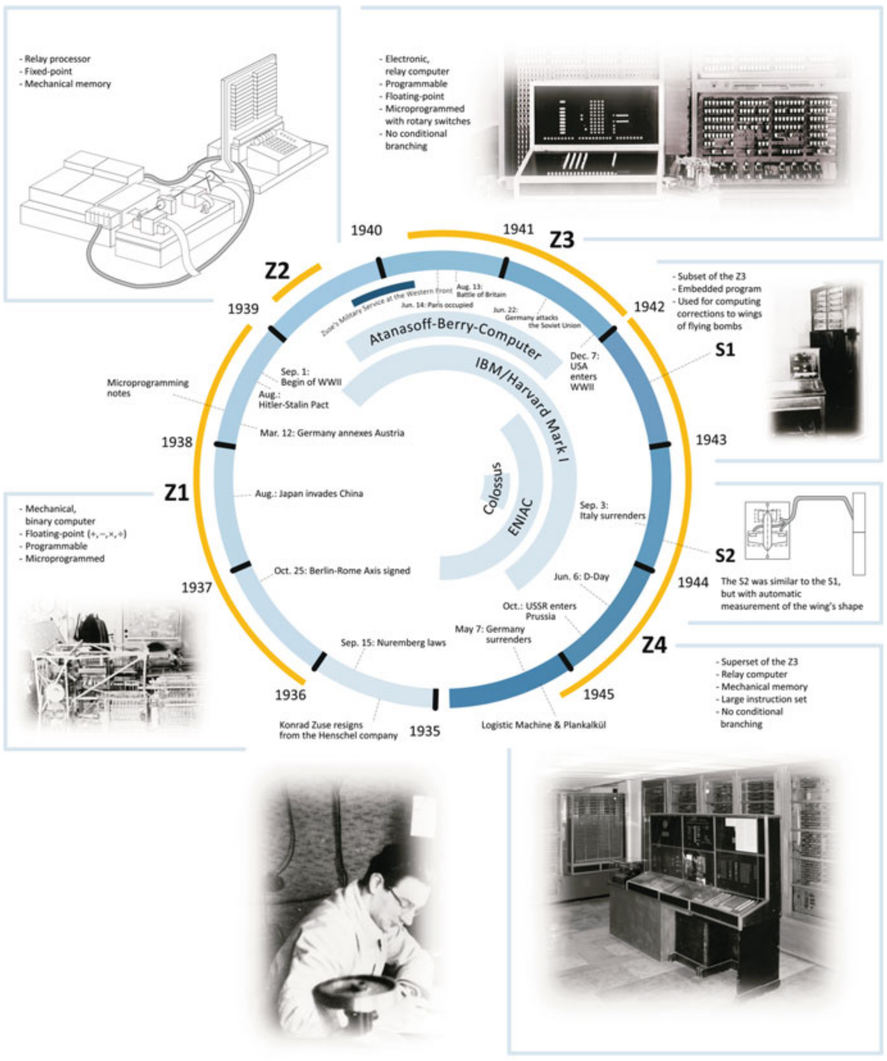
Raúl Rojas

# Preface

Konrad Ernst Otto Zuse was born in Berlin in 1910 as the son of Emil and Maria Zuse. He is generally considered the father of the computer in Germany. He started thinking about automating computations as a student in the mid-1930s and built a mechanical computer from 1936 to 1938/1939. The machine was called V1 (*Versuchsmodell 1*, i.e. Experimental Model 1). By the end of the war, he had built three more important computers: the V2, V3, and V4. The four were renamed Z1, Z2, Z3, and Z4 to avoid any association with Wernher von Braun's V2 rockets. While the Z1 was a purely mechanical device (based on what Zuse called "mechanical relays"), the processor logic of the Z2, Z3, and Z4 was based on electromagnetic relays. Nevertheless, the mechanical memory of the Z2 and Z4 still used sliding metal components as two-state memory elements. The logic design of all four machines was completely binary: the decimal input was converted to base two for all internal calculations in the processor. The result was converted back to decimal for display (except in the Z2, which displayed the result as 16 bits).

The Zuse machines Z1, Z3, and Z4 used the floating-point representation, where numbers are stored as a binary mantissa with its sign and an exponent of base two (for example, $+1.010 \times 2^3$). The Z2 transitional machine was a fixed-point prototype, a proof of concept. All four computers had a processor, a memory, an input keyboard, and a visual display for results. Programs were punched, instruction by instruction, on 35 mm film tape. The processor of the Z1 could perform the four basic arithmetic operations, the Z2 only a subset. The Z3 could also extract square roots of numbers. The Z4 had a much larger instruction set than any of the other machines.

Zuse was drafted twice during the war. On both occasions, he was able to get his discharge from the front so that he could work on structural analysis for the Henschel Flugzeug-Werke, while at the same time continuing to build his calculating machines through his own company. He built two small special machines for the military (called S1, in 1942, and S2, in 1943/1944), which executed a hardwired program that calculated the appropriate profile corrections for the wings of flying bombs (Zuse was in charge of these computations at the Henschel factory in Berlin). Both machines were binary and used fixed-point numbers.


- Relay processor
- Fixed-point
- Mechanical memory
- Electronic, relay computer
- Programmable
- Floating-point
- Microprogrammed with rotary switches
- No conditional branching
1940
1941
Z3
Z2
1939
1942
- Subset of the Z3
- Embedded program
- Used for computing corrections to wings of flying bombs
S1
Zuse's Military Service at the Western Front
Aug. 13: Battle of Britain
Jun. 14: Paris occupied
Jun. 22: Germany attacks the Soviet Union
Atanasoff-Berry-Computer
IBM/Harvard Mark I
Dec. 7: USA enters WWII
Microprogramming notes
Sep. 1: Begin of WWII
Aug.: Hitler-Stalin Pact
1938
Mar. 12: Germany annexes Austria
1943
Z1
Aug.: Japan invades China
Colossus
ENIAC
Sep. 3: Italy surrenders
- Mechanical, binary computer
- Floating-point (+,−,×,÷)
- Programmable
- Microprogrammed
S2
1937
Oct. 25: Berlin-Rome Axis signed
Jun. 6: D-Day
1944
The S2 was similar to the S1, but with automatic measurement of the wing's shape
Oct.: USSR enters Prussia
May 7: Germany surrenders
Z4
Sep. 15: Nuremberg laws
1936
1945
- Superset of the Z3
- Relay computer
- Mechanical memory
- Large instruction set
- No conditional branching
Konrad Zuse resigns from the Henschel company
1935
Logistic Machine & Plankalkül

Chronology of Zuse's computers and important events during World War II

In 1941, after successfully demonstrating the Z3 relay machine computing determinants, Zuse's company received first a loan and then a military contract to supply Henschel with the Z4, a machine that would be faster, have a larger instruction set, and several punched tape readers, one for the main program and the others for program libraries. The Z4 was nearing completion by the end of the war but remained in storage for several years after the surrender. Finally, it was refurbished and leased to the ETH Zurich in 1949–1950. This concludes the early history of Zuse's computers. With this successful transaction, Zuse was able to restart his company, the first computer start-up in Germany. The figure above shows the chronology of Zuse's early computers and how it corresponds to important events during the war. The diagram also shows the overlapping development of three important American computers and Colossus, a British war effort.

The ten years of Zuse's *Sturm und Drang* period (1935–1945) end with two theoretical results: the design of the so-called logistic machine and its profound relationship to the programming language "Plankalkül", the first high-level computer language ever proposed.

Photograph of Konrad Zuse working on the computer Z4 (Image: Konrad Zuse Internet Archive, http://zuse.zib.de/)

In this book, we follow the development of Zuse's ideas, computer by computer, explaining their architectures and capabilities. As we will see, the main weakness of all the machines was the omission of the conditional jump in the instruction set. Even the Z4, the improved and more sophisticated computer, did not have a conditional jump until the ETH required its inclusion as a prerequisite for acquiring the machine. Until then, Zuse's computers could only perform long sequences of forward calculations, or a single loop obtained by attaching the beginning and end of a punched tape. The conditional jump instruction greatly expanded the Z4's usefulness for complex numerical calculations. Zuse stated in later years that he was aware of the need for a conditional jump, and even indirect addressing, but that both required the program to be stored in memory in order to be effective (Zuse, 1972). However, given the small size of the mechanical and electromagnetic storage units in his computers, storing the program in memory was out of the question.

Zuse's competition in the USA was represented by three machines at this time: the Atanasoff-Berry computer, a special-purpose binary device for solving systems of linear equations, the IBM/Harvard-Mark I, a massive relay machine unveiled in 1944, and the ENIAC, the first programmable vacuum tube computer, completed in 1945 (Bruderer, 2020). As you can see from the chronology in the figure, Zuse was the early pacesetter for these developments. The Z1 was completed before the American computers were even designed. However, all three American computers could be shown to work before the Z4 was completed. Zuse's initial advantage dissipated during the war.

The Z4 was the computer that Zuse had dreamed of in 1935. Its realization took ten years of intensive work under difficult wartime conditions. The design of Plankalkül crowns these ten years, the most creative of Zuse's life. Plankalkül was a remarkable achievement because it aimed to establish a comprehensive symbolic calculus for computer programs. Consequently, Zuse devised a notation capable of expressing both predicate calculus formulas and equivalent imperative programs. In this sense, Plankalkül is both a logical specification language and an algorithmic language. Using Plankalkül notation, Zuse created the first symbolic processing programs.

It would be another five years after the war before Zuse was able to lease/sell the Z4 to the ETH, but by then the inventor who had designed all of his computers single-handedly had become an entrepreneur, and new machines were increasingly developed by committee in collaboration with his team of engineers. Zuse's company, which was re-established in 1949, operated independently until 1964, when it was acquired by Brown, Boveri & Co. Siemens bought 70% of the company in 1967 and the rest two years later. When it was liquidated in 1969, Zuse KG had delivered 251 computers in Germany and other European countries during its twenty years of operation.

Paradoxically, as late as 1950, when all new computer prototypes in other countries were electronic, Zuse was still thinking about mechanical binary components as low-cost substitutes for relays or expensive electronic tubes. He was certainly the last great maestro of mechanical computers, in the tradition of Babbage's Analytical Engine, but also one of the first builders of electromagnetic computers in the world. His pioneering achievements lie between two intellectual and technological eras, in the transition from the second to the third industrial revolution.

Nevada, USA Raúl Rojas
August 2023

## References

Bruderer, H. 2020. *Milestones in analog and digital computing*, vol. 1 and 2, 2075 p., 3rd edn. Cham: Springer Nature Switzerland. https://doi.org/10.1007/978-3-030-40974-6.

Zuse, K. 1972. *Der Plankalkül*, vol. 63 . Sankt Augustin: Berichte der Gesellschaft für Mathematik und Datenverarbeitung.

# Acknowledgements

As with any project that has taken so long, there are many people to thank for their help and support over the years. First and foremost, I am deeply grateful to Konrad Zuse, whose generosity in sharing his knowledge and experience was instrumental in the development of this work. He was already the *grand seigneur* of computing in Germany, yet he was always interested in engaging with young researchers. Unfortunately, he passed away before our first simulation of the Z3 for the Internet was completed. It would have given him a great deal of satisfaction to see his machine running in the first Internet browsers.

My student Alexander Thurm wrote the first Java simulation of the Z3 almost 30 years ago. My colleagues Georg Heyne and Frank Darius were the hearts and minds behind the reconstruction of the Z3 with modern relays. Our first Zuse-related project was to build a modern relay version of the addition/subtraction unit of the Z3. Thorsten Vetter, Wolfram Däumel, Lothar Schönbein, and Peter Zielske developed different parts of the electronics for both projects. Cüneyt Göktekin wrote the user interface for the Z3 reconstruction. He and Gerald Friedland as well as Lars Knipping developed important parts of the Plankalkül interpreter. Mike Krüger, Olaf Langmack, and Denis Kunißdeveloped the parser and executable code generator for the interpreter. Ludmila Scharf translated Zuse's chess code into Java for an Internet simulation. Hadwig Dorsch provided me with important documents and blueprints of the reconstructed Z1 built for the German Museum of Technology in Berlin. She and Eva Kudrass also gave us direct access to the machine, a rare privilege in a museum.

I must also thank Horst Zuse, Konrad Zuse's eldest son, who over the years has provided insight and support for all of our Zuse projects at Freie Universität Berlin. He was able to raise the entire funding for our reconstruction of Z3. The Konrad Zuse Society was always interested in my research and organized talks on some of the papers that are now chapters of the book.

For many years, the Konrad Zuse Center in Berlin has generously provided the server for hosting all documents in the Zuse Internet Archive, initiated by me and funded by by the German Research Foundation (DFG), as well as the simulations of Z1 and Z3. Unfortunately, the development of Java made our applets obsolete. I would also like to acknowledge the fruitful collaboration with Dr. Wilhelm Füßl of Deutsches Museum in Munich during the successful digitization of Zuse's *Nachlass*. Special thanks to Susanne Schöttker-Söhl for her transcription work on the documents and her contributions to the digitization process.

I would like to thank friends and colleagues who provided valuable feedback and suggestions during the development of this book: Victor Perez Abreu, Rafael Gamboa, Alexander Förster, Michael Engel, David Hemmendinger, Stefan Höltgen, and Eva Kudrass for their suggestions and corrections. I am especially indebted to Prof. Wolfgang Bibel for his assistance in debugging the manuscript and for helping me to clarify the historical achievements contained in Plankalkül. His guidance and expertise were instrumental in ensuring the accuracy and clarity of the content. I would also like to thank Günter Feuer for his meticulous reading of the various chapters. His careful attention to detail and astute suggestions for deletions and consolidations greatly improved the overall readability of the manuscript. I am also grateful to Sarah Hiller for her work in cleaning up the LaTeX code and organizing the references in the book. Philomena Maher read the manuscript and corrected my English. My thanks to all of them.

I would like to express my deepest gratitude to my wife, Magui, and my daughter, Tania, for their unwavering support and understanding throughout my projects. They patiently stood by me during the long hours of work, especially during the intense periods when I was immersed in deciphering Zuse's intricate circuits. It was a challenging journey, but the ultimate satisfaction came from being able to visualize in my mind's eye the harmonious orchestration of interlocking mechanical parts within the first computer in Germany. This book is dedicated to both of them as a token of my love and gratitude.

# Contents

# Chapter 1
# Konrad Zuse and the Dawn of the Computer Age

*This chapter provides an overview of the basic facts about Konrad Zuse and his early computing machines, i.e., those built in the period 1936–1945. The chapter summarizes the whole book.*

The inventor Konrad Zuse (1910–1995) is a legendary figure in Germany, where he is widely celebrated as the "father of the computer." In 1941, Zuse unveiled the world's first programmable computing machine in his Berlin workshop. The Z3, as the machine became known, was shown to a select group of visitors to demonstrate its capabilities. Young Konrad had begun designing computers long before the war. Against all odds, he persevered during and after the global conflict.

In other countries, however, Konrad Zuse is not so well known. This is not surprising, since most of his early computers were not recognized, either in Germany or abroad, until after 1945. Many American books on the history of computing mention the German inventor only briefly (Campbell-Kelly et al. 2013). Typically, his work receives only a passing mention, even though, as this book shows, modern computers bear more resemblance to Zuse's Z1 or Z3 than to the American ENIAC or the Harvard Mark I, two other machines built during World War II.

## 1.1 Birth and Education

Konrad Ernst Otto Zuse was born on June 22, 1910 (2 years before the birth of Alan Turing) in Wilmersdorf, now a district of Berlin, as the son of Emil and Maria Zuse. His father was a Prussian civil servant in the postal service. Emil Zuse moved the family to Braunsberg (now Braniewo in Poland) when Konrad was 2 years old (see Fig. 1.1). Konrad attended elementary school in Braunsberg and received his basic education at the local Hosianum Gymnasium (where the famous German mathematician Karl Weierstraß had once taught). In 1923, the family moved again,

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_1

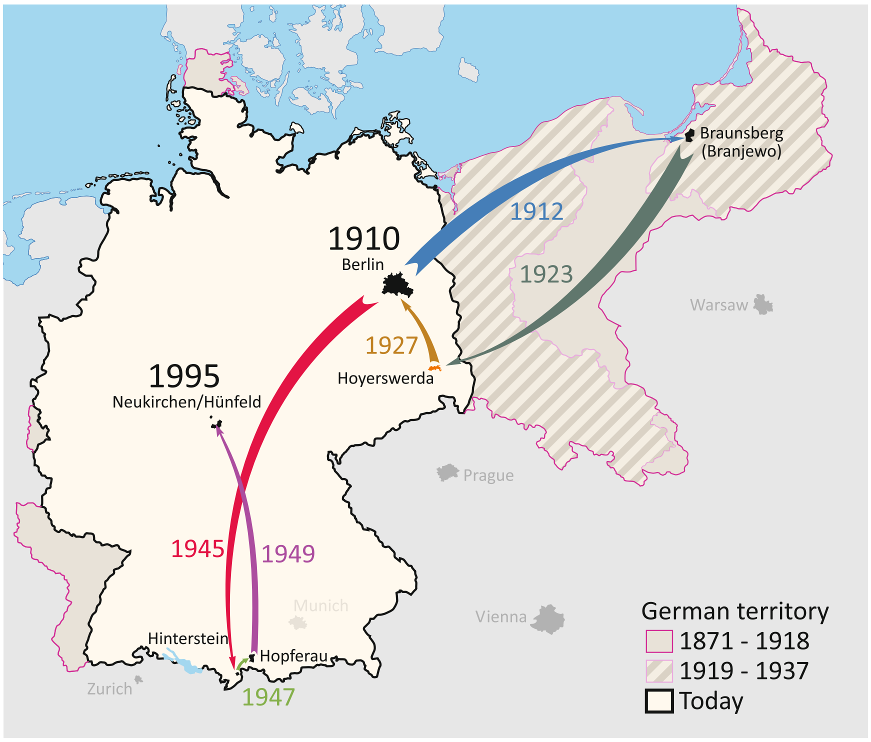


**Fig. 1.1** The map of Germany after 1918 showing the location of the cities where Konrad Zuse lived and the year of migration. The shaded parts are the territories lost by Germany after World War II. The birth date is 1910, and the date of death is 1995

this time to Hoyerswerda (a town near the present-day border with Poland). Zuse was enrolled at the local *Realgymnasium*, an institution that prepared students for admission to several technical universities in Germany. In his autobiography, Zuse describes the new environment, which included a local mining industry, as better suited to his technical aspirations (Zuse 1970). It also helped that the school required fewer hours of Latin, a language Zuse detested.

In 1927, Konrad Zuse received his high school diploma (*Abitur*) and soon after began his studies at the *Technische Hochschule zu Berlin* (renamed *Technische Universität Berlin* after World War II). Zuse mentions in his memoirs that he was 2 years younger than the other students (Zuse 1970). Later in life, he regretted that he had not tried to learn more at school.

When he enrolled at the university, Zuse had not yet made up his mind about his future profession. He first tried mechanical engineering, then switched to architecture, took a year off, tried graphic design for advertising, and finally, chose civil engineering. Zuse wrote that he eventually discovered that this type of profession was ideal for him because it allowed him to combine his artistic interests with his technical skills, especially in mechanical design. It was also a profession that gave the student more creative freedom (Zuse 1972). As a high school student, Zuse was already a technical dreamer and tinkerer, often retreating to work with his "Stabil" mechanical set (a kind of German Meccano), which allowed him to build prototypes of complex machines. As a student, he won several prizes for his Stabil constructions (the last in 1928), which he liked to show off to his friends (Rojas 2001).

## 1.2 First Ideas: The Spreadsheet Computer

While studying civil engineering at the TH Berlin, Zuse learned to perform highly repetitive structural calculations, such as those needed to determine the stress on structures like bridges or cranes. These calculations were typically performed manually or with the aid of desk calculators. Spreadsheets with all the necessary formulas preprinted on them were painstakingly filled out, row by row, column by column. It was a tedious and repetitive task that led Zuse to consider the possibility of automating this work. In these spreadsheets, the engineer simply had to enter data and follow a fixed computational path. Therefore, he thought, a machine could take over (Kurrer 2010).

In his autobiography, Zuse traces his interest in computing machines back to 1934/35. In 1934, he submitted a "Studienarbeit" on the systematic arrangement of computations for structural analysis (Fig. 1.2) (Kurrer 2010). By then, he was thinking full-time about automating computational tasks. His initial concept was to map spreadsheets onto a plane of memory cells. In each cell, it would be possible

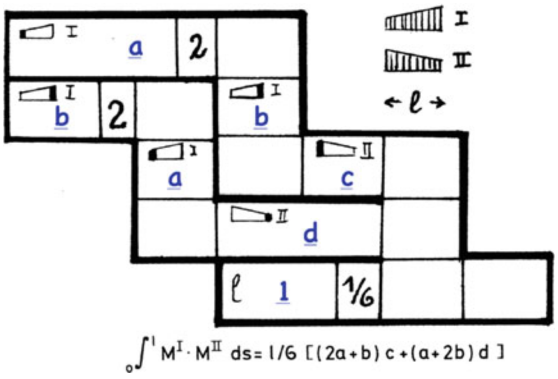

I
a
2
I
I
b
2
b
I
a
II
c
II
d
ℓ
1
1/6
I
II
ℓ
$\int_0^l M^I \cdot M^{II}\, ds = 1/6\,[(2a+b)\,c + (a+2b)\,d]$

**Fig. 1.2** Example of a spreadsheet for a static forces calculation (Zuse 1970). Numbers in cells are multiplied horizontally and added vertically, until the lowest right cell has been filled. The initial constants (in this case $a, b, c, d$) are written in the cells containing a symbol. That initial data trigger the subsequent computations. The final result, corresponding to the formula at the bottom, is written in the lowest cell to the right

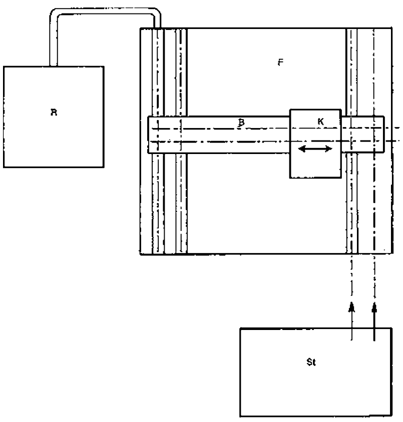


**Fig. 1.3** A read/write head able to displace across a plane for reading and writing numbers from the spreadsheet cells (using vertical pins) (Zuse 1970). The box St manages the control, the box R represents the calculating portion of the device

to store a number using vertical pins representing zeros or ones according to their height. A mechanical device would traverse the entire plane (using something like the mechanism of a modern $xy$ plotter) and would be able to read numbers from each cell (encoded using vertical pins) for the computations needed in the spreadsheet (Fig. 1.3). It could then store the result in a new cell. The device would be something like a pocket calculator, but with a mobile read/write head going from cell to cell in the planar spreadsheet (Zuse 1970).

The "spreadsheet computer" envisioned by Zuse already contained some interesting ideas. One was to use the binary system to represent positive numbers, the other to represent negative numbers using the complement representation, so that subtraction could be treated as addition with a complementary number. Zuse quickly realized that the memory cells did not have to be arranged in any particular order, as they were in the spreadsheets. If the cells were numbered (i.e., addressable), they could be retrieved by their address. So he went on to design a computer with a processor and addressable memory based on binary numbers and their complement (the two's complement representation). Also during these years, he developed what he called the "semi-logarithmic" (i.e., floating-point) representation that he would use in all of his early computers. He also wrote a complete description of the binary system and the algorithms he intended to use (Zuse 1937).

We show in this book that Konrad Zuse pursued almost the same basic computer architecture during the period 1936–1945, through different incarnations of the basic ideas. His successive machines were called V1, V2, V3, and V4, with the capital V standing for *Versuchsmodell* (experimental model). The V was changed to Z after the war to avoid any association with the V2 rockets. Between the Z3 and Z4, Zuse built two specialized machines for the German military, the S1 and S2. The S stands for "*Sondermaschine*" (special machine). The S1 and S2 were based on a small subset of the Z3 architecture, but worked with fixed-point numbers (integers).

## 1.3 The Z1 and Z3 Machines

In July 1935, right after graduation, Zuse began working as a stress analyst for the aircraft manufacturer Henschel-Flugzeug-Werke. The German aircraft industry was expanding at a furious pace, in violation of the Versailles Peace Treaty. Only 2 years earlier, Adolf Hitler had been elected chancellor, assuming dictatorial powers, and the country was on the brink of war. Zuse's work at Henschel consisted of supervising the structural calculations needed to correct the wings of aircraft with a full metal fuselage.

Zuse remained in his position at Henschel for less than a year before resigning on May 31, 1936, in order to found his own company, which would be based on his design for a computing device. In early 1936, shortly before his departure, he wrote a lengthy memo entitled "Computing Machine for the Engineer" (Zuse 1936d) in which he detailed his vision of an automatic device comprising storage and processing components capable of performing extended sequences of basic arithmetic operations. Paradoxically, Zuse's brief tenure at Henschel would prove crucial for him in the years to come. Twice in his life, his superiors at the armament company helped him secure a discharge from the army, arguing both times that he was needed as an engineer, not on the battlefield.

In mid-1936, with the financial support of his parents, Zuse began building the automaton that had existed only in his notebooks. Some friends at the university helped by working for him, while others offered small monetary contributions so that he could finish what would become the Z1 machine. In 1937, he showed his machine to Kurt Pannke, a designer of special calculators (Zuse 1970), who was impressed enough to contribute 7000 Reichsmark for further development of the machine (at that time, a house on the outskirts of the city could be bought for 30,000 Reichsmark).

Pannke's financial support notwithstanding, one aspect that this book makes clear is that the most important difference between Zuse and other computer inventors working in the late 1930s was the fact that he was essentially building his machines alone, whereas in the United States, scientists like John Mauchly (Burks and Burks 1988) and Howard Aiken (Aiken and Hopper 1982) had the resources of universities, the military, or major corporations at their disposal. The logical and mechanical conception of the Z1 was entirely Zuse's brainchild.

Zuse, essentially unaware of the internal structure of calculators built at the time, started from scratch and developed an entirely new type of mechanical assembly. While existing desktop calculators were based on the decimal system and used rotating mechanical components, Zuse decided to use the binary system and metal plates that could move linearly back and forth. That is, the plates could only slide from position 0 to position 1 and vice versa. Zuse's basic mechanical component was a switch that could be "opened" or "closed" like an electromagnetic relay.

Such simple mechanical elements were all that was needed for a binary machine, but important obstacles had to be overcome. It was necessary to specify the complete logical description of the machine and then "wire" it accordingly. The mechanical

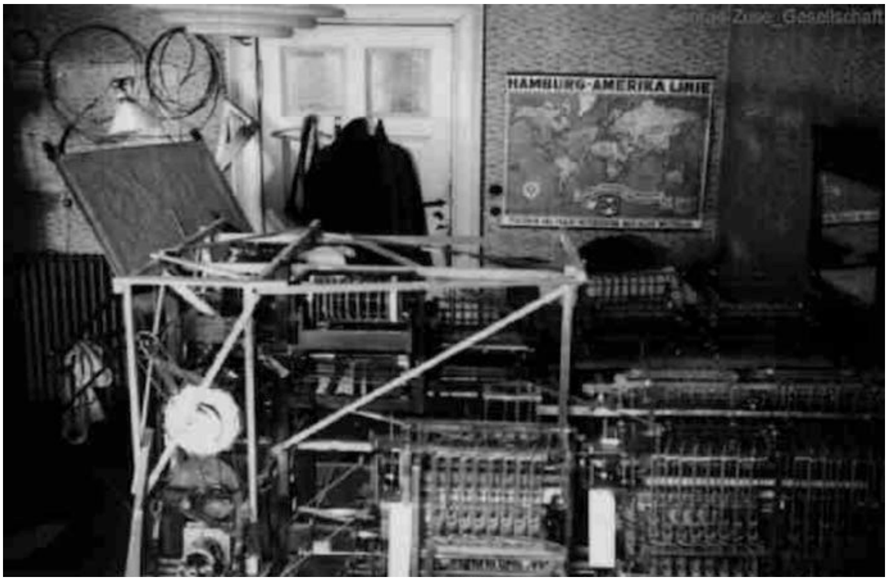


**Fig. 1.4** One of the few existing pictures of the mechanical Z1 built in Zuse's living room (Image: Deutsches Museum)

components, however, posed a formidable challenge, since any movement of one logic gate had to be mechanically coupled to the movement of the other gates. Horizontal displacements of the components had to be transformed into sliding displacements over different planar layers of the machine or even into vertical movements. From today's perspective, the 3D mechanical design of the machine was much more complicated than conceiving its purely logical structure. It is fair to say that none of Zuse's friends understood exactly how the machine worked, even though they spent weeks at his home making the thousands of metal parts needed for the apparatus. One of his assistants wrote: "I am honest enough to say that I worked blind, and that I did not know how the monster that was being built there would one day work" (Zuse 1970).

The Z1 was operational in 1938 (Fig. 1.4). It was shown to several people who saw it rattle and clatter as it computed the determinant of a $3 \times 3$ matrix. However, the machine was never reliable enough. The mechanical components, all cut by hand from metal plates, tended to jam. Zuse later called the Z1 a "dead end." Nevertheless, the mechanical Z1 proved that the logical design was sound. An electromagnetic realization, using telephone relays, could be considered as the next step. Helmut Schreyer, an electronic engineer and college friend of Zuse, suggested the use of vacuum tubes when he saw the machine. In fact, Schreyer chose this as his dissertation project and developed some vacuum tube circuits for an electronic device. Zuse, however, doubted that vacuum tube machines could be as cheap and reliable in the long run as telephone relays or even mechanical components (Zuse

1943). Zuse's goal was to develop a robust, programmable replacement for existing mechanical calculators that could be used in large or medium-sized companies. This was to be a computing machine for the engineer, eventually small enough to be placed on a desk.

In 1938, Schreyer and Zuse showed some electronic circuits to a small group at the university. When asked how many vacuum tubes would be needed for a calculating machine, they replied that 2000 tubes and several thousand other components would be needed. The academic audience groaned in disbelief—the most complex vacuum circuits of the time contained no more than a few hundred tubes, and the electrical power required to run such a machine would be prohibitive. However, just 7 years later, ENIAC, built at the Moore School of Engineering in Philadelphia, would show the world that vacuum tube machines, while expensive, were entirely feasible (Burks and Burks 1981).

The impending invasion of Poland in 1939 had immediate consequences for Zuse: he was drafted into the army on August 26. With the help of Kurt Pannke, he tried to get a transfer to Berlin to continue his work on the next computing machine. Helmut Schreyer, who worked as an engineer at the university, also tried to get Zuse discharged by offering to build an automatic air defense system that could be operational in 2 years. His offer was met with the sardonic reply that the war would be over by then. Eventually, Zuse's acquaintances at Henschel were able to secure his discharge and transfer to the Henschel aircraft factory in Berlin-Adlershof, where he was rehired to supervise static force calculations. Later, he automated the calculations necessary to correct the wings of the "flying bombs" (radio-controlled missiles) being built at the factory.

In March 1940, Zuse began working for "Special Section F" at the Henschel factory (the flying bombs unit, headed by Prof. Herbert Wagner). Two by-products of his work there were the calculating machines S1 and S2 (first named HS-1 and HS-2, where HS refers to *Henschel-Sondermaschine*). The S2 could automatically sense and measure the profile of rocket wings, convert the analog measurements into digital numbers, and compute corrections based on those values. The previous model, the S1, required manual input of these numbers using a decimal keyboard. The S1 and S2 were probably the first digital computers used for factory process control. The instrumentation used in the S2 was one of the first industrial analog-to-digital converters, although it was never used in real production. From a computational point of view, both machines were a subset of the machines described below. Their existence remained unknown to the general public for years after the war (Fig. 1.5).

In 1940, moving away from the mechanical design of the Z1, Zuse assembled the Z2 machine, an experimental prototype that used a relay-based integer processor and a mechanical memory cannibalized from the Z1. This small machine helped Zuse convince the *Deutsche Versuchsanstalt für Luftfahrt* (DVL) to partially fund the development of the Z1's successor, the Z3, which would be built using only relays.

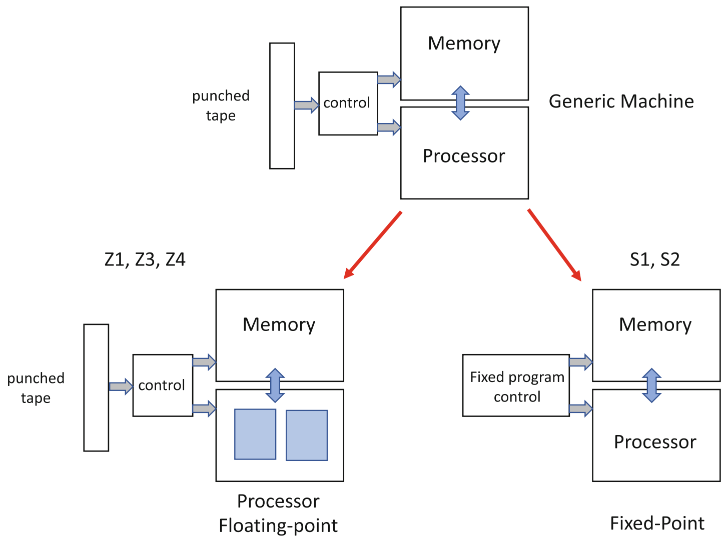


**Fig. 1.5** The abstract architecture of all Zuse's machines (on top) and its concretization in several machines that he built from 1936 to 1945. The Z1, Z3, and Z4 worked with floating-point numbers, the S1 and S2 with fixed-point numbers. The relay machine Z2 is not shown in the figure. It had the abstract structure of the Z1, but used fixed-point numbers. It was built as a proof of concept for the use of relays as binary elements

## 1.4 Construction and Capabilities of the Z1, Z3, and Z4

In the Z1 and Z3, the input and the results were floating-point numbers (i.e., numbers such as $+12.654 \times 10^6$, with an integer and a fractional part multiplying a decimal power). Zuse developed a binary representation for floating point very similar to the internal number format used in modern computers. Each number was stored in three parts: the sign of the number, the exponent of the number in two's complement notation, and the mantissa (also called the significand) of the number. To handle each part, the processor of the Z1 and Z3 consisted of two main blocks, one for processing the exponents of numbers and one for processing the mantissas (Rojas 1997). Zuse called his approach "semi-logarithmic" notation, since in the floating-point representation the exponent of base 2 represents the integer part of the logarithm (base 2) of the stored number. Zuse dated this idea to 1934 (Zuse 1972).

The two machines, Z1 and Z3, shared a common architecture. Their main components (Fig. 1.6) were:

1. The memory for storing numbers (16 in the Z1, 64 in the Z3).
2. The processor for performing arithmetic operations.
3. A punched tape for storing the sequence of program instructions.
4. A decimal input/output console.

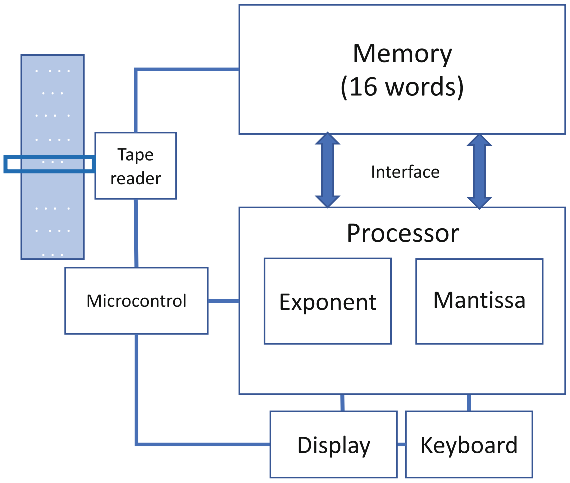


**Fig. 1.6** Separation between processor and memory in the Z1

Instructions were read from the tape to be executed one by one by the processor. The console allowed the user to enter decimal numbers with a decimal keyboard (similar to the keyboard of a cash register), while the results were displayed in a panel with decimal digits that were selected to be uncovered mechanically in the Z1 or highlighted with lamps in the Z3.

The instruction set of the Z1 and Z3 included the four arithmetic operations (addition, subtraction, multiplication, and division). The Z3 also included the square root operation. There were two additional operations for reading and displaying decimal results (convert from decimal to binary and vice versa) and two for transferring numbers between the processor and memory (load and store). The Z3 was very much like an early electronic calculator of the 1970s, but much slower: a multiplication had 18 machine cycles and did the calculation in 3 seconds. Division and square root operations were performed in about the same time.

With the instruction set mentioned above, it was possible to compute any arithmetic formula of the kind used in the engineering applications Zuse had in mind. However, the instruction set did not provide a conditional branch instruction, so it was relatively difficult, though not impossible, to perform conditional computations. Also, the two ends of the punched tape could be joined to form a loop, so that repeated execution of the same program was possible.

I have always thought that Zuse's vision of a spreadsheet computer, where all computations flow deterministically from start to finish, made him overlook the crucial importance of the conditional branch, even for small programs. It is possible that the static computations that Zuse had to supervise at Henschel were almost always embedded in spreadsheets. In his autobiography, Zuse explains the lack of the conditional branch as a way of keeping the complexity of the machine's computations down, but his explanation is not convincing (Zuse 1970). Zuse offers another explanation in his 1945 design for a high-level language, the Plankalkül. He writes: “I deliberately did not include the conditional branch and the computation of addresses in the machines to be developed, because this would have delayed their delivery. Also, the necessary technical conditions were not available during those war years (for example, the construction of storage units with sufficient capacity to store the computer program)” (Zuse 1972).

Zuse kept the number of logic gates for the processor low by relying on controllers that acted as microsequencers, one for each instruction in the instruction set. A microsequencer in the Z3 consisted of a rotating arm that advanced one step in each cycle of the machine, like a rotary dial. A motor provided the clock cycles needed to synchronize all parts of the machine. In the case of the Z3, the operating frequency was set at five cycles per second. That is, five times per second, the rotating arm in a microsequencer activated the next microstep of the current operation. For example, in the case of multiplication, repeated addition and shifting of numbers were required (as happens when you multiply two numbers by hand). The required 18 suboperations were all started by a microsequencer with 18 contacts for the rotary switch. The microsequencer can be thought of as a kind of hardwired program that reduces very complex instructions to a sequence of simple operations. Therefore, the entire internal operation of the machine could be changed by rewiring the microsequencers without having to change the rest of the processor. This resulted in a very efficient and flexible architecture and explains how Konrad Zuse was able to build a machine that rivaled the British or American computers of the same period, despite having far fewer resources at his disposal.

During World War II, Zuse worked for the Henschel factory, but was finally able to start his own business in April 1941. The *Zuse Ingenieurbüro und Apparatebau, Berlin* was the first company in the world founded with the sole purpose of developing computers. The successful demonstration of the Z3 brought Zuse a contract with the DVL and then with the Ministry of Aviation to develop an even larger computer, the Z4. This machine had a very similar design to the Z3, but with a larger instruction set. The machine was built (with an initial memory of 12 words, expandable to 64) and was almost operational by February 1945. By this time, Zuse’s company already had 20 employees (Table 1.1).

## 1.5 The Aftermath of the War and Plankalkül

In early 1945, Zuse fled with the Z4 before Berlin fell to the Soviet Army. One of his collaborators was able to get the machine shipped by train and somehow managed to misrepresent it as being of strategic military value. The Z1 and Z3 had already been destroyed in air raids 2 years earlier, leaving the Z4 as the only asset of Zuse’s company. After several detours, Zuse and his team settled in Bavaria, where he survived the following years by painting postcards, consulting, and attempting to restart his company. During this period of forced inactivity, he completed his manuscript on the Plankalkül, a remarkable document first published in 1972.

**Table 1.1** Overview of the Zuse machines built from 1936 until 1945 (Zuse 1946). The abbreviation HFW refers to Henschel-Flugzeug-Werke, and RLM to the Reich's Aviation Ministry

| Name | Years | Ordered by | Type | Technology | Status | Status |
|---|---|---|---|---|---|---|
| V1 | 1936–1938 | | Algebraic | Mechanical | Experimental | Destroyed |
| V2 | 1939–1940 | | Algebraic | Relays, mechanical memory | Experimental | Destroyed |
| V3 | 1939–1942 | DVL | Algebraic | Relays, 64 memory cells | Experimental | Destroyed |
| V4 | 1942–1945 | RLM | Algebraic | Relays, mechanical memory | Commercial | Preserved |
| S1 | 1942 | HFW | Special-purpose | Relays | Used | Disassembled |
| S2a | 1943–1944 | HFW | Special-purpose | Relays | Unused | Disassembled |
| S2b | | HFW | Automatic sensing | Relays | Unused | Disassembled |
| L3 | 1944 | | Logic | Relays | Theoretical | |
| R1 | 1940–1945 | | RLM | Vacuum tubes | Experimental | Destroyed |

The Plankalkül (calculus of programs) was the world's first high-level programming language (Zuse 1972). It was designed by Zuse between 1939 and 1945, at a time when the first computers were being built in the USA, United Kingdom, and Germany. It represents one of the most important contributions to the history of ideas in the field of computing, although it was first implemented in 1999 by our research team in Berlin.

The Plankalkül corresponds to Zuse's mature conception of how to build a computer and how to allocate the total computing work to the hardware and software of a machine. Zuse called the first computers he built "algebraic" in contrast to the "logistic machines" (Zuse 1943). The former were built specifically to handle scientific computations, while the latter could handle both scientific and symbolic processing problems. In modern computers, there are two ALUs, one for integers and one or more for floating-point operations. It would not have been impossible to do logic operations in the Z3 or Z4, but it is cumbersome to do so with floating-point registers. For Zuse, the distinction between algebraic and logical computation emphasized the need for two types of machines. In fact, in 1950, he applied for a patent for a "combined machine" with a floating-point processor and a logic processor running in parallel and with a common memory unit (Zuse 1950).

Around 1944, Zuse greatly simplified the hardware needed to perform logic operations. What he came to call the "logic machine" was never fully developed (although a small prototype was built), but his design called for a one-bit word memory and a processor that could compute only basic logic operations (conjunction, disjunction, and negation). It was a sort of minimalist computer in which the memory consisted of a long chain of bits that could be grouped in any way to represent numbers, characters, arrays, and so on. In some ways, the logistic machine resembles Alan Turing's 1936 proposal for what we now call a Turing machine.

The Plankalkül was the software counterpart of the logistic machine. Complex structures could be built as arrays of elementary ones, the simplest being a single bit. Today we would call Zuse's arrays multidimensional "tensors," but the addressing was more in the form of a tree. A matrix, for example, would be a tree with rows hanging from the root. An element in the matrix could be addressed by first specifying the row and then the element in the row.

Also, sequences of instructions could be grouped into subroutines and functions, so that the user was dealing only with a powerful high-level instruction set that masked the complexity of the underlying hardware. The Plankalkül strongly exploited the concept of modularity, so important in computer science today: multiple layers of software made the hardware invisible to the programmer. The hardware itself only had to execute a minimal set of instructions.

In Plankalkül, the programmer uses variables to store the results of computations. There are no separate variable declarations: any variable can be used in any part of the program, and its type is written along with its name. Variable assignment is done as in modern imperative programming languages, where a new value overwrites the old one. Many operations are those used in modern programming languages (addition, subtraction, etc.). Plankalkül is universal. It can handle conditional instructions of the type "IF-THEN-ELSE" and provides an iteration operator W, which repeats the execution of a sequence of instructions until a terminal condition is met. Using these constructs, any kind of computation can be expressed with Plankalkül.

When Zuse started to develop the Plankalkül, he wanted to develop a complete "calculus of programs." In the case of algebraic expressions, he wrote, we can equate a quadratic polynomial to zero and, by applying algebraic rules, derive an explicit formula for its roots. He aspired to do the same for the predicate calculus. However, this would have involved developing automatic proof methods for general formulas of the predicate calculus, a very ambitious undertaking. In the 1972 edition of Plankalkül, Zuse wrote that solving general logic formulas "is not so easy." Consequently, this problem was not further explored in the manuscript. Nevertheless, Zuse emphasized that Plankalkül served both as a notation for describing logical concepts and as an algorithmic language capable of facilitating various computations (Zuse 1972).

Although Zuse published some small papers about the Plankalkül and tried to make it known in Germany, it never generated enough interest. The main problems were its ambitious scope, the large number of possible commands, its modular architecture requiring incremental compilation, and the presence of dynamic structures, set-theoretic operations, and functionals. Some aspects of the definition were ambiguous, and the lack of type checking would have made debugging extremely difficult. A practical implementation of Plankalkül certainly requires a major revision of Zuse's 1945 draft. However, Plankalkül was very much ahead of its time, considering that many of the concepts on which it was based would be rediscovered much later (including logic and functional programming). Decades would pass before programming languages reached the level of sophistication of Plankalkül.

## 1.6 Rebirth of Zuse's Company

After the war, in 1947, Zuse restarted his company as *Zuse Ingenieurbüro*. It had a new lease of life when Prof. Eduard Stiefel of the *Eidgenossische Technische Hochschule* (ETH) in Zurich went to Bavaria to see the refurbished Z4 in operation. He decided to lease/buy the machine for his university. With this funding and some partners, Zuse restarted his company for the second time in 1949 as *Zuse Kommanditgesellschaft*.

The Z4 was installed in Zurich in 1950, several months before the first UNIVAC was delivered in the United States (Stern 1981; Bruderer 2012), and was thus the first commercial computer in the world. For several years, the Z4 was also the only commercial computer installed in continental Europe. The machine had the same logical structure as the Z3, but contained more memory and an expanded instruction set. It was used at the ETH for 5 years and is now part of the historical collection of Deutsches Museum in Munich. It is the only Zuse machine built before 1945 that has survived.

Zuse's company (with the new name "Zuse KG") flourished after the war, and many more machines were built. They were all numbered progressively according to their introduction, i.e., Z5, Z11, and so on. For some years, Zuse continued to build relay computers and even advocated the use of micromechanical elements. Gradually, however, electronic components were miniaturized, their reliability increased, and with the dominance of American companies in this field, Zuse KG had no choice but to develop vacuum tube and transistor-based machines. Zuse KG's first transistorized computer was the Z23, a commercial success: 80 machines were shipped to customers in Germany and 18 to other countries. The *Deutsche Forschungsgemeinschaft* actively promoted the machine and subsidized universities that bought it. The Z23 was the computer used to start many computer science programs in Germany.

The Z23 and the Z22 (the latter built with vacuum tubes) were remarkable in that they represented the first radical departure from the architecture of all previous Zuse machines. Their internal structure consisted of serial registers, which required fewer components. The number of instructions was kept to a minimum. A compiler allowed programmers to write code with a syntax somewhere between assembly code and a high-level programming language. Machines such as the Z22 and Z23 were largely designed by Zuse's engineers.

Another important development, and Zuse's final encore, was the introduction in 1961 of the Graphomat, a plotter that could be used by architects and geologists to plot diagrams and drawings. The Graphomat could be connected to Zuse's computers and used special gears to provide smooth, continuous motion in any direction. The gears were designed by Zuse himself.

The Z23 and the Graphomat were successful, but development of the next line of computers proved too costly. Eventually, the dominance of the American computer industry in Europe, as well as the late adoption of an all-electronic design, brought Zuse KG into financial difficulties. The company accepted an industrial investor, but was later sold to Brown Boveri and Co. in 1964. Seventy percent was then sold to Siemens in 1967 and the remainder in 1969. Production of the Zuse series of computers was discontinued. Zuse retired after the Siemens takeover and received a pension. In the years that followed, he continued to write, apply for patents, and argue for his place in the history of computing.

In retrospect, it can be said that Konrad Zuse's greatest achievement was the development of a family of fully digital, floating-point, programmable machines built in almost total intellectual isolation from 1936 to 1945. His dream was to create a small computer for business and scientific applications. He worked tirelessly for many years to achieve this goal. Unfortunately, his 1941 patent application (Zuse 1941) for the Z3 computing machine was rejected by a German judge in 1967 for lack of "inventiveness." The decision on the application was delayed so long because of the war and because the major computer companies fought in court against Zuse, who always considered himself the one and only inventor of the computer. His public statements on the subject sometimes revealed some bitterness about his lack of recognition in other countries.

## 1.7 Epilogue

Konrad Zuse married Gisela Brandes in 1945, before Berlin was under siege. Gisela gave birth to their first son a few months later, and four more children followed in the ensuing years. However, Konrad Zuse was not a family man—over the years his obsession was to build new and better machines. After his retirement, he received many honors in Germany, including the Federal Cross of Merit and the Siemens Ring. In 1999, he was named a Fellow of the Computer Museum in California. He received numerous honorary doctorates and an honorary professorship. In addition, the most important German prize in the field of computer science bears his name.

His early machines have been reconstructed: a model of the Z1 was built by Zuse himself in the 1980s and is on display at the German Museum of Technology in Berlin (Schweier and Saupe 1988). The Z3 was reconstructed by Zuse's engineers in 1960 and has been part of the historical collection of Deutsches Museum in Munich since 1969. A new functional replica of the Z3, with smaller relays, was built by us in Berlin in 2001 and is on display in the Zuse Museum in Hünfeld, Germany, which also houses several computers of the Zuse KG. Konrad Zuse's notebooks and documents were sold to Deutsches Museum in 2006, where they are now stored in the library's stacks and digital archives.

It has often been said that the computer was a by-product of World War II, or at least that its birth was catalyzed by the events surrounding the conflagration. In the case of Konrad Zuse, this is only partially true. The inspiration for his first computing machine, the Z1, predates the war. The 6 months Zuse spent at the front in 1939–1940 were certainly an interruption in the project he had been working on for almost 3 years. Without the war, the Z3 would have been finished sooner. But once the war broke out, Zuse was able to convince the military establishment that calculating machines were useful for aerodynamic calculations. The successful demonstration of the Z2 prototype led to a contract with the DVL, which financed most of the construction of the Z3. Once the Z3 was operational, Zuse developed the special-purpose machines S1 and S2 and began building the more powerful computing machine he had been dreaming of all these years, the Z4.

Although at the time almost no one in Germany fully understood the significance of Zuse's work, at least those responsible for the strategic management of aeronautical research and development recognized the importance of fast computing. It is noteworthy that Zuse was able to leave the front twice, and that he was partially relieved of his day-to-day responsibilities at Henschel-Flugzeug-Werke to take care of his own company. This would not have happened if the military experts had not thought that his commercial work was useful and necessary for the war effort.

Konrad Zuse was certainly no resistance hero, but it is true that he never sought political office or a position in the academic or industrial establishment. While professors and researchers at German universities, especially at the TH Berlin, flocked to the Nazi Party to further their careers, Zuse's own vocation was cut short by the war. Unfortunately, not much is known about his political views at the time. In his memoirs, Zuse devotes only a few paragraphs to the regime and politics during the war. Ideologically, he was very impressed by Oswald Spengler's theory of the decline of Western civilization. He continued to mention Spengler even in his later years.

It was probably Konrad Zuse's personal misfortune that he conceived all the elements of the computer earlier and more elegantly than the other computer pioneers of his time (except Turing), but that he lived in Germany when the country was on its way to a war of aggression and eventual self-destruction. Outside the country, and outside a very small circle in Berlin, no one took notice of the Z1, Z2, Z3, and Z4. The S1 and S2 were secret machines. Zuse's work was not rediscovered until the late 1940s, but it was too late for his machines to have a major impact on the design and construction of modern electronic computers. Zuse's work became, at best, a footnote in early scholarly books on the history of computing. This has changed in recent decades as more has become known about the life and work of this most remarkable computer pioneer (Bruderer 2020).

## References

Aiken, H.H., and G.M. Hopper. 1982. The Automatic Sequence Controlled Calculator. In *The Origins of Digital Computers, Monographs in Computer Science*, ed. B. Randell, 203–222. Berlin: Springer. https://doi.org/10.1109/EE.1946.6434251.

Bruderer, H. 2012. *Konrad Zuse und die Schweiz: Wer hat den Computer erfunden?* Munich: Oldenbourg Wissenschaftsverlag. https://doi.org/10.1524/9783486716658.

Bruderer, H. 2020. *Milestones in Analog and Digital Computing*. Vols. 1 and 2, 3rd ed., 2075. Cham: Springer Nature Switzerland. https://doi.org/10.1007/978-3-030-40974-6.

Burks, A.W., and A.R. Burks. 1981. The ENIAC: First General-Purpose Electronic Computer. *Annals of the History of Computing* 3 (4): 310–399, https://doi.org/10.1109/MAHC.1981.10043.

Burks, A.R., and A.W. Burks. 1988. *The First Electronic Computer – The Atanasoff Story*. Ann Arbor: The University of Michigan Press.

Campbell-Kelly, M., W. Aspray, N. Ensmenger, and J.R. Yost. 2013. *Computer: A History of the Information Machine*. 3rd ed. The Sloan Technology Series. https://doi.org/10.4324/9780429495373.

Kurrer, K.E. 2010. Konrad Zuse und die Baustatik – Zur Vorgeschichte der Computerstatik (Teil I). *Bautechnik* 87 (11). https://doi.org/10.1002/bate.201010046

Rojas, R. 1997. Konrad Zuse's Legacy: The Architecture of the Z1 and Z3. *IEEE Annals of the History of Computing* 19 (2): 5–16. https://doi.org/10.1109/85.586067.

Rojas, R. 2001. Konrad Zuse – War der Erfinder des Computers doch kein Musterschüler? Telepolis.de

Schweier, U., and D. Saupe. 1988. Funktions- und Konstruktionsprinzipien der Programmgesteuerten Rechenmaschine "Z1". Arbeitspapiere der Gesellschaft für Mathematik und Datenverarbeitung 321.

Stern, N. 1981. *From ENIAC to UNIVAC*. Bedford: Digital Press.

Zuse, K. 1936. Die Rechenmaschine des Ingenieurs. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0209/.

Zuse, K. 1936d. Patentanmeldung Z 23 139 IX / 42m: Verfahren zur selbsttätigen Durchführung von Rechnungen mit Hilfe von Rechenmaschinen [für Zuse]. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0990/.

Zuse, K. 1937. Einführung in die allgemeine Dyadik. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0210/.

Zuse, K. 1941. Patentanmeldung Z-391. German Patent Office, Berlin.

Zuse, K. 1943. Rechenplangesteuerte Rechengeräte für technische und wissenschaftliche Rechnungen. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0217/.

Zuse, K. 1946. Zur Entwicklung von Rechengeräten bis zum Jahre 1945. Nennung von Namen und finanziellen Unterstützungen. Available online at the Zuse Internet Archive.

Zuse, K. 1950. Patentschrift Nr.926449, Kombinierte numerische und nichtnumerische Rechenmaschine. Deutsches Patentamt, 11 Seiten.

Zuse, K. 1970. *Der Computer – Mein Lebenswerk*. Landsberg: Verlag Moderne Industrie,

Zuse, K. 1972. Der Plankalkül. 63, Berichte der Gesellschaft für Mathematik und Datenverarbeitung, Sankt Augustin.

# Chapter 2
# The Race to Build the Computer in World War II

*This chapter traces the invention of the computer in several countries during the interwar period and up to 1945, concentrating on events in Germany and comparing Zuse's machines with those invented in the USA. In the second part, we look at Helmut Schreyer's electronic prototype and some aspects of his life and work during the Second World War.*

## 2.1 Berlin Between the Wars

The armistice between the Entente powers and Germany was signed on November 11, 1918, effectively ending the First World War with the defeat of Germany. The peace document, the Treaty of Versailles, was signed a year later on June 28. But after 4 years of war, Germany could not be stabilized. Peace in Europe did not mean peace within Germany itself. It would only be 20 years before most of Europe was engulfed in the continuation of the war.

Two days before the signing of the Armistice in 1918, the German Kaiser was forced to resign, and Germany was transformed into a republic. Under the Treaty of Versailles, Germany had to give up its overseas colonies, lost 13% of its territory in Europe, could not have an air force, and had to pay onerous reparations to the Allies. The 132 billion gold marks in payments was three times Germany's pre-war annual gross domestic product. The effect on the country was devastating, as the famous economist John Maynard Keynes had predicted. In his book "The Economic Consequences of the Peace," he argued that reparations would ruin Germany and create economic instability in Europe. And so it did, setting the stage for the rise of the Nazi Party.

The Weimar Republic was the democratic government established after the end of the German Empire (the new constitution was adopted in the city of Weimar). Economically, this period was marked by the global recession of the 1930s and

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_2

hyperinflation in Germany, especially in the years 1921–1924. The unrest in the country and the impossibility of paying reparations led to the signing of the Dawes Plan, which granted international loans to Germany and modified the reparations system, linking it to a percentage of German exports.

Politically, the Weimar Republic was a period of social unrest. Between 1919 and 1933, before Adolf Hitler came to power, there were 14 different German chancellors. Some of them lasted less than a year, as the alignment of the parties represented in the Reichstag (the parliament) was constantly changing. The Social Democratic Party was the largest, but it was surrounded by many smaller nationalist and radical groups.

Meanwhile, the USA was struggling with the Great Depression of the 1930s. The worst years were 1929–1933, but the depression lasted until 1939. We recognize these dates immediately: Adolf Hitler became chancellor of Germany in 1933, and the Second World War started in 1939.

### *2.1.1 Science and Art in Berlin*

It is paradoxical that while Germany was engulfed in social chaos and unrest, the arts and sciences flourished. The end of the Empire opened the door to democracy and experimentation in all fields, so much so that this period has been called the "Weimar Renaissance." Writers such as Thomas Mann, Bertolt Brecht, and Franz Kafka were very active, while young composers and painters flocked to the metropolis. Berlin was the cultural center of the country, the city of freedom and openness. The "*Großstadt*" was an ambivalent mix of cultural vibrancy, poverty, and political violence.

Berlin was also the capital of science in Germany during the inter-war period. Between 1919 and 1932 alone, 16 Nobel Prizes were awarded to German scientists and artists—14 of them with connections to Berlin (Meyer 2000). Among them were Albert Einstein, who received the prize in 1921, Fritz Haber (1919) and Gustav Hertz (1922) (Fig. 2.1).

It was in this context that two people who would later play a significant role in the history of computing arrived in the city. The first was the Hungarian János (John) von Neumann, who lived in Berlin between 1926 and 1930. He was a multifaceted scientist who had studied chemistry, physics, and the foundations of mathematics at the ETH in Zurich and also in Berlin. He received his PhD in mathematics from the University of Budapest in 1926 at the age of 23. In Berlin, he was not a full professor but a *Privatdozent* and moved to Princeton when he was invited to work at the Institute for Advanced Studies. It was there that he met Alan Turing a few years later.

The second person was, of course, Konrad Zuse. It is not hard to imagine the young Zuse crossing paths with von Neumann in the street. Neither of them would have guessed how closely their later work would be intertwined.

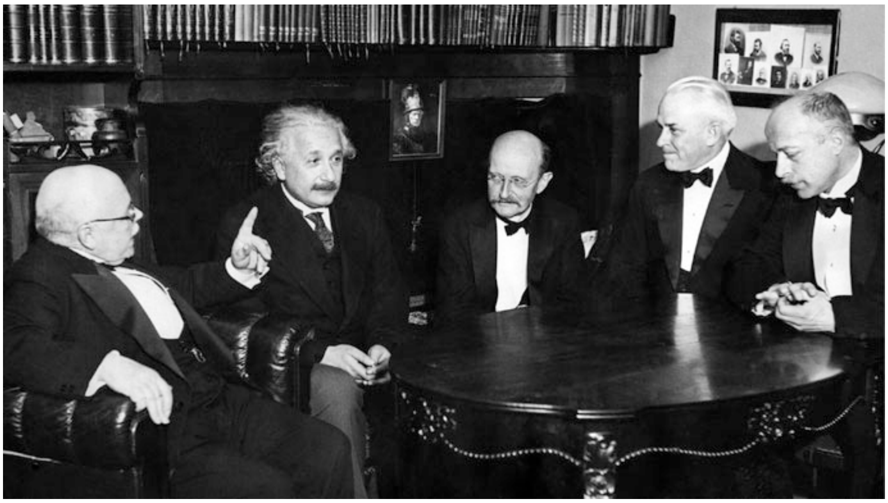


**Fig. 2.1** A meeting between the Nobel Prize winners Walther Nernst, Albert Einstein, Max Planck, Robert Andrews Millikan, and Max von Laue (Image: Wikimedia Commons)

### *2.1.2 The Years of Computability*

The computer first appeared as a theoretical object. The Princeton mathematician Alonzo Church published a paper in 1932 entitled "A Set of Postulates for the Foundation of Logic" in which he proposed the *lambda calculus*, a logical approach to mathematics and computability. What Church achieved with this and other papers was to show that all mathematical objects can be reduced to functions, and that mathematical computations can be expressed as functions acting on functions. The natural numbers, for example, are themselves functions. The number one can be understood as the function that applies a function once to another function. The number two applies the function twice, and so on. Church could define functions for addition, subtraction, multiplication, division that could act on natural numbers, thus covering the whole range of possible arithmetic operations. His way of defining numbers already provided a method for implementing loops (a fixed number of repeated applications of a function to data), but he was also able to show that there was a way of applying functions recursively until a condition was satisfied, providing a way to simulate the WHILE structure of modern programming languages. From today's point of view, lambda calculus is just another programming language, albeit a symbolic one. This explains the Church thesis: anything computable can be computed by the lambda calculus. Note that we call this statement a thesis, not a theorem, because it is just one possible definition of a computable function.

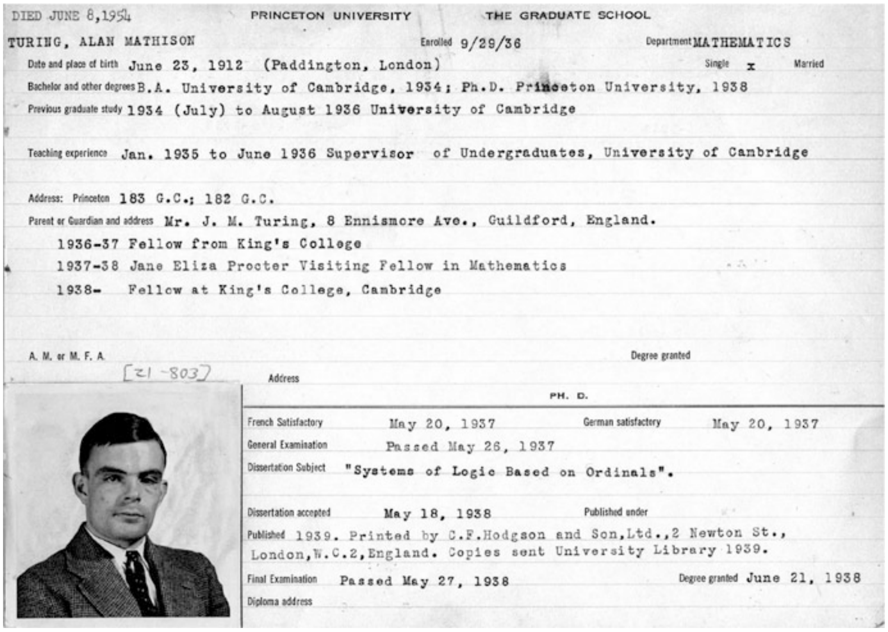
While Church was immersed in the lambda calculus at Princeton, Alan Turing was taking a different approach at Cambridge. He was interested in solving one of the problems formulated by the German mathematician David Hilbert at the turn of the century. It was the famous *Decision Problem*, which required proof of the existence of a finite mathematical procedure that could determine whether or not a given mathematical proposition follows from a list of axioms. To solve the problem, Turing invented what is now called a "Turing machine," which is a small processor with states and state transitions that operates on an unbounded string of symbols stored on a tape. There is a read/write head that moves along the tape, allowing the processor to modify the symbols one at a time. The current state and the symbol under the read/write head initiate a transition to a new state, replacing the previous symbol and then moving the read/write head one position to the left or right.

Turing was able to show that it is possible to implement many useful functions using his theoretical machine (that is, one can design the table of states and state transitions that are needed by the processor). We can have one machine for addition, another for multiplication, another for division, and so on. But then, Turing showed that it is possible to define a "universal machine," that is, one that can read any table of state transitions from its tape and apply it to symbols that are also stored in the tape. So we need only one Turing machine, the universal one, with its states and state transitions, and we need a memory (the tape) where we keep any specific table for a specific function and the data we want to use. The Universal Turing Machine is the "hardware" of our computer and the tables encoded in the tape are the "software" for a specific computation. This was all described in Turing's 1936 paper entitled "On Computable Numbers, with an Application to the Entscheidungsproblem," which appeared in the *Proceedings of the London Mathematical Society.* Turing's approach made it possible to define computability in terms of his universal machine, which is equivalent to the lambda calculus. This is why we now speak of the "Church-Turing thesis": what is computable by humans can be computed using the lambda calculus or a Turing machine (Fig. 2.2).

This is where the theory of computability began: at Princeton in a more theoretical and at Cambridge in a more operational way, so that theorists would later spend much time searching for the smallest possible Universal Turing Machine (in terms of the number of states and symbols used).

In 1936, there was also a person investigating the logical capabilities of relay circuits. It was Claude Shannon, who would later become the father of information theory. A year later, he submitted his master's thesis entitled "A Symbolic Analysis of Relay and Switching Circuits" to the Massachusetts Institute of Technology (MIT). There he studied the manipulation of binary variables using Boolean algebra with electronic circuits based on relays and switches.

We can then say that in 1936/1937, the invention of the computer was something that was being considered from a theoretical point of view by researchers such as Church and Turing and by engineers like Shannon and Konrad Zuse. But there was no sense of urgency, all these investigations were proceeding at their own pace, not dictated by any external pressure. There were many simple mechanical calculating machines in use at the time. There were also analog computers for solving differential equations. Vannevar Bush had just built his"Differential Analyzer" at MIT, where he represented continuous variables by the movement of gears and mechanical rods.



DIED JUNE 8,1954 PRINCETON UNIVERSITY THE GRADUATE SCHOOL

TURING, ALAN MATHISON Enrolled 9/29/36 Department MATHEMATICS

Date and place of birth June 23, 1912 (Paddington, London) Single x Married

Bachelor and other degrees B.A. University of Cambridge, 1934; Ph.D. Princeton University, 1938

Previous graduate study 1934 (July) to August 1936 University of Cambridge

Teaching experience Jan. 1935 to June 1936 Supervisor of Undergraduates, University of Cambridge

Address: Princeton 183 G.C.; 182 G.C.

Parent or Guardian and address Mr. J. M. Turing, 8 Ennismore Ave., Guildford, England.

1936-37 Fellow from King's College
1937-38 Jane Eliza Procter Visiting Fellow in Mathematics
1938- Fellow at King's College, Cambridge

A. M. or M. F. A. Degree granted

[21-8037] Address

PH. D.

French Satisfactory May 20, 1937 German satisfactory May 20, 1937

General Examination Passed May 26, 1937

Dissertation Subject "Systems of Logic Based on Ordinals".

Dissertation accepted May 18, 1938 Published under

Published 1939. Printed by C.F.Hodgson and Son,Ltd.,2 Newton St., London,W.C.2,England. Copies sent University Library 1939.

Final Examination Passed May 27, 1938 Degree granted June 21, 1938

Diploma address

**Fig. 2.2** Transcript of Alan Turing for his PhD work at Princeton University (Image: University Archives Princeton University)

Nor should we forget the Hollerith punch card machines, which were used in many companies to process statistical data and for other purposes. Invented by Hermann Hollerith in the 19th century, they provided massive processing power (for the time) for handling company databases. The company that sold these machines was IBM, formed in 1911 through a merger of companies that brought the rights to the Hollerith machines into its fold. Since the 1930s, IBM sold Hollerith tabulators for counting and sorting large amounts of data, collators that could combine information from multiple cards, punchers that could copy cards, and verifiers that could check cards. It was also possible to hardwire a specific calculation that had to be repeated for a deck of cards (using plug-in modules).

If we were to pinpoint a year that could be considered foundational in computing history, it would undoubtedly be 1936. That was the year that Alonzo Church was developing his theories in Princeton, Alan Turing was taking major steps forward in Cambridge, Claude Shannon was finishing his degree, and Konrad Zuse made the crucial decision to leave his position at Henschel Flugzeugwerke and found the world's first computer company. When the time is right, invention happens simultaneously.

## 2.2 Computers in Wold War II

1933 was a dramatic year in Germany. The chaotic situation in the country led to a rapid rise in the vote for the National Socialist Party of Adolf Hitler, who was appointed Chancellor on January 31, 1933 (although his party did not have a majority in the Reichstag). This date marks a turning point in German history. Albert Einstein and many Jewish scientists were forced to leave the country. Democratic institutions rapidly deteriorated under Hitler's regime, and on March 23, 1933, less than 2 months after Hitler became Chancellor, the Enabling Act granted him sweeping dictatorial powers, which he would immediately use to begin planning for the next war. There were only three more elections to the Reichstag before the war, but only the Nazi party was allowed to participate.

The second European war ignited slowly. First, Hitler stepped up the rearmament of Germany. This had already begun in the 1920s, with some companies building dual-use civilian aircraft that could also be used for military purposes. In 1936, the German army occupied the Rhineland, which was supposed to be a demilitarized buffer zone under the Treaty of Versailles. Then, on March 12, 1938, Hitler annexed Austria to the Third Reich. In September, he occupied part of the Czech Republic. A year later, Hitler invaded Poland, forcing Britain and France to finally declare war on Germany (Fig. 2.3).

When war broke out, there was a sense of urgency in Europe and in particular an urgency to develop machines that could break Germany's secret codes. Indeed, within days of the invasion of Poland, Alan Turing was recruited to the Government Code and Cypher School at Bletchley Park, near London, where he would work with other fellow cryptographers for the next 3 years.

### *2.2.1 John Atanasoff's ABC*

Of all the computer pioneers, only three began building their respective machines before the war: Konrad Zuse (1936), Howard Aiken, who had been seeking funding for his machine since 1937, and John Atanasoff, who began building his "Atanasoff-Berry computer" in 1938.

Atanasoff and his assistant Clifford Berry built their electronic computer at the University of Iowa between 1938 and 1942. While all other computers at the time used parallel arithmetic hardware, Atanasoff chose a serial approach. Two rotating drums stored up to 30 fixed-point numbers. Each number was stored in one sector of the drum (out of 30 sectors) using 50 capacitors, i.e., 50 bits. The capacitors could be charged (representing a one) or discharged (representing a zero). As the two drums rotated, fixed sensor heads could sequentially read the bits of two vectors (one for each drum), and the electronics could operate on pairs of numbers to produce a new vector, which was stored in one of the two drums. The machine was used to reduce rows of a matrix representing a set of simultaneous equations (with up

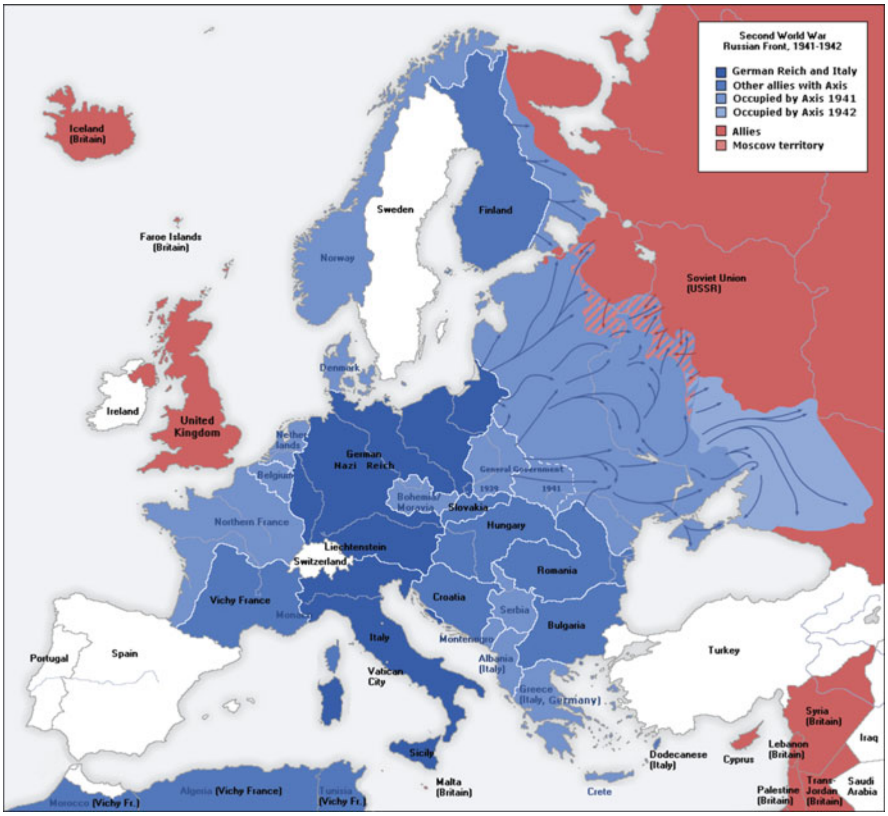


**Fig. 2.3** The map of Europe in 1941–1942, showing the advance of the Axis forces. Most of continental Europe was occupied. Spain was neutral, but was a tacit ally of Germany (Image: see page 223)

to 29 variables). One of the vectors was used to reduce the other, iteratively. The leading coefficient of the second vector was reduced to zero using the first vector and a combination of additions, subtractions, and shifts. The drums rotated once per second. The bits from the two vectors could be added sequentially using a small vacuum tube circuit. A shift could be performed by reading and rewriting the bits of a number, shifting them by one position (Randell 1982). The reduced vector could be stored as binary marks on a sheet of special paper. The vector could reloaded onto a drum reading the paper marks.

It is important to note that the ABC was not fully automatic, as this description makes clear. The machine required a human to operate on the vectors with the switches and select the next operation to be performed. The operator had to load each vector into a drum and command the storage of intermediate results. This was a computer, but one with a "driver" who followed the Gaussian reduction algorithm by pressing buttons until a solution to the system of linear equations was found. Of course, the machine was not universal, as it could not work continuously on its own. It was a special-purpose semiautomatic machine.

### 2.2.2 The Harvard Mark I

Howard Aiken led the construction of the Harvard Mark I at Harvard University between 1939 and 1944. He had been designing the machine since 1937, initially calling it the "Automatic Sequence Controlled Calculator." Aiken successfully secured funding from IBM in February 1939, with additional support later from the US Navy. Like Atanasoff, Aiken was a physicist who wanted to create a machine capable of performing the extensive calculations he needed. The Mark I was an electromechanical machine, like Zuse's Z4. It was a sort of hybrid between the mechanical nature of the Hollerith machines and the electronics available at the time. The machine was built by IBM and shipped to Harvard in February 1944. It was officially unveiled on August 7, 1944. It was a massive undertaking, containing 765,000 electromechanical components. They were synchronized with a rotating 15 m long metal shaft.

Aiken used the decimal system for the internal representation of numbers. Rotating gears were used to store and transmit numbers in memory (with 23 decimal digits). There was no sharp distinction between processor and memory, since each of the 72 memory cells was an accumulator capable of adding or subtracting the integers (fixed-point numbers) transmitted to it (Aiken and Hopper 1982). Each accumulator could transfer its contents to any other accumulator via what we would now call a bus. There were additional units to store tables of numbers and also a multiplier and a divider unit. Sixty constants could be set manually by the operator. The program was stored on a punched paper tape, and two reading units made it possible to run two programs in parallel. For checking purposes, the same program could be run twice. The code specified from which accumulator to read and to which accumulator to send the result. If two variables were needed (for a multiplication, for example), two punched cards were needed to specify the three accumulators involved (two for input and one for output) (Fig. 2.4).

The Mark I was not universal because it lacked conditional branching. Loops could be implemented by loop unrolling or by connecting the two ends of the control tape. An initial start tape initialized all parameters before the looped tape was inserted into the tape reader.

From this description, one might assume that the Mark I was similar to the Z4, and indeed it is, but the Mark I was a massive machine. The number of instructions was embarrassingly large, and there were also interpolation tables for functions such as logarithms, exponentials, and trigonometric functions. The oversized instruction set actually made it quite difficult to program the Mark I, as the programming manual clearly shows (Hopper et al. 1946).

The Mark I was used for military purposes before the end of World War II. It performed ballistic calculations and is believed to have been involved in calculations

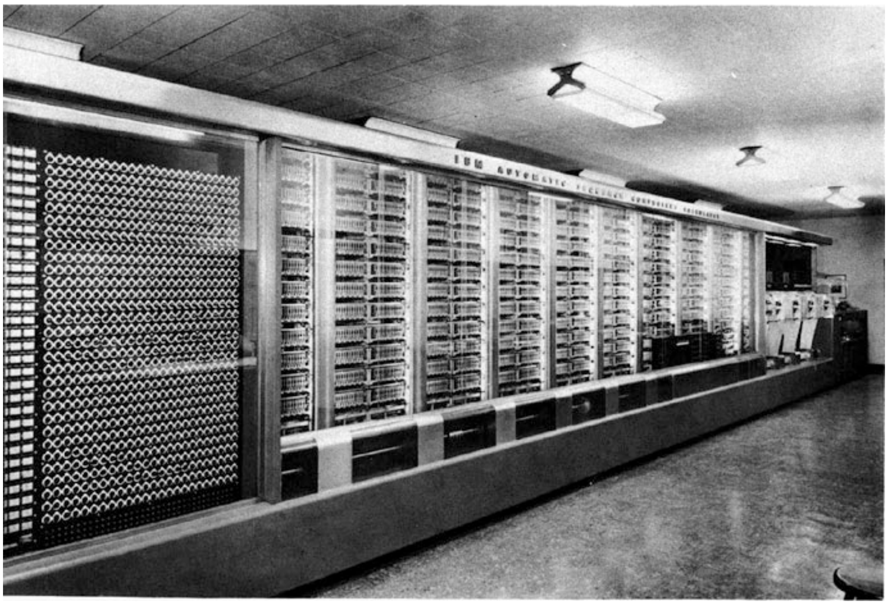


**Fig. 2.4** The Harvard Mark I (Hopper et al. 1946)

related to the Manhattan Project. During the 5 years of development of the Mark I (by a team of 70 people), Zuse built the Z3 and was finishing the Z4 in Berlin.

### 2.2.3 The ENIAC

The *Electronic Numerical Integrator and Computer* (ENIAC) was built at the Moore School of Electrical Engineering at the University of Pennsylvania between May 1943 and 1945. It solved its first problem in December 1945 and was officially unveiled in February 1946. Until that time, a "computer" was a person who did computational work. Indeed, this was one of the first jobs to be replaced by the emerging "electronic brains," as ENIAC was sometimes called (Fig. 2.5).

From a computer architecture point of view, ENIAC was a parallel dataflow machine. There was no separation between memory and arithmetic elements. Decimal fixed-point numbers were stored in any of 20 accumulators, each of which could transfer its contents to any of the other accumulators, or receive a number and add or subtract it from its stored contents (as in the Mark I). The ENIAC had a multiplier and a divider. Since the machine operated at a clock rate of 100,000 pulses per second, the designers felt that an external program (in a punched tape) would not fully exploit the computing speed of the ENIAC. Therefore, programs were hardwired, connecting with cables the output of one accumulator to the input

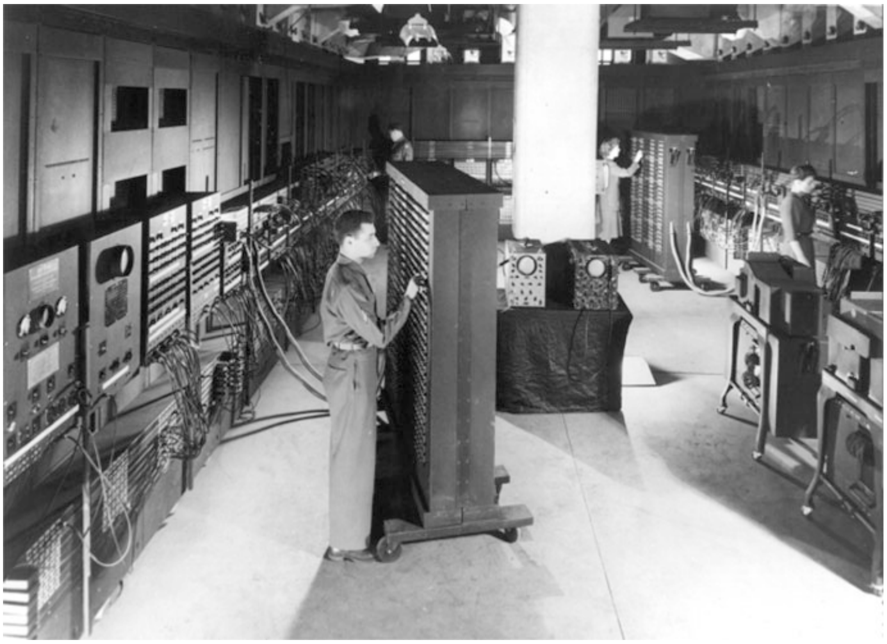


**Fig. 2.5** The ENIAC at the Moore School of Engineering, UPenn (Image: Wikimedia Commons)

of another, weaving in this way a computational graph representing the complete computation (Burks and Burks 1988).

Sequencing of operations was achieved by connecting the arithmetic units in the desired order, even in parallel. The accumulators were asynchronous and self-clocking. A unit that had finished its computation signaled this with a pulse to the next unit in the computation graph. ENIAC could perform all basic arithmetic operations, read constants stored in a special unit, and iterate a computation, since the master controller could restart a computation thread a fixed number of times (thus implementing a loop). The master controller could also stop a thread if an accumulator changed its sign. Although the original ENIAC design did not include the possibility of conditional branching, the machine operators quickly realized that this could be done by feeding the sign of an accumulator as a start signal for an accumulator at the beginning of a thread of computations.

But ENIAC's claim to fame is not its outdated and cumbersome architecture. It is its raw computing power, as it could run at the speed of vacuum tubes. It could perform a multiplication in 200 microseconds. That was 15,000 times faster than Zuse's Z3 and 30,000 times faster than the Mark I for that operation. And while Aiken's machine marked IBM's entry into the computer arena, the electronic computer opened the door to several new companies that would emerge in the following years. By the time ENIAC was officially unveiled, the war had been over for almost 10 months.

### 2.2.4 *Wunderwaffen and the Z4*

There is another computer that is sometimes mentioned as one of the first. It is the electronic calculating machine developed at Bletchley Park and called Colossus. However, the machine was not really a programmable computer, it was a special-purpose device that could try many possible decodings of the German military code until it found the right decoding key. Colossus was fast, as was ENIAC, but it is not a real contender for the title of the world's first computer.

Germany had essentially lost the war by the end of 1944, but Hitler decided not to capitulate, even though the American and Russian armies were approaching Berlin from two sides. The map in Fig. 2.6 shows the last remaining areas under the control of the German army by May 1945, when Germany surrendered. Konrad Zuse had left Berlin in February and made his way from Berlin to Bavaria, avoiding both armies. He was carrying the Z4, the only computer he had built during those years that survived the war. With the Z4 as his only asset, Zuse would rebuild his computer company in the years to come.

It is ironic that the Z4 was saved from destruction in Berlin because one of Zuse's employees arranged for its transport out of the besieged city, using the name of the

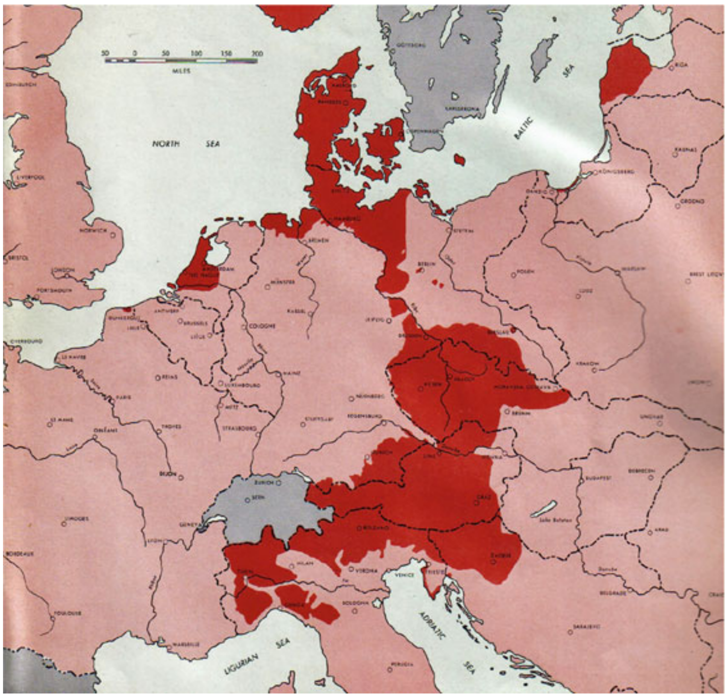


**Fig. 2.6** Last remaining regions under control of the German military on May 7, 1945 (Image: Wikimedia Commons)

machine (V4 until then) to suggest that it was another of the “Wunderwaffen” that would save Germany from defeat. More ironic still is that Zuse met Wernher von Braun’s team during his journey to Bavaria. He tried to distance himself at all costs from von Braun, who had been a member of the SS and was responsible for the deployment of the V2 rockets and the bombarding of London, as well as for the manufacture of the rockets exploiting slave workers. In Bavaria, the V4 promptly became the Z4 and was stored in a barn, while being refurbished, until one day a friendly Swiss mathematician came to see the unusual machine. It was Eduard Stiefel, a professor at the ETH in Zurich, who would later rent the machine for his mathematics institute. It was in Zurich in 1950, 8 years after construction began, that the Z4 performed its first truly useful calculations.

### *2.2.5 The First Computers*

As this book shows, the main flaw of the Z1, Z3, and Z4 was the lack of a conditional branch in the instruction set. It would not have been difficult to implement: although it is rather cumbersome when the program is stored on a tape, the necessary mechanism would have required only a few additional components.

Sometimes the dividing line between calculating machines and universal computers is drawn by distinguishing machines with externally or internally stored programs. I have argued elsewhere (Rojas 1993) that this is not a valid criterion. An external program can act as an interpreter of numerical data. It becomes an integral part of the processor, and the data become the program, much as a Universal Turing Machine works as an interpreter. I have argued that what is needed for universal computation is a minimal instruction set and indirect addressing (Rojas 1994). Indirect addressing can be simulated by writing self-modifying programs, so that the instruction set becomes the defining criterion. A machine with enough memory, an accumulator, and capable of executing the instructions CLR (clear), INC (increment), LOAD, STORE, and BZ (branch if zero) is a universal computer. In this sense, the Z1 and Z3 were not fully fledged computers, but neither were any of the other early machines. The ABC was a special-purpose machine for Gauss elimination, the Harvard Mark I lacked conditional branching, although it provided loops. The ENIAC was not even programmable by software: the building blocks had to be hardwired in dataflow fashion. Conditional branching was available in ENIAC, in a limited form, and self-modifying programs were out of the question.

Tables 2.1 and 2.2 show the most important information about the early computers mentioned above. As should be clear from the tables none of them fulfills all the necessary requirements for a universal computer. We also include the Mark 1 machine, which was built in Manchester from 1946 to 1948 because, as far as we know, it was the first machine to meet our definition of a universal computer.

The Mark 1 was built under the direction of F.C. Williams and T. Kilburn (Lavington 1975). This machine stored its program in a digital random access memory implemented with CRT tubes. All the necessary instruction primitives

**Table 2.1** Comparison of architectural features

| Machine | Memory and CPU separated? | Conditional branching? | Soft/hard programming? | Self-modifying programs? | Indirect addressing? |
|---|---|---|---|---|---|
| Zuse's Z1 | ✓ | × | Soft | × | × |
| Atanasoff's | ✓ | × | Hard | × | × |
| H-Mark I | × | × | Soft | × | × |
| ENIAC | × | Partially | Hard | × | × |
| M-Mark 1 | ✓ | ✓ | Soft | ✓ | × |

**Table 2.2** Some additional architectural features

| Machine | Internal coding | Fixed- or floating-point? | Bit-sequential arithmetic? | Architecture | Technology |
|---|---|---|---|---|---|
| Zuse's Z1 | Binary | Floating | No | Sequential | Mechanical |
| Atanasoff's | Binary | Fixed-point | Yes | Vectorized | Electronic |
| H-Mark I | Decimal | Fixed-point | No | Parallel | Electromechanical |
| ENIAC | Decimal | Fixed-point | No | Dataflow | Electronic |
| M-Mark 1 | Binary | Fixed-point | Yes | Sequential | Electronic |

were available (in modified form), and although it lacked indirect addressing, self-modifying programs could be written. The first program ran in June 1948 and computed the highest proper factor of $2^{18}$ by brute force. In September, Alan Turing was appointed Reader in Mathematics at Manchester and wrote some programs for the world's first universal computer. His vision of universal computation, published in 1936, the same year the Z1 memory unit was completed, had finally become reality. Tables 2.1 and 2.2 are clear: until 1945, there was no "first" computer, in the singular. Rather, the invention of the computer was a collective achievement that spanned two continents and 12 years.

## 2.3 Helmut Schreyer and the Electronic Computer

In Germany, someone also had the idea of building an electronic computer using vacuum tubes. It was suggested by Helmut Schreyer to Zuse when he first saw the Z1.

Helmut Schreyer is best known as Zuse's friend and colleague. They met around 1935 (Petzold 1985) and were members of the academic society "Motiv," a student association at the *Technische Hochschule Berlin*, now the Technical University of Berlin. Schreyer supported Zuse, provided him with an electric jigsaw for cutting the mechanical components of his Z1, assembled a paper tape reader for the calculating machine, and also suggested using telephone relays and even vacuum tubes for his computer. Schreyer is remembered today as one of the first to imagine the construction of an all-electronic calculating machine, without actually completing it.

### *2.3.1 The Closest Friend*

Little is known about Schreyer. Some biographical fragments have been published, but his life remains shrouded in mystery. This is surprising, considering that his name is often mentioned as a co-inventor of the computer. For example, a website of the Technical University of Berlin mentions Schreyer as a prominent alumnus of the electrical engineering department. In this section, we look at the Schreyer phenomenon and the contradiction between creativity, research, and an academic career during World War II.

So first of all, let's be clear: Helmut Schreyer became a member of the NSDAP in 1933. He was one of the early Hitler enthusiasts who flocked to the party in 1933. At that time, between January and April 1933, the number of party members rose from 850,000 to over 2.5 million. The number of applications was so great that on April 19, 1933, a ban on new memberships was imposed, effective May 1, 1933. It was not until 1937 that new members were admitted. The first members were guided by ideology, while the latter only wanted to make a career. Schreyer's party affiliation is significant because it may hold the key to other important events in his life (such as the fact that he was never called up for military service). It is also important because Zuse's attitude toward the political situation in Germany has sometimes been misrepresented. For example, a fictional Zuse in Friedrich Christian Delius' novel "The Woman for Whom I Invented the Computer" (Delius 2009) says, "There were no party members in my circle of friends." But no one was closer to Zuse than Schreyer.

Helmut Theodor Schreyer was born on July 4, 1912, in the small town of Selben, near Halle and Leipzig. He was the son of the pastor Paul Schreyer and Martha Schreyer, née Schlaich. In the curriculum vitae for his dissertation in 1941, Schreyer wrote: "In 1915 I came to Mosbach in Baden, where I attended elementary school in 1919 and then the Realgymnasium, graduating in 1933. After practical work at the General Electric Company, I began studying electrical engineering and telecommunications at the Technical University of Berlin in November 1934, graduating in December 1938." (Schreyer 1941) When Schreyer began his studies at the TH Berlin, he became a member of Motiv. According to Konrad Zuse (Zuse 1970), he visited his workshop in his parents' living room for the first time in 1937 and immediately suggested that the mechanical machine should be converted into an electromagnetic or electronic one. From 1938, Schreyer was a guest of Professor Wilhelm Stäblein at the "Institute for Vibration Research," probably as a graduate student. The previous name of the institute was the Heinrich Hertz Institute, but it was changed in 1933 because Hertz was of Jewish descent. Prof. Stäblein must have held Schreyer in high esteem, because in January 1939, just one month after Schreyer's graduation, he became his assistant in the Department of Telephone and Telegraph Engineering. Stäblein himself had a background in industry, having worked in the research department of AEG from 1927 to 1936. Schreyer was one of his first assistants, along with Herbert Raabe.

Being a student in Germany around 1930 was very different from today's era of mass universities. With a population of 65 million, there were only about 100,000 students in Germany. Today, there are 2.9 million students out of a population of 82.1 million. This means that in 1933 there were 1.5 students per thousand inhabitants; today there are about 23 times as many. Few families could afford the luxury of sending their children to university. Student societies and associations were at the end of their liberal heyday and were mostly conservative (Giles 1985). Konrad Zuse himself characterized the AV Motiv as conservative (Zuse 1970). In the Weimar Republic, however, the elite status of the students gradually collapsed. War returnees and the admission of women to the universities caused the number of students to rise from 79,000 in 1914 to 125,000 in 1924. In the years that followed, complaints about overcrowding in the universities grew louder. Students as a group became increasingly impoverished. This created a fertile ground for radical right-wing ideas.

Unfortunately, we do not know much about Helmut Schreyer's exact political views, but he certainly acted in accordance with the spirit of the times. Years before they came to power, the Nazis were already enjoying great success in student elections throughout Germany. One reason for this may have been the economic crisis and the resulting lack of prospects for students. The *Nationalsozialistischer Deutscher Studentenbund* was founded in 1926, but was able to take over the presidency of the German Student Union, the umbrella organization of student representatives, as early as 1931. The Nazis were more successful with students than with any other social group. As part of the process of conformity, the National Socialists ordered the transformation of student associations into "comradeships" and placed indoctrination and military and sports exercises at the center of their activities. Konrad Zuse tells in his memoirs that the AV Motiv changed its name to "Comradeship Wilhelm Stier" and that he and 10 other friends volunteered for a military exercise in the barracks (Zuse 1970).

We know that Zuse was called to the front twice (in 1939, when the war broke out, and in 1942, for only a week). The first time, in November 1939, Schreyer wrote to request Zuse's exemption from military service (Schreyer 1939). He referred to the computing machine under construction (the Z3) and its importance for scientific and military applications. The second time, Prof. Werner Osenberg helped Zuse as part of his crusade to encourage the return of scientists from the front. Osenberg, a member of the SS, was the head of the planning office of the Reich Research Council and tried very early on to bring home research capacity from the front. The fact that Schreyer was apparently never called up can only be explained by the protection of Prof. Stäblein or by his duties at the Institute for Vibration Research. In any case, membership in the NSDAP was useful for an academic career. Many professors and assistants lost their jobs during the "bringing-into-line period" and its aftermath. The road was clear for those who aspired to an academic career: "Between 1933 and 1939, about 45% of all university positions were filled, considerably more than is usually the case in a comparable period. The expulsion of the Jews accounts for about half of this" (Jarausch 1993). So was it Schreyer's careerism or his convictions that explain his party membership? His membership card lists May 1, 1933, as the

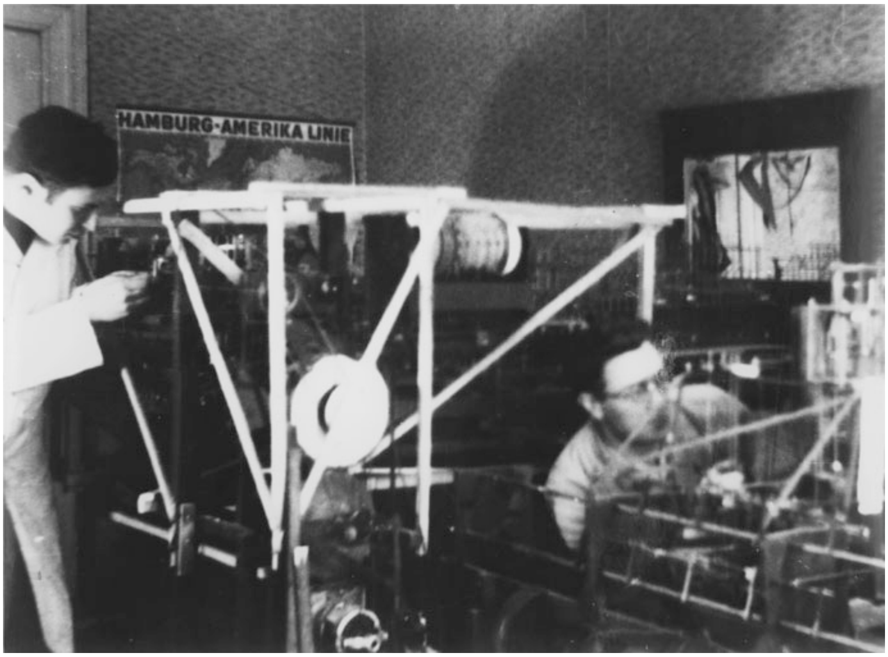


**Fig. 2.7** Helmut Schreyer (left) and Konrad Zuse (right) working on the Z1 (Image: Deutsches Museum)

day he joined the party, i.e., the exact date of the ban on new membership in the NSDAP. He became member 2.544.065. This means that Helmut Schreyer was one of the last applicants to be admitted (Fig. 2.7).

Hartmut Petzold was able to identify some of Helmut Schreyer's wartime activities from reports of the TH Berlin and the Institut für Schwingungsforschung (Institute for Vibration Research), which was one of the so-called "Four-Year-Plan-Institutes," i.e., institutions that were specifically funded and established in the course of the "Four-Year-Plan" for the rearmament of Germany, and from interviews: "In the following years Schreyer worked as an assistant on war-related work, including an accelerometer for the V2 rocket, detectors for unexploded bombs, and converters of analog radar measurements into acoustic signals for fighter planes. He was able to present the relay chain as a new type of frequency divider that worked perfectly from 1 to 5000 Hz. A larger computer circuit failed for lack of material." (Petzold 1985).

Like Zuse, Schreyer had a "day job" and an evening passion (the calculating machines). The day job was developing electronics for new weapons, the night job and passion was writing a dissertation entitled "The Tube Relay and its Circuit Technology," which he defended in October 1941 (Schreyer 1941). The accelerometer for the V2 rocket, for example, was designed to replace the use of radar to measure the speed of the rocket using the Doppler effect. A ground station tracked the rocket on radar and sent a radio signal to shut down the engine when it reached a certain speed. An accelerometer, if accurate enough, could have integrated the acceleration to provide the rocket's speed. This would have made the rocket fully autonomous and possibly more accurate. Apparently the accelerometer was not used.

Until 1933, Berlin was the European capital of science, but it suddenly became the capital of German weapons. In a secret memorandum of August 1936 on the "Four Year Plan," Hitler set the goal of getting Germany ready for war within 4 years. About 30 "Four Year Plan Institutes," 20 of them at universities, were established under the administration of Hermann Göring's Four-Year-Plan authority (Hachtmann 2003). The Institute for Vibration Research was one of them. This clearly outlined its scientific mission, i.e., weapons research. In 1933, Henschel Flugzeug-Werke was founded in Berlin Adlershof. This was the beginning of Henschel's involvement in the construction of aircraft and, a little later, remote-controlled bombs. This all happened while Zuse and Schreyer were students. *Nolens volens* they were caught up in the whirlwind: in 1935, Zuse began working as a structural engineer at Henschel, while 4 years later, Schreyer became an assistant at the Four-Year Plan Institute for Vibration Research.

Many new weapons were developed in Berlin as part of the preparations for the war and after it broke out. The first night vision systems, for example, were built by AEG in 1935. They were mounted on tanks and later on rifles. Helmut Hoelzer, the developer of the world's first electronic analog computer, was employed by AEG in Berlin from 1939 until he was sent to Peenemünde to work with Wernher von Braun (Tomayko 1985). A list compiled by German and Austrian researchers in 1947 mentions 1600 people living in Germany and Austria at the time who were relevant to the intelligence services. Of these, 135 were still living in Berlin (Anonymous 1947).

In a sense, the two students, Konrad Zuse and Helmut Schreyer, lost the ground beneath their feet. From today's perspective, it is difficult to understand why Zuse reveals so little about this period in his memoirs. The renaming of the AV motif was more than just an anecdote, it was the effect of the conformity enforced by Nazi ideology. The expulsion of Jewish students and professors is also hardly commented on. Zuse mentions only that three Jewish students "voluntarily" resigned from the Motiv Association. According to an official report from 1934, 134 Berlin university professors were expelled, 22% of all expelled scientists in Germany. The effects on German science, especially in Berlin, were clearly visible and must have caused great anxiety at the universities. Zuse and Schreyer must have had political exchanges with other students, of which there is little in the memoirs.

### *2.3.2 The Electronic Computer*

The rest of the story is well known. Helmut Schreyer designed some circuits for an electronic calculator, including a memory consisting of glow lamps, for which he applied for a patent (granted in 1943). Another application about the elementary electronic components for the calculating machine had already been filed in November 1940 (Petzold 1985). Schreyer also designed a special unit that could convert a decimal number into binary code. F.L. Bauer commented extensively on the elementary circuits (Bauer 2009). The designs were lost in a bombing raid toward the end of the war. Prof. Wilhelm Stäblein lost his life in an air raid on Erlangen/Nuremberg, where the Institute for Vibration Research had been evacuated. Wilhelm Stäblein's second assistant, chief engineer Herbert Raabe, lost his job in the course of the denazification of the universities (Butzer et al. 2009). Like Schreyer, he was a member of the NSDAP. Schreyer suddenly lost his main protector and other important contacts in the academic milieu. After the war, he was never able to regain a foothold in academic life. Finally, in 1949, Schreyer emigrated to Brazil, where he became a professor of electrical engineering at the Technical University of the Army and director of the Telecommunications Laboratory of the Brazilian Post Office. In 1977, he was made an honorary citizen of Rio de Janeiro. In the town of Delitzsch, formerly Selben, there is now a Dr. Helmut-Schreyer-Straße.

To understand Schreyer's circuits, it is important to note that although the logic of a binary machine can be built very easily using vacuum tubes, it would have been very expensive to use the same technology for all the components of the machine. In the USA, for example, John Atanasoff used capacitors to store bits and tubes to process them. The Manchester Mark I machine used a cathode tube as memory. Bits were burned in as dots and periodically refreshed to keep them shining. The afterglow of the cathode tube provided enough time for the refresh process. Schreyer, on the other hand, used a glow lamp for storage. The lamp could be switched on if a higher than normal voltage was applied at the same time as the cathode was activated (to store a 1). If the cathode was inactive (to store a 0), the lamp would not glow. With the regular voltage, the lamp continued to glow if it had been turned on before and could be read by a connected vacuum tube. All in all, this was a very simple construction that could have been implemented as a large memory with the technical means available at the time. In his dissertation of 1941, Schreyer described special elementary components that could be used for various tasks. The dissertation was classified as secret. Zuse suspected that Stäblein had the dissertation classified in order to land new armaments contracts (Petzold 1985).

It has been speculated whether or not Schreyer's machine could have been the first German electronic computer. However, Schreyer was by no means alone in this race. John Atanasoff has already been mentioned. When Atanasoff designed his (not universal) calculating machine in 1938, which was completed as a prototype with 300 vacuum tubes in 1939, Schreyer was just beginning to design his circuits. Construction of the American ENIAC began in 1942 and lasted until 1946. However, John Mauchly had already visited Atanasoff in 1941 and had been working on the idea of building an electronic computer ever since. He enrolled in electrical engineering at the University of Pennsylvania, even though he already had a doctorate in physics. The builders of ENIAC were all first-rate electronic engineers.

Helmut Schreyer would have been able to compete with the American designs only if his superiors had understood the importance of such a calculating machine early enough and if the elegance of Zuse's design had come into play. In his 1939 letter to the army, Schreyer describes an electronic machine capable of performing up to 10,000 operations per second (Schreyer 1939). But even Zuse was skeptical about the possibility of using electronic components. As late as 1952, he wrote: "In 1939, people laughed at us because we wanted to build electronic devices. Today they laugh at us because we did not build electronic machines. We thought at that time, the electronic machine is wonderful, but first there must be robust elementary gates. Until then, we will build electromagnetic devices. Today, if we could sell an electronic computer with a clear conscience, we would immediately use electronic components. But as far as we know, we have not reached that moment" (quoted in Petzold 2004).

## 2.4 Conclusion

Only very late after the war did Konrad Zuse receive the recognition he deserved. He gradually became famous, and his pioneering contributions were recognized. Helmut Schreyer could not achieve the same degree of fame because his own contributions never went beyond the mere possibility of electronic computing. Schreyer's experiments remained unfinished and completely unknown outside a small circle. As for his political views, Schreyer's membership in the NSDAP was mentioned in print by Paul Ceruzzi as early as 1983, but this was hardly noticed in Germany at the time (Ceruzzi 1983).

Perhaps it was Zuse's and Schreyer's bad luck that they were not among the many scientists transported to the USA in Operation Paperclip (Bower 1987). The so-called "Osenberg List" with the names of 15,000 German scientists fell into the hands of the Allies and was the basis for the first recruitment attempts by the Americans. The files of selected scientists were clipped, hence the name of the operation. Entire biographies were cleaned overnight, such as that of Wernher von Braun, a member of the SS and head of the German rocket program, who later rose to become director of NASA. Or Herbert Wagner, Zuse's supervisor at Henschel, who after the war made a career in the USA developing new cruise missiles. Had Stäblein not died in the war, he might have been brought to the USA. One can only speculate what Zuse or Schreyer could have achieved scientifically and commercially with the immense resources available in the USA.

We can only speculate about the reasons for Schreyer's emigration to Brazil after the war. We will never know: the generation of scientists of that time surrendered all too quickly to their rulers and then remained all too stubbornly silent. Until 1941, many were still able to get drunk on military success. At some point, however, the magnitude of the catastrophe could no longer be ignored. But neither Zuse nor Schreyer gave a thorough account of the 12 momentous years between 1933 and 1945, of what they knew, what they didn't know, or what they didn't want to know.

## References

Aiken, H.H., and G.M. Hopper. 1982. The Automatic Sequence Controlled Calculator. In *The Origins of Digital Computers*, ed. B. Randell, 203–222. Monographs in Computer Science. Berlin: Springer. https://doi.org/10.1109/EE.1946.6434251

Anonymous. 1947. Objective List of German and Austrian Scientists. Joint Intelligence Objectives Agency.

Bower, T. 1987. *The paperclip conspiracy – The hunt for Nazi scientists*. Boston: Little, Brown and Company.

Bauer, F.L. 2009. Helmut Schreyer – ein Pionier des "elektronischen" Rechnens. In *Historische Notizen zur Informatik*. Berlin: Springer. https://doi.org/10.1007/978-3-540-85790-7

Burks, A.R., and A.W. Burks. 1988. *The First Electronic Computer – The Atanasoff Story*. Ann Arbor: The University of Michigan Press.

Butzer, P.L., M.M. Dodson, P.J.S. Ferreira, J.R. Higgins, O. Lange, and P. Seidler. 2009. Herbert Raabe's Work in Multiplex Signal Transmission and His Development of Sampling Methods. *Signal Processing* 90 (5): 1436–1455. https://doi.org/10.1016/j.sigpro.2009.11.018

Ceruzzi, P.E. 1983. *Reckoners: The Prehistory of the Digital Computer, from Relays to the Stored Program Concept, 1935-1945*. Westport, CT: Greenwood Press.

Delius, F.C. 2009. *Die Frau, für die ich den Computer erfand*. Berlin: Rowohlt Verlag.

Giles, G.J. 1985. *Students and national socialism in Germany*. Princeton: Princeton University Press.

Hachtmann, R. 2003. *Science Management in the "Third Reich". History of the General Administration of the Kaiser Wilhelm Society*. Göttingen: Wallstein Verlag.

Hopper, G.M., H.H. Aiken, R.M. Bloch, and R.L. Hawkins. 1946. *A Manual of Operation for the Automatic Sequence Controlled Calculator*, Vol 1. Cambridge, MA: Harvard University Press.

Jarausch, K.H. 1993. The Expulsion of Jewish Students and Professors from Berlin University under the Nazi Regime. Lecture June 15

Lavington, S.H. 1975. *A History of Manchester Computers*. Manchester: Blackwell Publishers.

Meyer, B. 2000. Berlin – Stadt der Nobelpreisträger. Berlinische Monatsschrift 6. https://berlingeschichte.de/bms/bmstxt00/0004prol.htm

Petzold, H. 1985. *Computing Machines – A Historical Study of their Manufacture and Use from the Empire to the Federal Republic*. Düsseldorf: VDI Verlag.

Petzold, H. 2004. Hardwaretechnologische Alternativen bei Konrad Zuse. In *Geschichten der Informatik: Visionen, Paradigmen, Leitmotive*, ed. H.D. Hellige. Berlin: Springer. https://doi.org/10.1007/978-3-642-18631-8-5

Randell, B., ed. 1982. *The Origins of Digital Computers, 3rd edn. Monographs in Computer Science*. Berlin: Springer. https://doi.org/10.1007/978-3-642-61812-3

Rojas, R. 1993. Who Invented the Computer? The Debate from the Viewpoint of Computer Architecture. In *Fifty Years Mathematics of Computation*, ed. W. Gautschi, Vol. 48, 361–366. Proceedings of Symposia in Applied Mathematics. https://doi.org/10.1090/psapm/048/1314871

Rojas. R. 1994. On Basic Concepts of Early Computers in Relation to Contemporary Computer Architectures. In *IFIP 13th World Computer Congress*, 324–331.

Schreyer, H. 1939. Technische Rechenmaschine. Zuse Papers 004/002 www.zib.de/zuse

Schreyer, H. 1941. Das Röhrenrelays und seine Schaltungstechnik. PhD thesis, TH Berlin.

Tomayko, J.E. 1985. Helmut Hoelzer's Fully Electronic Analog Computer. *Annals of the History of Computing* 7 (3): 227–240. https://doi.org/10.1109/MAHC.1985.10025

Zuse, K. 1970. *Der Computer – Mein Lebenswerk*. Landsberg: Verlag Moderne Industrie,

# Chapter 3
# The Z1: Architecture and Algorithms of Zuse's First Computer

*This chapter provides a comprehensive description of the Z1, the programmable mechanical computing machine built by Konrad Zuse in Berlin between 1936 and 1938. We explain the main structural elements of the machine, the high-level architecture, and the internal data flow. The Z1 was capable of performing the four basic arithmetic operations using floating-point numbers. Instructions were read from a punched tape. A program consisted of a sequence of arithmetic operations, interspersed with memory store and load instructions, occasionally interrupted by input and output operations. Numbers were stored in a mechanical memory. The elementary mechanical components were Zuse's "mechanical relays." Notably, the Z1 did not include conditional branching in its instruction set.*[1]*.*

*While the architecture of the Z1 is similar to the relay computer Zuse finished in 1941 (the Z3), there are some significant differences. The Z1 implements operations as sequences of microinstructions as in the Z3, but does not use rotary switches as microsteppers. The Z1 uses a digital incrementer and a set of conditions that are mechanically transformed into a sequence of microinstructions for the exponent and mantissa units, as well as for the memory blocks. Microinstructions select the appropriate block and layers in the machine, using a vertical stack of control plates. The exception circuits (in case the mantissa is zero), which are necessary for normalized floating-point, were not implemented. Zuse knew that they were necessary, but they were first integrated into the Z3. In other words, the Z1, being a prototype, could not compute with zero.*

*The information for this chapter was gathered from careful study of the blueprints that Zuse drew for the reconstruction of the Z1 for the German Museum of Technology in Berlin, from some letters, and sketches in notebooks. Although the machine has been on display since 1989 (non-operational since the mid-1990s), no detailed high-level description of the machine's architecture has been available in print until now.*

[1] This chapter is based on two preprints: (Rojas 2014), (Rojas 2016)

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_3

## 3.1 Konrad Zuse and the Z1

Konrad Zuse (1910–1995) built his first computing machine between 1936 and 1938 (he experimented with small mechanical devices in 1934 and 1935). Zuse studied civil engineering at the *Technische Hochschule Berlin* (now the *Technical University of Berlin*). His first employer was the company Henschel Flugzeug-Werke, which had started building military aircraft in Berlin in 1933 (Materna 2010). The task of the 25-year-old was to carry out the long chains of structural calculations required for the manufacturing process of aircraft components. As a student, Zuse had already started thinking about ways to mechanize computation (Zuse 1970). So after only a few months working for Henschel, he decided to quit, build a mechanical computer, and start his own business, in fact, the first computer company in the world. At that time, he wrote a short document describing his vision of the "computer for the engineer," its structure, and how it could be programmed (Zuse 1936b).

During the period 1936–1945, Konrad Zuse was unstoppable, even after two brief calls to the front. He managed to be called back to Berlin to work part-time for Henschel and part-time for his own company. In those 9 years, he built four computers, as well as the two special-purpose machines. Zuse's original abbreviations for the names of the machines were V1, V2, V3, and V4 (meaning *Versuchsmodell*, or prototype). After the war, he changed the V to Z for obvious reasons. The V1 (hereafter Z1) was a fascinating technical feat: it was a completely mechanical computer, but instead of using gears and wheels to represent the 10 decimal digits (as Babbage had done in the previous century, and IBM had done with its Hollerith machines), Zuse decided to build a completely binary computer. He even wrote a document explaining the "dyadik," or binary system (Zuse 1937). His machine was based on components in which the forward linear movement of a small rod or metal plate represented a 1, and no movement represented a 0 (or vice versa, depending on the component). Zuse developed novel types of mechanical logic gates and finished the first prototype of the machine in his parents' living room. The sequence of events that led to the Z1 and subsequent machines was described by Zuse himself in his autobiography (Zuse 1970).

The Z1 was a mechanical but also surprisingly modern computing machine: it was based on the binary system, it used a floating-point representation for all numbers and could perform the four basic arithmetic operations. The program was read from a punched tape, and the results could be stored in or read from memory (16 words). The machine cycle was about 4 Hz.

The Z1 was very similar to the Z3, finished in 1941, whose architecture I first described in (Rojas 1997) (see Chap. 5). However, the detailed high-level architecture of the Z1 has never been published before. The original prototype was destroyed in a bombing raid in 1943. Only a few sketches and photographs of the mechanical components survived. In the 1980s, Konrad Zuse, who had retired many years earlier, received funding from Siemens and other German sponsors to build a full-scale replica of the Z1, which is now housed in Berlin's German

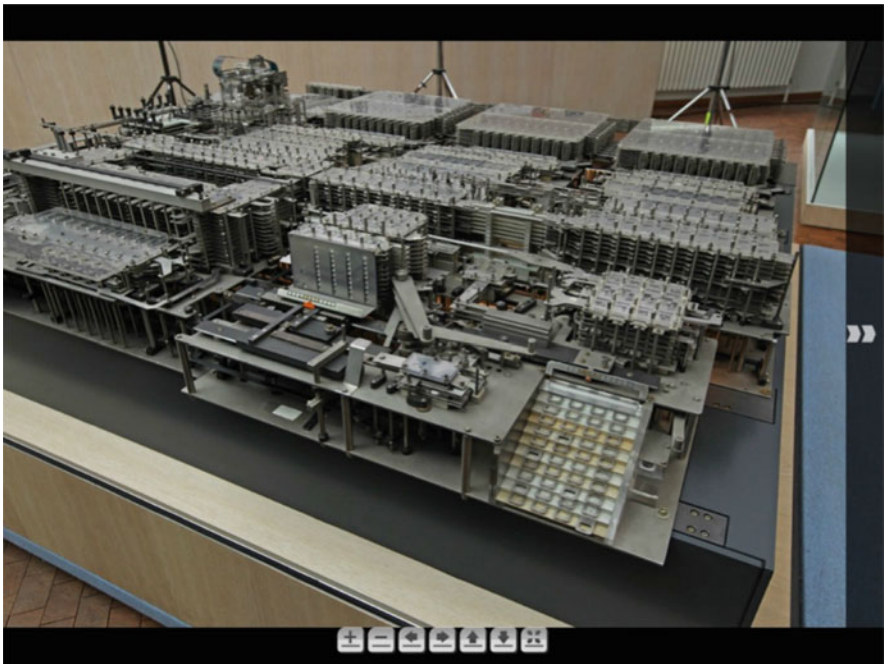


**Fig. 3.1** A view of the reconstructed Z1 in Berlin (from the Konrad Zuse Internet Archive: http://zuse.zib.de). The user can rotate the view around the machine and zoom in and out. The virtual display is based on thousands of linked photographs

Museum of Technology (Fig. 3.1). Zuse built the machine with the help of two engineering students: a complete set of blueprints was prepared, with drawings of every single mechanical component (to be cut from sheets of steel). Zuse supervised the reconstruction process over the course of several years at his own home in Hünfeld, Germany. The first sketches of the Z1 reconstruction were made in 1984. In April 1986, Zuse drew a timetable expecting to have the machine finished by December 1987. When the machine was delivered to the museum in 1989, it was shown running on several occasions. However, the reconstructed Z1, like the original, was never reliable enough to run unattended for long periods of time. When the machine jammed, Zuse had to personally travel to Berlin to have repairs done. Unfortunately, only a few modules of the machine have been shown in operation since his death in 1995.

Although we have a reconstruction of the Z1 in Berlin, fate struck twice. Other than drawing these blueprints, Zuse made no serious effort to write a complete top-down description of the reconstructed Z1. This would have been necessary, because it is evident from comparing the reconstructed Z1 with old photographs of the Z1 of 1938 that the new machine was "streamlined." The higher precision of the machining equipment available to Zuse in the 1980s allowed him to build the reconstruction using layers of steel plates that could be placed closer together

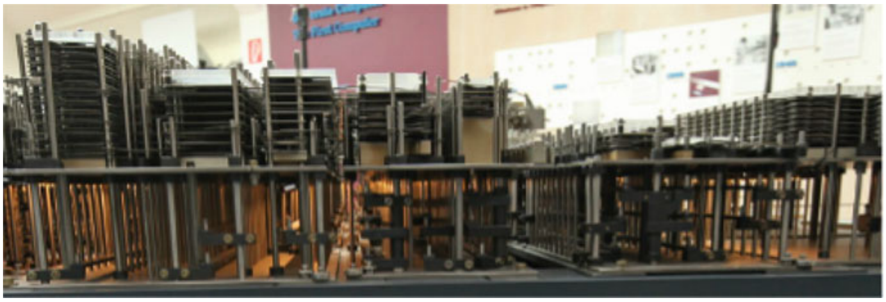


**Fig. 3.2** The mechanical layers of the Z1. The eight memory layers can be seen on the right; the 12 processor layers on the left. The lower section with levers is used for transmitting the clock cycles to all parts of the machine (http://zuse.zib.de)

(Fig. 3.2). The new Z1 has a significantly smaller volume than the old one. It is also not entirely clear whether the new Z1 is strictly a one-to-one mechanical clone of the logic of the original machine or whether Zuse's experience with the Z3 and later machines allowed him to improve parts of the reconstructed Z1. In the set of mechanical blueprints drawn between 1984 and 1989, there are at least six different designs for the addition unit, with between five and eight, and eventually up to 12 mechanical layers. Zuse left no detailed written record that would allow us to answer such questions. Worse, he rebuilt the Z1 and left no comprehensive description of it—for the second time! He acted like those celebrated watchmakers who only draw the components of their watches: first-rate watchmakers would need no further clarification. His two student assistants documented only the memory and the tape reader, a heaven-sent piece of information (Schweier and Saupe 1988). Visitors to the museum in Berlin can only marvel at the thousands of components visible in the machine. They can both marvel and despair, for it is almost impossible, even for professional computer scientists, to visualize the inner workings of this mechanical Leviathan. The machine is there—but in suspended animation.

This chapter is based on a careful study of the blueprints of the Z1, scattered annotations in Zuse's notebooks, and numerous on-site inspections of the machine. The reconstructed Z1 has been non-operational for so many years because the steel plates Zuse used bend under pressure. For this chapter, more than 1100 large-format drawings of the machine's components were reviewed, as well as 15,000 pages of notebooks (only a small fraction of which contained information about the Z1 though). I could only watch a short video of parts of the machine in operation (filmed more than 20 years ago). Deutsches Museum in Munich houses 1079 blueprints from Zuse's private papers, while the German Museum of Technology has another 314 in its archives. Fortunately, some of the blueprints also specify the definition and timing of some microinstructions for the Z1 and also some examples of bit-by-bit handwritten calculations done by Zuse. Such examples were probably used by the inventor to check the internal operation of the machine and to find bugs. This information was like a Rosetta stone, allowing us to correlate the Z1 microinstructions with the diagrams and blueprints and with our relatively deep knowledge of the relay-computer Z3 (for which we have complete circuits Rojas 1998). The Z3 is based on the same high-level architecture as the Z1 but differs in a number of important ways.

This chapter proceeds top-down: first, we review the block architecture of the Z1, the layout of the mechanical components, and we also provide some examples of the mechanical gates used by Zuse. Then we look in more detail at the core elements of the Z1: the clocked addition units for exponent and mantissa, the memory, and the microsequencer for arithmetic operations. We show the interplay of the mechanical elements and how the "sandwich" layout of steel plates helped Zuse organize the computation. The appendix describes the arithmetic and I/O operations of the Z1 using tables.

## 3.2 Block Architecture

The Z1 was a clocked machine. Being a mechanical device, its mechanical clock signal was divided into four subcycles that consisted of the movement of mechanical components in four orthogonal directions, as shown in Fig. 3.3 (left side, see

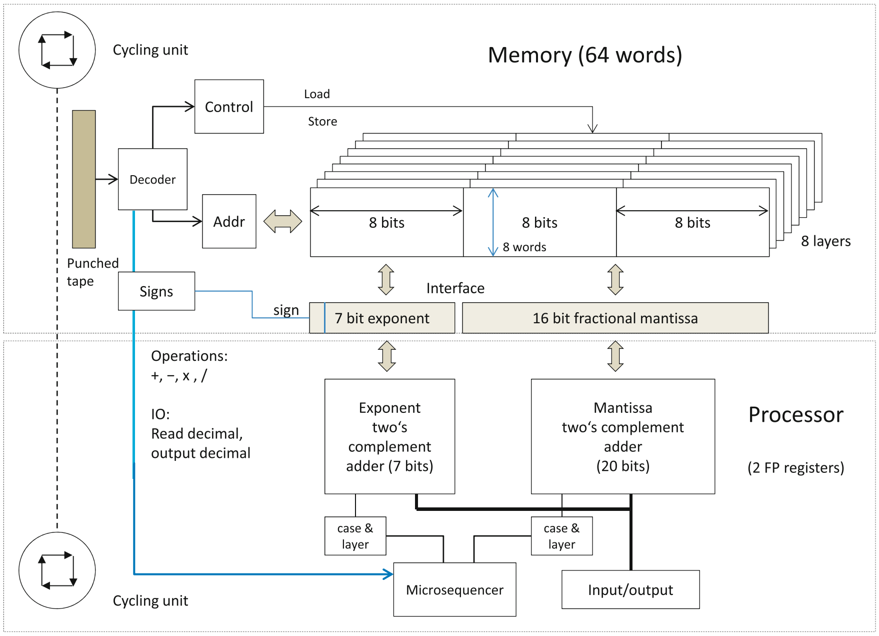


**Fig. 3.3** Block diagram of the Z1 (1936–1938) in accordance with the reconstruction of 1989. The original Z1 had only 16 words of memory instead of 64. The punched tape was made of 35mm film tape. Each instruction was encoded using 8 bits

"cycling unit"). Each movement direction was called an "engagement" by Zuse. He aimed for a 4 Hz clock cycle, but the Berlin reconstruction never operated faster than at 1 Hz (with four subcycles per second). At this speed, one multiplication takes around 20 seconds.

The Z1 has a number of features that were later adopted in the Z3. From a modern perspective, the most important innovations in the Z1 (see Fig. 3.3) were the following:

(a) It was based on a fully binary architecture for the memory and the processor.
(b) The memory was separated from the CPU. In the Berlin reconstruction, the memory and punched tape reader make up about half of the machine. The processor, I/O panels, and the microcontrol unit make up the other half. The original Z1 had 16 words of memory; the reconstruction has 64.
(c) The machine was programmable: instructions were encoded on a punched tape using eight bits (two bits for the opcode and six bits for memory addressing; for operations, three bits were used to encode the opcode of the four arithmetic and two I/O operations). Thus, there were only eight instructions: the four basic arithmetic operations, also "load-from" and "store-to" memory, one instruction to read data from a decimal panel, and another to display the contents of the result register on a mechanical decimal display.
(d) Floating point was used for internal data representation, in both memory and processor. Therefore, the processor was divided into two parts: one for handling the exponents and one for handling the mantissas. In memory, the mantissa had 16 bits for the bits after the binary point. The bit to the left of the point was always one (normalized floating point) and did not need to be stored in memory. Exponents were represented with 7 bits in two's complement format (thus running from $-64$ to $+63$). The sign of the floating-point number was stored in an additional bit. Therefore, the word length in memory was 24 bits (16 bits for the mantissa, 7 for the exponent, and 1 bit for the sign).
(e) The special case of zero in arguments or results (which cannot be expressed with a normalized mantissa, where the leading bit is always 1) can be handled within the floating-point representation as special values of the exponent. This was done in the Z3, but not in the Z1 nor in its reconstruction. Therefore, the original Z1 could not work correctly with zero as an argument or intermediate result. Zuse was aware of this enormous shortcoming, but he left the solution to the relay machine, which was easier to wire.
(f) The CPU was microcoded: operations were broken into sequences of microinstructions, one for each machine cycle. The microinstructions produced a specific flow of data within the Arithmetic Logic Units (ALUs), which ran nonstop, adding in every cycle whatever two numbers were stored in its two input registers.
(g) Curiously, memory and processor ran independently: the memory would put data on, or retrieve data from, the communications interface, whenever the punched tape gave the command. The processor would fetch, or put data on the interface, when a load or store operation was executed. It was possible to

run only the processor and shut down the memory, in which case the data on the interface, supposedly coming from the memory, would be zero. It was also possible to run only the memory and shut down the processor. This allowed Zuse to debug each half of the machine independently. When running together, a shaft connecting the cycling units in each half synchronized both parts of the machine.

Further innovations in the Z1 were similar to some of the ideas later present in the Z3. The instruction set was virtually the same, but the Z1 could not extract square roots. The Z1 used discarded 35mm film tapes as punched tape. In 1936, Zuse had briefly considered storing the main program and the subprograms in memory along with the data, but since the Z1 had a very small memory unit, a punched tape was a practical alternative (Zuse 1936a).

Figure 3.3 shows the abstract diagram of the reconstructed Z1. Note the two main halves of the machine: the memory is in the upper half, and the processor is in the lower half. Each half had its own rotating cycling unit, which further divided each cycle into four mechanical movements in the directions indicated by the arrows. These four movements could be communicated to any part of the machine by means of levers distributed under the computing components. The punched tape was read one instruction at a time. The instructions had different durations. Load and store operations took one cycle; all other operations needed several cycles. The memory address was contained in the lower six bits of the 8-bit opcode, allowing the programmer to refer explicitly to 64 memory addresses.

Memory and processor communicated through a buffer between the two units (the components 12abc in Fig. 3.5). In the CPU, the internal representation of the mantissa was extended to 20 bits: two additional bits were used before the binary point (for the binary powers $2^1$ and $2^0$), and two additional bits for the lowest order binary powers ($2^{-17}$ and $2^{-18}$), to increase the accuracy of the CPU for intermediate results. Therefore, in the processor, the mantissa had 20 bits representing the binary powers from $2^{+1}$ to $2^{-18}$.

The decoder would take an instruction from the punched tape reader, decode the operation, and start controlling the memory unit and processor, as needed. A number could be read from memory into the first of two CPU floating-point registers (using a load operation). Another load operation would read a number from memory into the second CPU register. The two registers could be added, subtracted, multiplied, or divided in the processor. Such operations require the addition or subtraction of the exponents (with a two's complement ALU) while another ALU is needed for the mantissas. The sign of the result of a multiplication or division is handled in a special "sign unit."

An input instruction made the machine enter into idle mode. This allowed the operator to enter data by pulling four decimal digits from a mechanical panel, and by entering the exponent of the floating-point representation with a small lever, and also the sign of the number. The operator could then restart normal operation. An output instruction also made the machine idle and displayed the contents of the result register on a decimal mechanical panel until the operator pressed a lever to restart normal operation.

The microsequencer in Fig. 3.3, along with the exponent and mantissa addition units, constitutes the core of the computation capabilities of the Z1. Each arithmetic or I/O operation was divided into "phases." The microsequencer started counting them and selected the appropriate microoperation in the corresponding layer, out of 12 possible layers of mechanical components in the addition units.

Therefore, a minimal program in a punched tape could be, for example: (1) load number from address 1 (implicitly into the first CPU register), (2) load a number from address 2 (implicitly into the second CPU register), (3) add, (4) display the result in decimal. This program allowed the operator to use the Z1 as a simple mechanical calculator. Of course, the sequence of computations could be much longer: they were programmed using the memory as storage for constants and intermediate results (in the latter Z4 computer, one tape once used for mathematical computations was 2 meters long).

The architecture of the Z1 can be summarized using modern terminology as follows: it was a programmable normalized floating-point Harvard architecture machine (processor and memory were separate), with an external read-only program, and a memory of sixteen 24-bit words. It was capable of accepting as input decimal numbers of four digits (and an exponent as well as a sign), for conversion to binary. It could perform the four arithmetic operations on the data. The binary floating-point result could be transformed back into decimal scientific notation readable by the user. There was no conditional or unconditional branching in the instruction set. There was no exception handling for zero results. Each instruction was decomposed into microinstructions "hardwired" in the machine. A microsequencer orchestrated the execution of the microinstructions. In an old video of the mechanism in action, it looks to the naked eye like the moving parts of an automatic loom. But this machine was weaving numbers.

## 3.3 Layout of the Mechanical Components

The Berlin reconstruction of the Z1 is based on a very clean layout. All mechanical components are optimally arranged. We have mentioned that Zuse designed at least six different versions of the processor. The relative positions of the main blocks were fixed from the beginning and may reflect the original distribution of the mechanical elements in the original Z1. There are two main divisions: a gap separates the memory from the processor. In fact, both parts of the machine can be pulled apart for debugging purposes, as they are mounted on separate tables with rollers. Another horizontal plane divides the machine into an upper part containing the computational components (those visible in photographs of the Z1) and a lower part containing all the synchronization levers. This "underworld" is only visible when the visitor bends down to look under the computational skyline. Figure 3.4 is a drawing from the blueprints showing the computation and synchronization layers for part of the processor. Note the 12 layers of computational components and the lower section with three levels for levers. This blueprint is a good example of how difficult it can

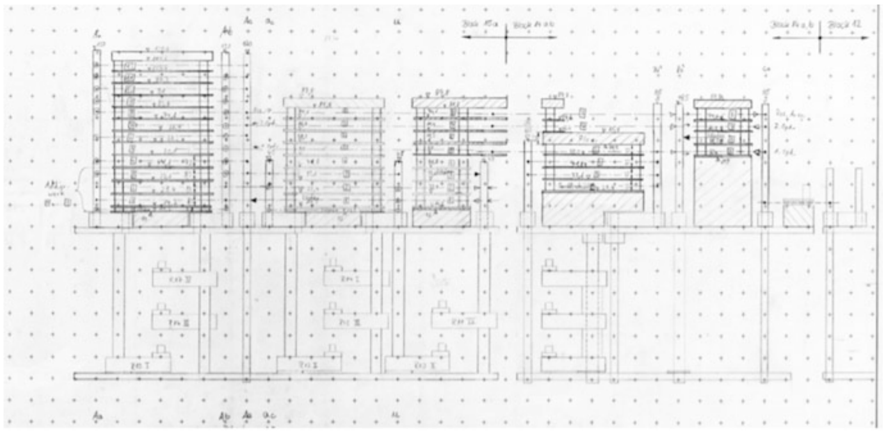


**Fig. 3.4** Schematics of the computation and synchronization layers of the exponent's unit of the Z1 (http://zuse.zib.de)

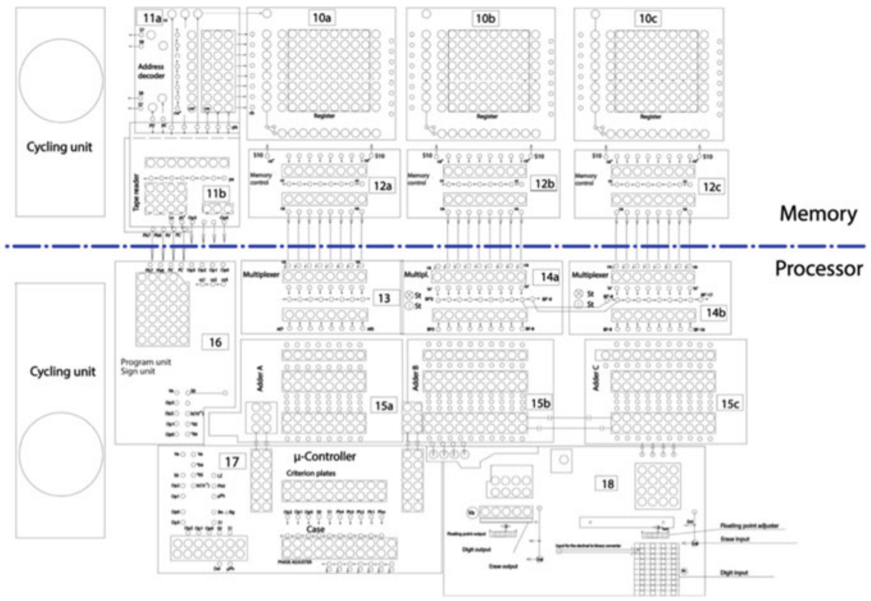


**Fig. 3.5** Diagram of the Z1, showing the mechanical building blocks

be to interpret the drawings. While there are many details about the size of the parts, there are few notes about their use.

Figure 3.5 shows the distribution of logic components in the reconstructed Z1, seen from above and as drawn by Zuse, further annotated with the logic functionality of each block (this sketch has been available since the 1990s). At the top we see the three memory banks. Each can hold eight 8-bit words per layer. Each bank has eight mechanical layers, so that a total of 64 words can be stored. The first memory bank (10a) is used for the exponent and sign. The last two banks (10b, 10c) are used for the lower 16 bits of the mantissa of the stored numbers. This bit distribution allowed Zuse to build three identical 8-bit memory banks and use them for exponent and mantissa, thus simplifying the mechanical design. Between memory and processor, there is a "buffer" to pass numbers to the processor (blocks 12abc) or to receive numbers from it. There is no way to encode constants in the punched tape. All numbers must to be entered by the user using the decimal input panel (block 18, right side) or must be generated by the computer itself as intermediate results.

Each unit in this diagram shows only the top vertical layer. Remember that the Z1 is built like a "sandwich" of mechanical parts. Each computational layer is separated from the layer above and below it (each layer has a metal floor and a metal ceiling). The communication between the layers is provided by vertical rods that can transmit motion from one layer to the neighboring layers. The vertical rods are the small circles drawn outside the rectangles representing layers of computation. The slightly larger circles drawn inside the rectangles represent logic operations. Inside each circle, we can find a binary gate (and going down through the layers up to 12 gates for each circle). This drawing allows us to estimate the number of logic gates present in the Z1. Not all units have the same height, and not all layers are populated with mechanical components. A conservative estimate of the number of binary elements would be 6000 gates.

Zuse assigned the numbers shown in Fig. 3.5 to the different modules of the machine. The purpose of the modules is as follows:

**Memory Block**

- 11a: Decoder for the 6-bit memory addresses
- 11b: Punched tape reader and opcode decoder
- 10a: Memory bank for 7-bit exponents and sign
- 10b, 10c: Memory banks for the fractional part of the mantissa
- 12abc: Interface for load and store operations to and from the processor

**Processor Block**

- 16: Control and sign unit
- 13: Multiplexer for the two ALU registers in the exponent part
- 14ab: Multiplexer for ALU registers, one-bit two-way shifters for multiplication and division
- 15a: ALU for the exponent
- 15bc: 20-bit ALU for the normalized mantissa (18 bits for the fractional part)
- 17: Microcode control
- 18: Decimal input panel on the right, output panel on the left

One can imagine computation flowing in this diagram from top to bottom: the data come from memory to fill the two registers, called F and G, available to the programmer. These two registers are distributed along blocks 13 and 14ab. The two registers are fed to the ALUs (blocks 15abc). The result is cycled back to register F or G (as result register), or back to memory. The result can be shown in the decimal display using the “re-translate” instruction (binary to decimal conversion).

In the following, we look at each module in more detail, concentrating on the main computational components.

## 3.4 The Mechanical Gates

The mechanical structure of the Z1 can be best understood by looking at a few simple examples of the type of binary logic gates that Zuse used in his machines. The classical representation of decimal digits has always used gears. A gear is divided into 10 sectors—by turning the gear, it is then possible to count from 0 to 9. Zuse decided as early as 1934 to use the binary system (which he called the dyadic system, following Leibniz).

The basic logic component used by Konrad Zuse in the Z1 was the binary “mechanical relay” (Zuse 1952b). The logic components could only move one step in one direction (Zuse arranged the components on a plane so that the permitted directions of movement were West, South, East, and North). The initial state of each component is state 0. Its state after a linear displacement is state 1. The components could move back and forth between the states 0 and 1. Logic gates pass movement from one plate to another, according to the value of the bits represented. The structures are three-dimensional: they consist of arrays of superimposed planar plates that transmit movement usually through cylindrical pins positioned vertically at right angles to the plates.

Figure 3.6 shows a diagram of a mechanical relay. Bit A is called the “control bit” or “control element.” On the left side of Fig. 3.6, we see the case where the initial state of bit A is such that there is no mechanical coupling between the actuator and the actuated plate: the motion of plate B is not copied to bit C when A is zero. However, if A moves down to its position 1, then mechanical coupling is achieved, and the movement of plate B is copied to bit C. This mechanical relay then represents only a one-step delay.

The right side of Fig. 3.6 shows the case where the initial state of A (that is A=0) is such that the mechanical coupling between plates B and C is present. When A moves to state 1 (up), the moving plate B loses its mechanical coupling to plate C. In this case, bit C will be the negation of bit A: when bit A is 0, C is 1 at the clock signal. When bit A is 1, bit C is 0.

Figure 3.7 shows how to implement the AND and OR logic gates using two mechanical relays. In the case of the AND circuit, the mechanical movement of the actuator plate (activated when a clock signal arrives) is only transmitted when A=B=1. In the case of the OR circuit, the movement is transmitted when A or B is equal to 1.

An interesting aspect of such mechanical constructions is that a long logic formula can be computed with zero delay, in principle, by such mechanical arrangements (assuming that motion between rigid plates is transmitted instantaneously).

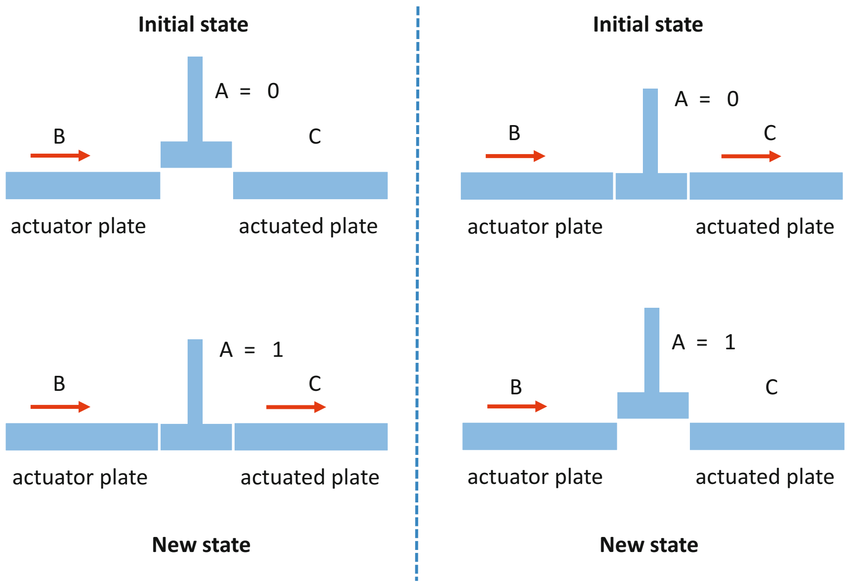


**Fig. 3.6** A mechanical relay with two different initial states of the control plate. On the left, the initial state does not provide mechanical coupling. On the right, the initial state provides mechanical coupling. The initial state is called 0; the state after a displacement is called 1

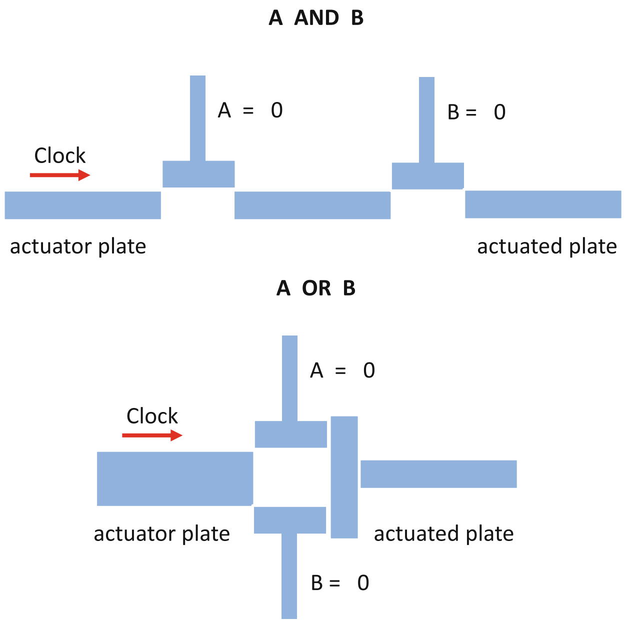


**Fig. 3.7** Two mechanical logic gates: AND on top, OR at the bottom

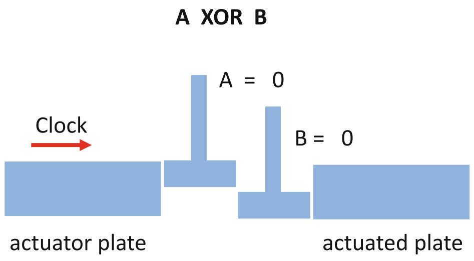


**Fig. 3.8** An XOR gate made of mechanical relays

A conjunction of 100 bits, for example, could be computed by concatenating 100 mechanical relays (which couple to their neighbors in state 1). Once all 100 control bits have been set, the actuator plate will move the actuated plate only if all bits are equal to 1.

It is easy to see that any kind of logic gate can be constructed using a mechanical relay. An XOR, for example, can be obtained as a variation of the AND gate with different initial conditions for the bits A and B, as shown in Fig. 3.8. Bit A moves down when it is equal to 1; Bit B moves up.

Now, Zuse did not use exactly this kind of mechanical arrangement, because the main problem is to make sure that all parts will move but also come back to their initial position. Control plates (such as bit A in Fig. 3.6) move in one direction and then in an orthogonal direction when the movement of the actuator plate is transmitted. It is not easy to do this with mechanical components. Therefore, Zuse's idea was to use small vertical rods as "connectors" and to let them move between two horizontal planes made of metal or glass.

Figure 3.9 shows what Zuse called the "elementary gate." The "actuator plate" can be regarded as the motion coming from the machine cycle. This plate moves cyclically from right to left and back again. The top plate is the data bit we use for control. It can be in position 1 or 0. The rod that goes through the openings displaces horizontally following the plate (keeping its verticality). If the upper plate is in the 0-position, the movement of the actuator plate cannot be transmitted to the actuated plate (see Fig. 3.9, left side). If the data bit plate moves to the 1-position, the movement of the actuator plate is transmitted to the actuated plate. This is what Konrad Zuse called a "mechanical relay," just a switch that closes a mechanical "current." This elementary gate can thus copy a bit from the upper to the actuated plate, rotating the movement of the bit by 90°.

The verticality of the connecting rods of the elementary gates was maintained by making them thick and short and constraining them to slide between two sheets of glass or metal. The glass minimized the friction with the rods, which could move in two orthogonal directions. This is what produces the obscure mechanical drawings that Zuse made during the different stages of the construction of the Z1. The main idea, as we have seen, is simple, but its mechanical realization is somewhat involved. Having said that, it must be pointed out that the vertical rods used by Konrad Zuse in most of his mechanical relays (Figs. 3.9 and 3.10) were an accident waiting to

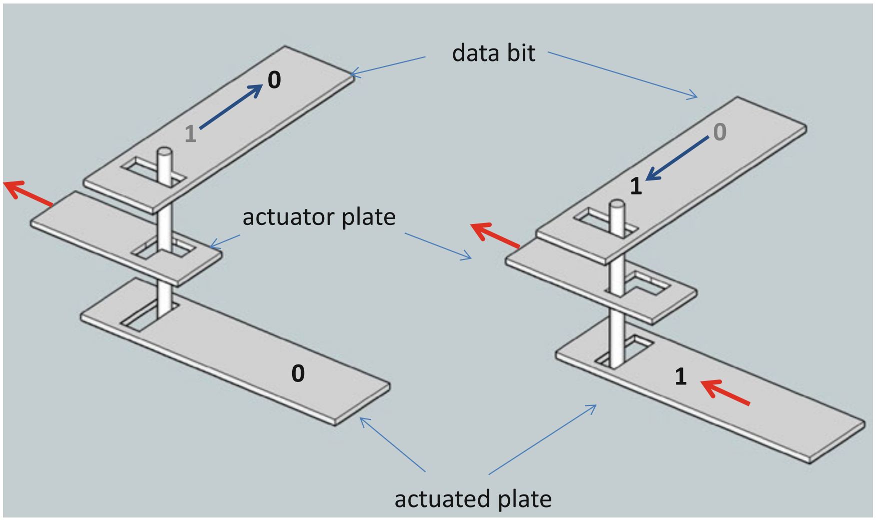


**Fig. 3.9** An elementary gate is a switch. If the data bit is 1, the actuator (or actor) and actuated plates are connected. If the data bit is zero, they are disconnected and the movement of the actuator plate is not transmitted to the actuated plate

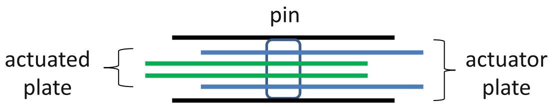


**Fig. 3.10** A mechanical relay and its control rod/pin sandwiched between plates

happen. The rods moved vertically, sandwiched between glass or metal plates, and had to be pulled or pushed symmetrically and gently to keep them from falling over. This was achieved by duplicating the actuated and actuator plates. The result was never completely satisfactory, except for the mechanical memories built by Zuse, which were still being used for the Z4 in the 1950s.

Figure 3.11 shows such plate arrangements as seen from above. The actuator plate is shown as a rectangle with its opening. The control plate (the data bit) in green pulls the filled circle (a rod) up or down. The actuated plate (red) can move to the right or left, but only when the rod is in such a position that the actuator's opening moves the rod. For each mechanical gate, seen from above, there is a drawing of the equivalent switch to the right. The control bit can close or open the gate. The actuator plate can be pulled or pushed (as shown by the arrows). Zuse's convention was to always draw the switch in the zero position of the control bit, as done in Fig. 3.11. Zuse preferred plates to be pushed by the actuator plate (right side of Fig. 3.11) rather than pulled (left side of Fig. 3.11). It is now very easy to build a negation gate by using a closed switch that is opened by setting the control bit to 1 (as shown in the bottom two diagrams in Fig. 3.11).

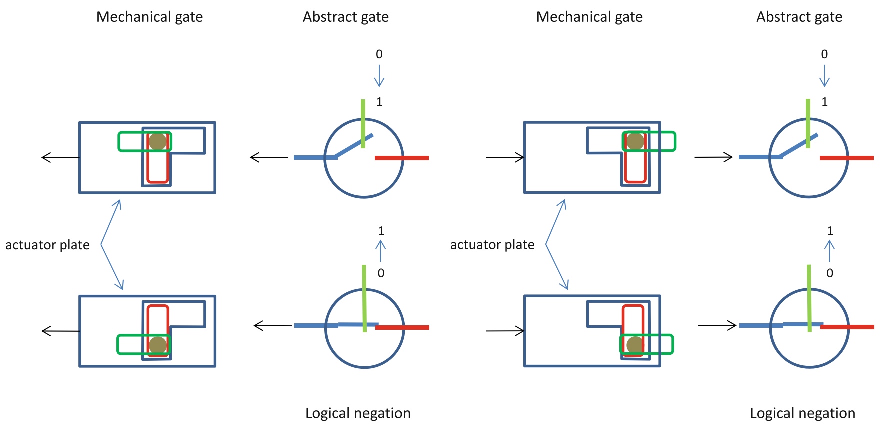


**Fig. 3.11** Some variations of the elementary gate and Zuse's abstract notation for mechanical relays. The relays are drawn as switches. By convention, the drawing always shows the position zero of the control bit. The arrows show the possible movements. The actuator plate can be pulled to the left (left side diagrams) or pushed to the right (right side diagrams). The initial position of the mechanical relay can be in the closed position (lower two diagrams). In that case, the relay acts as a negation since the output is the negation of the control bit

There are many possible mechanical realizations for the main idea, and Zuse showed great creativity always drawing the variation of a gate that best corresponded to the 3D structure of the machine (Zuse 1952c).

With a mechanical relay, it is now straightforward to build the rest of the logic operations. We show examples of the three basic gates: conjunction, disjunction, and negation. Zuse developed a symbolic notation for his relays that abstracts from the mechanical nature of the devices and emphasizes the logic properties. Figure 3.12 shows the necessary circuits, now using only the abstract notation. The equivalent mechanical realizations are easy to imagine. As always, the initial position shown in the diagram corresponds to state zero. Motion corresponds to state 1. A relay can be opened or closed by changing from state 0 to state 1.

Zuse designed mechanical gates where the movement could be obtained by pushing with the actuator plate (as in my examples in this section), but it is also possible to design variations where the actuator plate can pull the actuated plate. In Fig. 3.12, this possibility is indicated by the direction of the arrows.

Now everyone can start building his/her own Zuse mechanical computer. The basic element is the mechanical relay. More complex connections (like the relays with two actuated plates) can be designed, and the corresponding mechanics must be built with plates and rods (Zuse 1952b).

The main problem in building a complete computer is to connect all the components. Note that the control bit always moves orthogonal to the result bit. Each completed logic operation rotates the mechanical movement by 90°. The next logic operation rotates the movement by 90° and so on. After four gates, we are

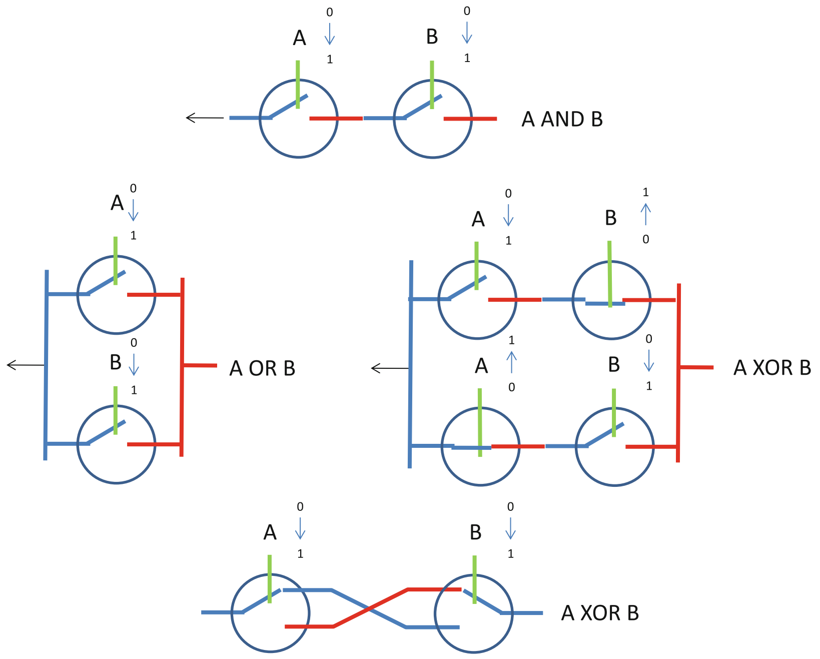


**Fig. 3.12** Some logic gates built from mechanical relays in abstract notation. The lowest diagram, an XOR, can be built by using mechanical relays with two possibly actuated plates, as shown in the diagrams. The mechanical equivalents are easy to design

back to the original direction of motion. This is why Zuse's cycling units used the four directions N-E-S-W. Within one machine cycle, it is possible to execute four layers of logic computations. The logic gates can be simple, such as a negation, or complex, such as an operation with two actuated plates (in an XOR). The timing in the Z1 is such that the machine completes an addition in four engagements: in engagement IV, the arguments are loaded. Engagements I and II compute partial sums and carries, and engagement III provides the final result.

Result bits can be transferred to different horizontal levels than the level at which the input bits move. That is, rods can also be used to move bits "up" or "down" between the layers of the machine. We will see this later for the addition circuits.

At this point, Fig. 3.5 should make more sense: the circles inside the different units are exactly the circles of Zuse's abstract notation and pinpoint the position of the logic gates. We can now abstract from the mechanics and discuss the Z1 from an architectural point of view.

### 3.4.1 The Mechanical Clock Cycle

A complex computer requires the presence of feedback loops in its circuits. The result of a previous step must be fed back to the processor for further computation. To reduce the degrees of freedom in the design process, Zuse settled on the four directions of movement mentioned before (E, S, W, N) and introduced the "common cycle," a clock cycle subdivided into four subcycles. During subcycle I, for example, a pulse in the East direction is transmitted from the clock unit to the machine. During subcycles II, III, and IV, the movements transmitted go in the South, West, and North directions, respectively. In a mechanical relay, the control plate can then be activated in subcycle I, and the actuator plate transmits its motion in subcycle II. In subcycle III, the control plate returns to its original position, and in subcycle IV, the actuator and actuated plates can also return to their original positions (Fig. 3.13).

The clock cycle in the Z1 is provided by a crank that can be turned manually or by an electric motor (Fig. 3.14). This means that the effective cycle time of the Z1

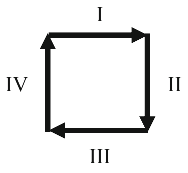


**Fig. 3.13** Zuse's "Einheitskreislauf" (common cycle)

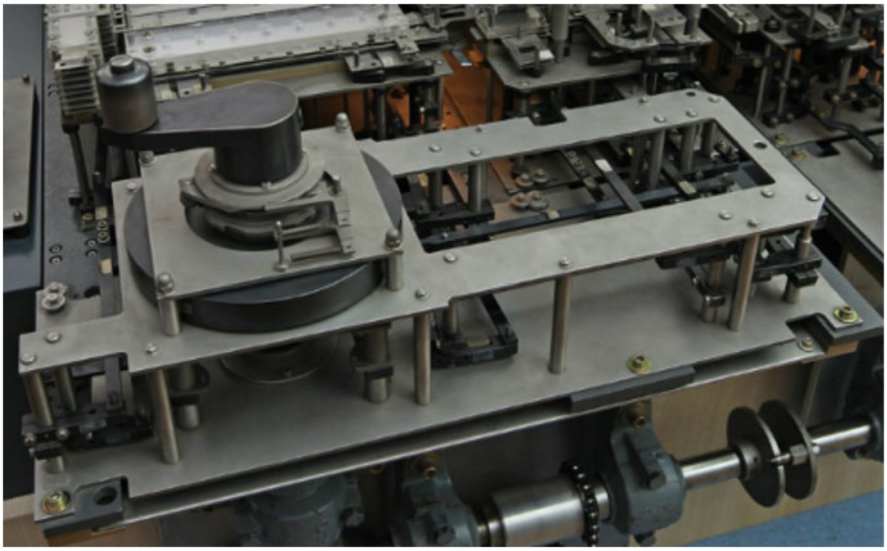


**Fig. 3.14** The crank for producing the common cycle. Note the levers used for transmitting the four directions of movement, as well as energy, all across the machine (http://zuse.zib.de)

was "user-dependent". It could be any value below the maximum permitted speed so that the components would not be subjected to excessive mechanical stress.

In a complex circuit, the actuated plate could act as the control plate for another circuit. If the control plate for the first circuit was set in subcycle I, the first circuit could be actuated in subcycle II, and the result could be used as the control bit for a subcircuit activated in subcycle III and so on. With this arrangement, the maximum depth of a circuit that started and finished its computation within one clock cycle was depth three. Fortunately, the maximum depth of a circuit for computing the addition of binary numbers is precisely three.

### 3.4.2 Transmission of Impulses

Zuse built the reconstructed Z1 in such a way that all the clock subcycles are transmitted by rods and levers located in the "basement" of the machine. The logic is located in the upper part of the Z1, distributed over several layers, like a sandwich of logic elements (Fig. 3.15).

Whenever the state bits are represented by movement in four possible directions, it is necessary to provide a means to change the direction of movement. This can be done using levers, as shown in Fig. 3.16

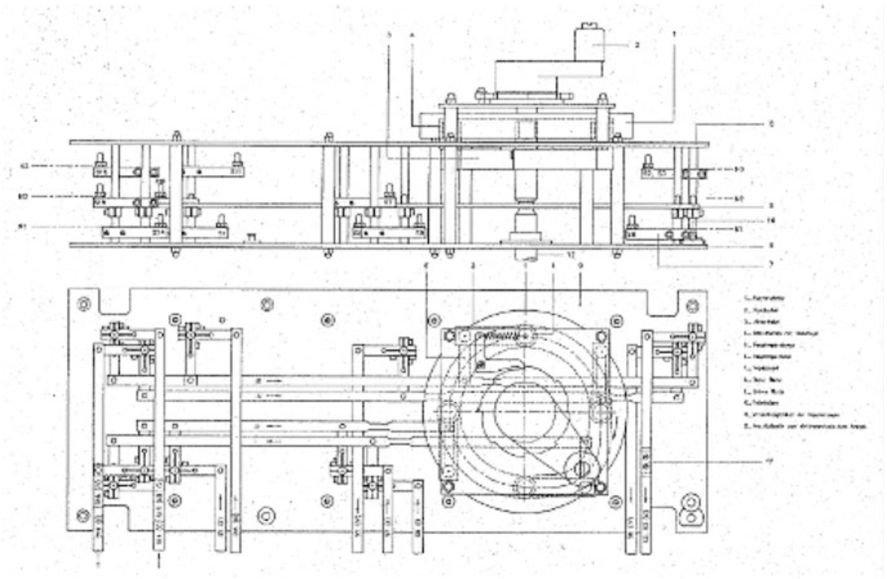


**Fig. 3.15** The diagram of the common cycle mechanism (Schweier and Saupe 1988). The four subcycles are sent in one parallel direction but can be turned 90° using levers

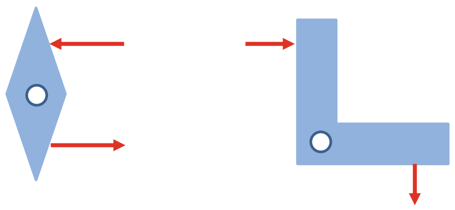


**Fig. 3.16** Mechanical movement: direction changers

It is also possible to have something like an electrical "rectifier" that only allows logic movement to flow in one direction. This is done by using mechanical gates where the actuator plates only push the actuated plate or plates, without using a rod going through multiple plates. This would be more in the spirit of the OR gate shown in Fig. 3.7, where either of the control plates A or B, or both, can push the actuated plate. Zuse had a special notation (using an arrow) for such independently "pushed" plates. He called this "rectification," like in electrical circuits when electricity can only flow in one direction.

An important problem Zuse had to deal with is the case where a circuit uses its result as new data. This is the case of the arithmetic unit (ALU), where a partial result has to be fed back to the ALU for further processing, for example, during a multiplication performed by repeated addition. But here we have a contradiction: when a circuit finishes its calculation, it must be brought back to its initial state (by the movements allowed by the common cycle). Partial results must be captured anddelayed until they are needed again in the ALU. To handle this, Zuse designed a "delay line," which is nothing more than a sequence of mechanical relays, positioned one after the other. The result of one relay is used as the control bit for the next relay and so on. The actuator plates are activated by successive subcycles. Any number of subcycles can therefore be inserted between the production of a result and its subsequent use, without having to use a memory cell to hold this value.

### 3.4.3 *An Example: The Mechanical Addition Unit*

Figure 3.17 shows the basic building block for a mechanical adder that works in one cycle (as proposed by Zuse in Zuse 1952a).

The figure shows the addition of the $i$-th column in the bits $a_k a_{k-1} \dots . a_0$ to be added to the sequence of bits $b_k b_{k-1} \dots . b_0$. The bit $a_i$ is set in the subcycle before subcycle I. In this example, subcycle I moves in the North direction. The following subcycles rotate in their direction of movement by 90°. The bit $b_i$ is pulled up (if it is 1) at the same time that subcycle I becomes active. At the end of subcycle I, the first two gates from the top have computed the conjunction and disjunction, respectively, of the two bits. The carry bit of the previous column of bits ($c_{in}$) is pulled to the right at subcycle II. A carry bit for the next column of bits ($c_{out}$) is generated if the two input bits were (1,1), or if they were one of the pairs (0,1) or (1,0), and there is an active carry from the previous column($c_{in}$).

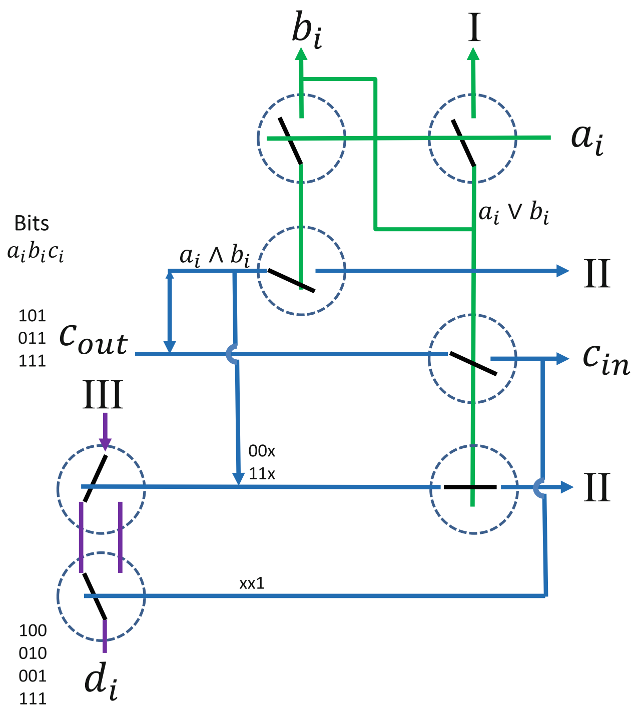


**Fig. 3.17** Circuit for the addition of two bits $a_i$ and $b_i$ with a carry $c_{in}$, and generation of the next carry bit $c_{out}$

The circuit of two gates activated by subcycle III computes the addition bit. If the input was (1,1) or (0,0), the carry bit from the previous column has to be one, to have $d_i = 1$. If the input was (0,1) or (1,0), the carry bit from the previous column has to be zero to have $d_i = 1$. I all other cases $d_i$ is equal to zero.

## 3.4.4 Preliminary Summary of the Mechanical Principles

Before we take a closer look at the Z1, let us summarize what we have learned in the previous sections:

- The state zero of a bit corresponds to non-movement at the initial position.
- The state one of a bit corresponds to a movement step along one of four possible orthogonal directions.
- In a mechanical relay, the actuator plate can either push or pull the actuated plate.
- Multiple actuator plates can push on the same actuated plate (rectification).

- The actuated plate of a mechanical relay can be the control plate of another mechanical relay.
- Mechanical relays can be combined to produce all logic functions.
- To simplify the circuits, a “common cycle” consisting of four orthogonal movements, called I, II, III, and IV, is used. The subcycles allow the machine to synchronize its computations.
- If a mechanical relay activates the control plate at subcycle I, II, III, or IV, it returns the control plate to the original position at subcycle III, IV, I, or II, respectively.
- If the actuated plate is moved in subcycle I, II, III, or IV, it returns to its original position in subcycle III, IV, I, or II, respectively.
- Movement along any of the four orthogonal directions can be reversed or rotated by 90°, clockwise or counterclockwise, using levers.
- A mechanical relay can be used as a delay element (since a mechanical relay only copies a bit after one subcycle).

Based on the principles listed above, Zuse’s mechanical diagrams read very much like electrical circuits and can also be used to build machines based on telephone relays (as Zuse later did without having to modify his notation).

## 3.5 The Memory of the Z1

Until now, the memory of the Z1 was the best-understood part of the Z1 (Zuse 1955). It was described by Schweier and Saupe (Schweier and Saupe 1988) in the 1980s. A similar type of memory was used for the Z4, which had a processor built with telephone relays, but the memory was mechanical, just like in the Z1. The mechanical memory of the Z4 is now housed in Deutsches Museum in Munich. Its operation was simulated on a computer by a student assistant.

The main concept used in the Z1 was that a bit could be stored using a vertical pin that could be set in one of two possible positions (Zuse 1954). One position represents 0, the other position represents 1. The diagram in Fig. 3.18 shows how a memory bit can be overwritten.

Figure 3.18 shows the sequence of plates’ movements required to write a 1 into a memory cell. From left to right, we first see the pin located at the 0 position. In the second image, the control plate moves up. In the third step, the horizontal actuated plate is moved from right to left, and this pushes the pin to the position 1. In the last step, the control plate moves down, and the horizontal plate returns to its original position. The overall effect is to move the pin from position 0 to position 1. From the diagram, it is not difficult to see how a zero can be stored: instead of moving the horizontal plate from right to left, we would move it from left to right. Reading a bit required a different combination of plate movements.

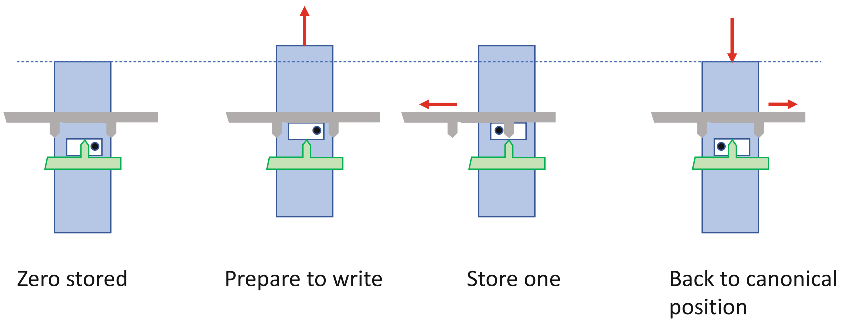


**Fig. 3.18** One mechanical bit in the memory. The pin can be stored in the 0 or 1 position. Its position can be read

Memory words were addressed by decoding the six bits used for an address. Three bits selected one of eight layers; the other three selected one of eight memory words. The decoding circuit for each layer was a classic binary tree of relays with three levels, as used in the Z3 (with a different number of levels). We do not delve further into the structure of the mechanical memory. The details can be consulted in Schweier and Saupe (1988).

## 3.6 The Addition Unit of the Z1

The addition unit of the reconstructed Z1 differs from the type of addition unit described by Konrad Zuse in a document finished after the war. In this document (Zuse 1952a), the binary digits are handled using OR, AND, and identity (NOT-XOR) logic gates. In the Z1 reconstruction, the addition unit uses two XORs and one AND computation.

The first two computations performed during an addition are: (a) the bitwise XOR of the two registers to be added, storing the result, (b) the bitwise AND of the two registers to be added, storing the result. The third step is the computation of the carry bits using the information from the AND and the XOR operations. Once the carry bits have been set, the final step is to compute a bitwise XOR of the carry bits with the result of the first bitwise XOR. The following example shows how to add two binary numbers using the steps described above.

| | | | | | |
|---|---|---|---|---|---|
| Number 1 | 1 | 0 | 1 | 1 | 1 |
| Number 2 | 0 | 0 | 0 | 0 | 1 |
| XOR | 1 | 0 | 1 | 1 | 0 |
| AND | 0 | 0 | 0 | 0 | 1 |
| Carries | 0 | 1 | 1 | 1 | 0 |
| Result (XOR) | 1 | 1 | 0 | 0 | 0 |

Zuse used "anticipating carriage" in all his machines. Instead of propagating a carry through the different binary powers sequentially, the carry can be set for all positions in one step. The example above illustrates the procedure. The first XOR is the partial result of the sum of the two registers without considering the carries. The AND computes the generation of carry bits: they are transported to the next bit to the left, but are further transported to the next binary position as long as there is a one in the result of the previous XOR computation. In the example, the first carry computed with the AND is transformed into three carries, which are finally XORed with the result of the first XOR. A sequence of consecutive 1's from the XOR operation acts as a kind of conveyor belt for transporting AND-generated carries until the chain of 1's breaks.

The circuit shown in Fig. 3.19 is the mechanical addition arrangement used in the reconstructed Z1. The diagram shows the addition of two bits stored in the a and b rods (a could be the $i$-th bit of register Aa, and b the corresponding bit of register Ab). The XOR and the AND computations are performed in parallel using the binary gates 1, 2, 3, and 4. The AND operates on gate 5, generating the carry bit $u_i + 1$, while the XOR closes or leaves open the "chain" of XOR bits using gate 6. Gate 7 is an auxiliary gate for passing the XOR result to the upper level. Gates 8 and 9 compute the final XOR to complete the addition. The movements of the different components are indicated by the arrows. All four cycle directions are used, that is, an addition takes one full cycle, from loading the operands to producing the result. The result is passed to rod e, the $i$-th bit of the register Ae.

This addition circuit is located in layers 1, 2, and 3 of the addition block (see Fig. 3.19). It is remarkable that Konrad Zuse, who had no formal training in binary logic, worked with anticipating carriage. The ENIAC, the first large-scale electronic computer, propagated the carry sequentially from one decimal position in an accumulator to the next. The Harvard Mark I used anticipating decimal carriage.

## 3.7 The Sequencer of the Z1

Each operation in the Z1 is broken down into a sequence of microinstructions. This is done by a kind of table of "criteria" consisting of 108 metallic decoding plates. Each plate is a decoder for a specific value of ten bits. When the specific 10-bit

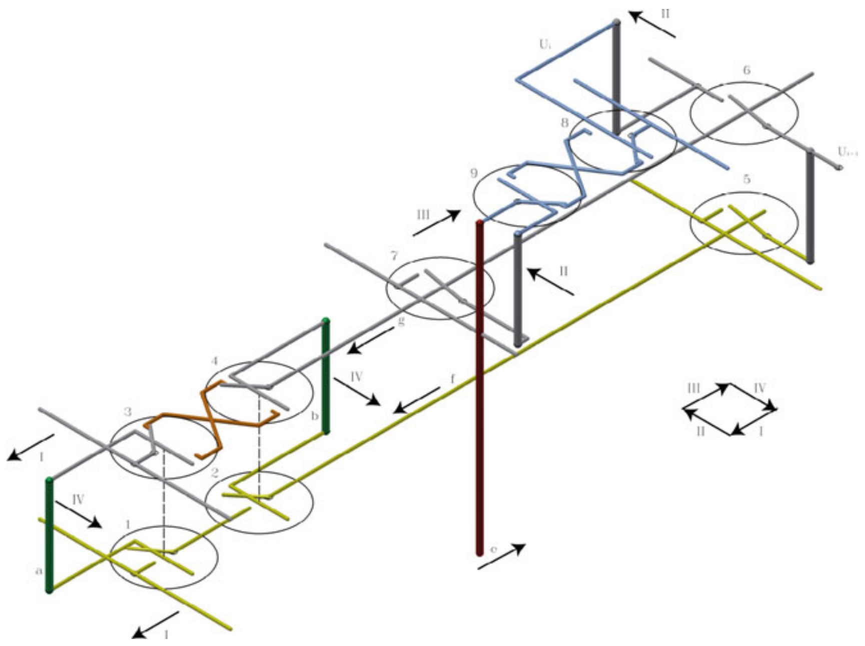


**Fig. 3.19** Addition unit of the Z3. Computation runs from left to right. Bitwise AND and XOR are computed first (gates 1, 2, 3, 4). The carry bits are computed in engagement II (gates 5 and 6). In engagement III, an XOR finishes the computation of the addition (gates 8 and 9)

sequence for the decoder plate is selected, the plate snaps into place and presses on four bits (A, B, C, D) that activate the levers for the required microoperation in the processor. The plates are arranged in a vertical stack across twelve levels of the Z1 so that when a decoder plate is activated, it selects both the microoperation and the layer of the machine in which it will be processed.

Figure 3.20 shows a decoder plate before it snaps in and selects a microoperation (upper diagram), and once it has been activated (lower diagram). The 10 bits to be decoded are labeled in Fig. 3.21:

- The Op0, Op1, and Op2 bits contain the binary opcode of the instruction
- The bits S0 and S1 are condition bits set by other parts of the machine. When S0=1, for example, an addition is transformed into a subtraction.
- The bits Ph0, Ph1, Ph2, Ph3, and Ph4 are used to count the number of microcycles (or "phases") in an instruction. Multiplication, for example, is executed in 20 phases and the five bits Ph0 to Ph4 advance from 0 to 19 during the operation.

These 10 bits specify the operation that is active (with the opcode), if there are special conditions (with the bits S0 and S1), as well as the current "phase", i.e., the microinstruction to be executed next. The 10 bits theoretically allow us to define up to 1024 different conditions or cases. An instruction can consist of up to 32 phases.

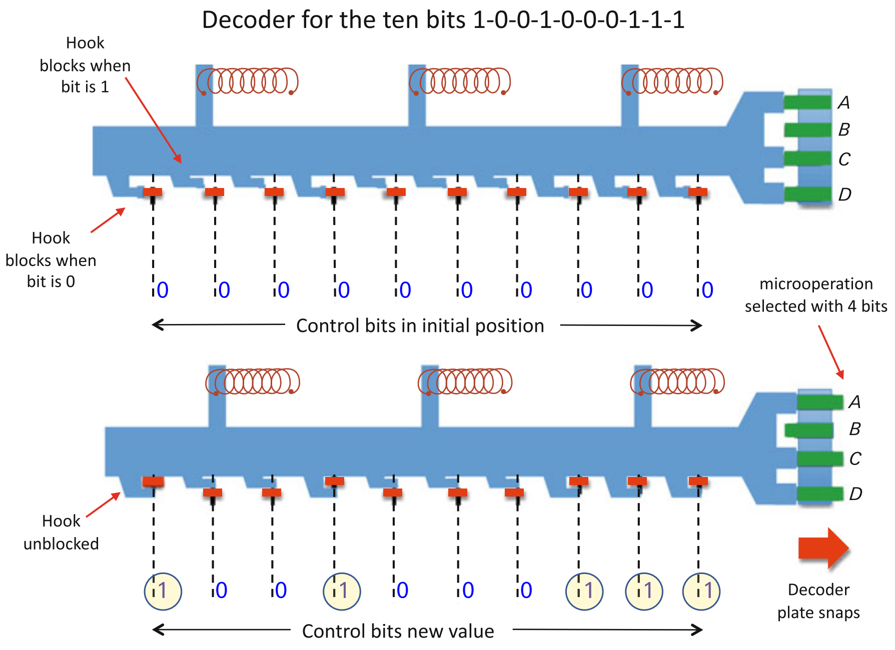

Decoder for the ten bits 1-0-0-1-0-0-0-1-1-1
Hook blocks when bit is 1
Hook blocks when bit is 0
0 0 0 0 0 0 0 0 0 0
A
B
C
D
microoperation selected with 4 bits
Control bits in initial position
Hook unblocked
1 0 0 1 0 0 0 1 1 1
Control bits new value
Decoder plate snaps

**Fig. 3.20** A decoder plate being blocked by the current value of 10 bits (upper image) or after it snaps to the right due to the value of the 10 bits (lower image)

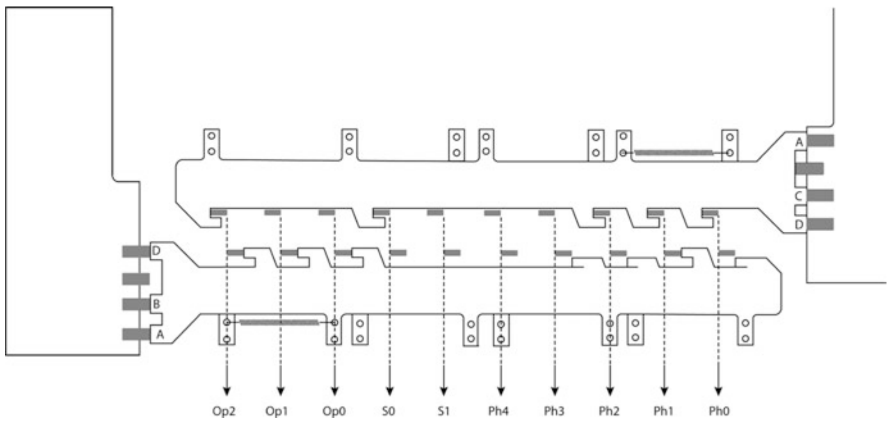


**Fig. 3.21** Two control plates. The upper one has been activated by the value of the 10 observed bits. The lower one is blocked. The upper decoder selects the microoperation for the mantissa's ALU, the lower one for the exponent's ALU

In Fig. 3.20, we can see one decoder plate. Springs pull it to the right. The decoder plate cannot move (upper image) because it is blocked by the current value of the 10 bits listed above. The value of the 10 bits is represented by 10 rods, shown here as 10 small red rectangles. The rectangles can move up (to the 1 position) or stay where they are (0 position). Note that there are two types of hooks in the lower part of the decoder plate. The longer hooks are blocked when the corresponding bit is 0. The shorter hooks can block the plate when the corresponding bit is 1.

The lower diagram in Fig. 3.20 now shows what happens when the values of the 10 bits are exactly those for which the plate was designed. Five bits are now ones, and their corresponding small rectangular rods have moved up. The long hooks are no longer blocked. The small hooks are not blocking either, given that their associated bits have the value zero. Now the decoder plate snaps to the right and selects a four-bit encoding of the required microoperation. Since this happens in a vertical stack of decoder plates, both the microoperation and the required layer of the Z1 get selected.

Zuse always put two decoder plates together at the same level in the vertical stack of plates. One of the plates could snap to the left to select the microoperation for the exponent ALU, and another could snap to the right to select the microoperation for the mantissa ALU (Fig. 3.21).

Controlling the Z1 thus amounts to adjusting the teeth of the metal plates so that each one of them responds to a specific ten-bit combination, to act on the left or right components. The left side controls the exponent's half of the processor while the right side controls the mantissa's half. The four bits A, B, C, or D are selected by a microcontrol plate (by not pressing on them). Figure 3.22 shows a top view of the stack of criteria plates.

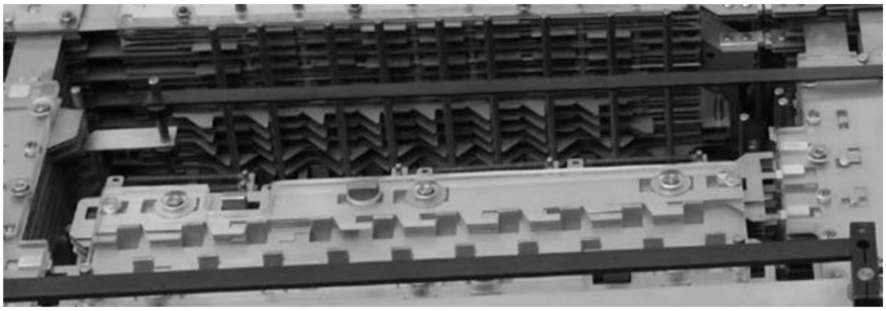


**Fig. 3.22** Photograph of the Z1 decoding unit. We can see one decoder on top. Up to 108 decoder plates were stacked vertically, in pairs (http://zuse.zib.de)

## 3.8 The Processor's Datapath

Figure 3.23 shows the floating-point processor of the Z1. The processor has one datapath for handling the exponents (left side) and one for handling the mantissas (right side). The floating-point registers F and G consist of 7 bits for the exponent and 20 bits for the mantissa (expanded from 16 in memory). The exponent-mantissa pair (Af,Bf) is called floating-point register F, and the pair (Ag,Bg) is called floating-point register G. The signs of the arguments are handled externally in a sign unit. The sign of a product or division is computed in advance. The sign of an addition or subtraction is set after the operation takes place.

In Fig. 3.23, we can see the registers F and G and their connections to the rest of the processor. The combined FP ALU (Arithmetic Logic Unit) contains two FP registers: the pair (Aa,Ba) and the pair (Ab,Bb). These registers are the direct inputs to the ALUs. They must be loaded and may retain partial results during several iterations due to the feedback bus from the ALU-outputs Ae and Be.

In the Z1, the data buses are used in "open collector" mode, that is, many inputs can push on the same data line (which is a mechanical component). There is no need to "electrically" isolate the data lines from the inputs, since no electricity is involved.

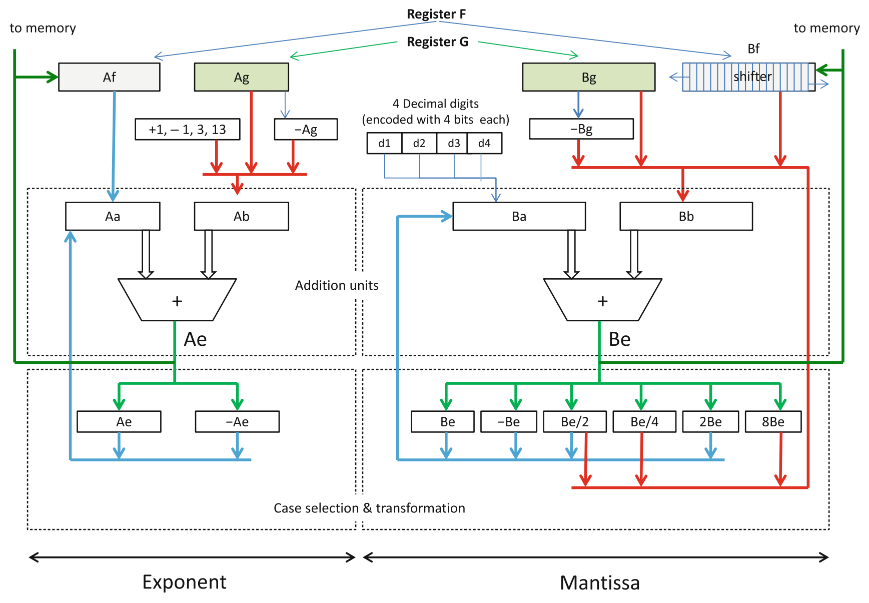


**Fig. 3.23** The processor datapath in the Z1. The left part corresponds to the exponent's ALU and registers and the right side to the mantissa's. The results Ae and Be can be fed back to the temporary registers or be negated or shifted. The four bits representing a decimal digit are copied to register Ba directly, one digit after another, using four bits. The decimal-binary conversion operates on this data

Since a zero input represents no movement of a mechanical part (no pushing), while a 1 represents a movement (pushing), there is no conflict between the parts. If two parts push on the same data line, the only important thing is that they act in step with the machine cycle (pushing only works in one direction).

The only registers visible to the programmer are (Af,Bf) and (Ag,Bg). They have no address; the first register loaded by a Load operation is (Af,Bf), the second register loaded afterward is (Ag,Bg). Once two registers have been loaded, the arithmetic operations can be started. (Af,Bf) is also the result register for arithmetic operations. The second register can be loaded after an arithmetic operation and be the second argument for a new arithmetic operation. This scheme of register usage is similar to that of the Z3, but the coordination between main and auxiliary registers is simpler in the Z1. The unit Bf can shift the mantissa it contains one place to the right or left. This capability is used for the multiplication and division algorithms, as explained in the sections below.

As can be seen from the processor datapath, the individual registers Aa, Ab, Ba, and Bb can be loaded with different kinds of data: values from other registers, constants (+1, +1, 3 and 13), negative values from other registers, and the values coming back from the ALUs. The ALU outputs can be negated or shifted. A shift of $n$ places to the left is represented by a box containing a multiplication with $2^n$; a shift of $n$ places to the right is represented by a division with $2^n$. These boxes are mechanical circuits producing the appropriate bit-shifts or bit-complements. The result of the addition of the registers Ba and Bb, for example, is stored in Be and can be transformed in several ways: the result Be can be negated (−Be), shifted one or two places to the right (Be/2, Be/4), or shifted one or three places to the left (2Be, 8Be). Each of these computations is performed in a different layer of the mechanical stack of layers that make up the ALU. The appropriate result, depending on the active computation, is returned to register Ba or Bb. Case selection is done by levers that activate the appropriate layer as selected by the microcontroller. The result Be can also go straight to the memory unit (the corresponding bus line is not shown in Fig. 3.23).

The ALU performs an addition in each cycle. All registers Aa, Ab, Ba, and Bb are cleared after an ALU computation and can be reloaded with the feedback values.

Register Ba has a special use for the conversion of four decimal digits to binary. Each decimal digit entered through the mechanical panel is transformed into four bits. Groups of four bits are fed directly into register Ba (at position $2^{-13}$), which can advance the four bits by performing a multiplication by a factor of 10, then adding the next digit to the partial result, multiplying again by 10, and so on. For example, to transform the number 8743 from decimal to binary, the digit 8 is entered first and multiplied by 10. Then 7 is added to the result, and the new sum (87) is multiplied by 10. The result (870) is then added to 4 and so on. This yields a simple algorithm for the conversion of decimal input to a binary number. During this process, the exponent half of the processor adjusts the exponent of the final floating-point result (therefore, the constant 13 in the exponent ALU, which corresponds to $2^{+13}$ (see decimal-binary conversion algorithm further down).

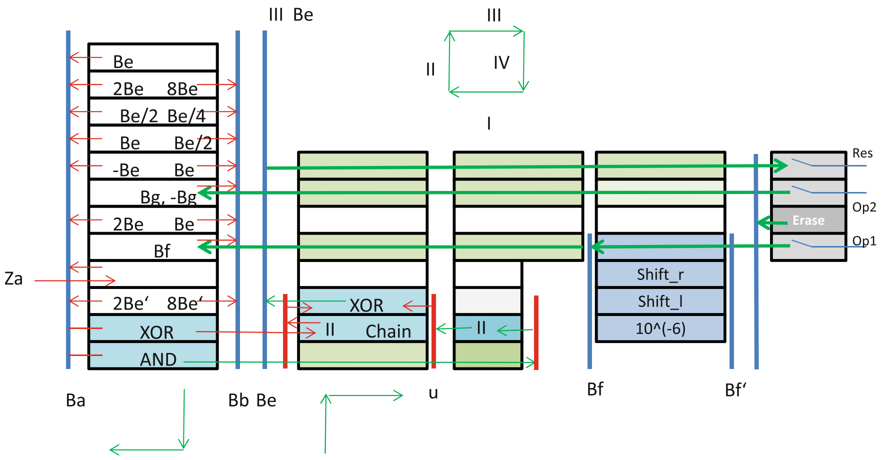


**Fig. 3.24** The layered spatial distribution of operations in the processor. The shifters for Be are on the left stack. The addition unit is distributed between the three leftmost stacks. The shifters for Bf are in the right stack as is the binary equivalent of $10^{-6}$. The result goes to memory through the line labeled Res on the right. The two registers Bf and Bg arrive from memory as first (Op1) or second operand (Op2)

Figure 3.24 shows the spatial distribution of the different elements of the processor datapath for the mantissa part. All the shifters have been allocated in different layers of the twelve constituting the leftmost module of the machine. The registers Bf and Bg come from the right side, directly from memory (layers 5 and 7). The result Be is fed back to memory, passing through level 8. The bits of the registers Ba, Bb, and Be are stored in vertical rods (only one bit is shown in this cross-section of the processor). The ALU is distributed across two stacks of mechanical layers. Levels 1 and 2 compute the AND and XOR of each bit in Ba and Bb. The results are passed to the right, where the carry bits and the final XOR are computed and stored in Be. The result Be can go back to be stored in memory or can be shifted in all the different ways shown, before being fed back to Ba or Bb, as desired. Some circuits seem redundant (there are two ways of loading Be into Ba, for example), but they represent alternatives. Level 12 loads Be into Ba unconditionally, level 9 only if the exponent Ae is zero. The green boxes in the diagram are empty layers where no computation takes place and mechanical components can pass through. The box around the bars Bf and Bf' contains the shifter for Bf needed for multiplication (where the bits of Bf are read one by one, starting with the lowest binary power).

Now you can imagine the computational flow in this machine: data flow from the registers F and G into the machine, filling the A and B register pairs. A single addition or sequences of additions/subtractions (for multiplication or division) are performed. Partial results are recycled into the A and B registers until the result is complete. The final result is then loaded into register F, and a new computation can be started (Fig. 3.25).

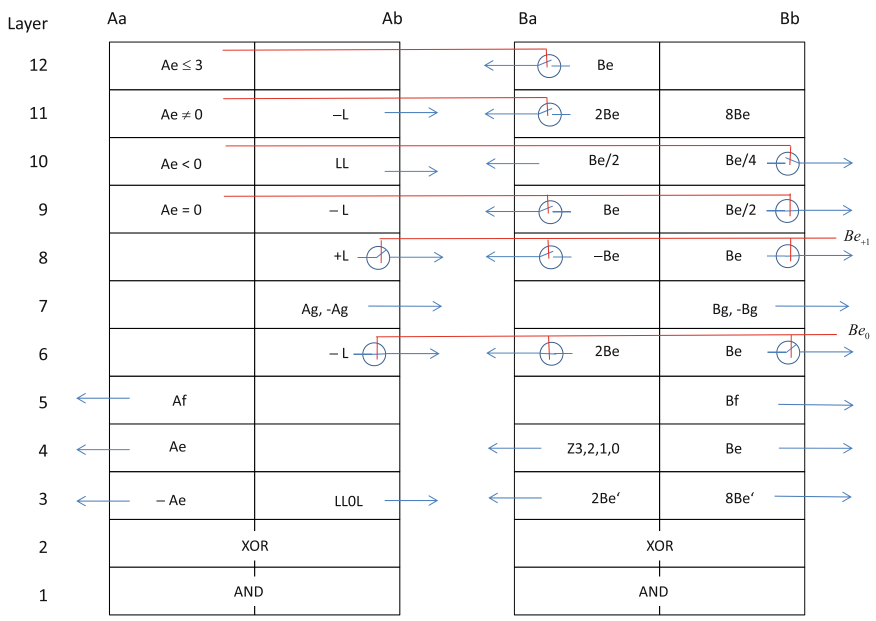


**Fig. 3.25** Communication between the exponent and the mantissa ALUs

## 3.9 Conclusions

The original Z1 was destroyed during an Allied bombing raid flown in December 1943 over Berlin. It is impossible to decide today if the original Z1 was identical to the reconstructed Z1. The few surviving photographs show that the original was bulkier and had a less regular shape. We can only take Zuse's word for it. However, I believe that he had no real reason to consciously "embellish" the original machine through the reconstruction. Memory can be a tricky fellow, though. The few notes Zuse scribbled between 1935 and 1938 seem to be consistent with the later reconstruction. The Z3 was finished in 1941, and according to Zuse, the logic design was very similar (Fig. 3.26).

Siemens (the company that acquired Zuse's firm), together with the companies Nixdorf, AEG, DATEV, and Krupp Atlas, financed the reconstruction of the Z1 in the 1980s. Zuse did all the construction work at his home, with the help of two students. When the Z1 was finished, part of the wall on the upper floor of Zuse's house had to be removed so that a large crane could lift the machine for transport to Berlin.

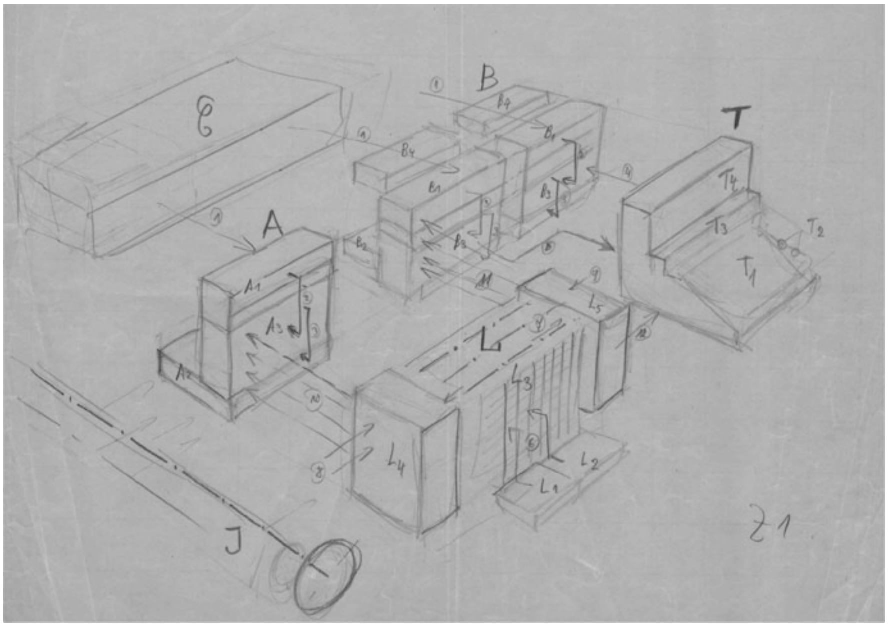


**Fig. 3.26** Sketch of one of Zuse's early designs for a Z1 reconstruction. Undated (http://zuse.zib.de)

The reconstructed Z1 is a very elegant computer, consisting of thousands of components but not one too many. It would have been possible to use only two shifters at the output of the mantissa ALU (a shift of one bit to the left, and one bit to the right), but the choice of shifters made by Zuse speeds up the basic arithmetic operations significantly, at a low cost in components. I find the processor of the Z1 rather more elegant than the processor of the Z3 because it is more compact and "fundamental." It is as if when Zuse moved to telephone relays, the simpler and more reliable components allowed him to be "profligate" with the size of the CPU. The same thing happened when the Z4 was finished. The Z4 was just a bigger Z3, but the computer architecture was roughly the same, although the Z4 had more instructions. The mechanical Z1 never worked consistently, and Zuse himself later called the mechanical realization "a dead end." He used to joke that the 1989 reconstruction of the Z1 was quite accurate, because the original was not reliable, and neither was the reconstruction. Oddly enough, the mechanical memory design was reliable enough to be reused for the Z4, as a way to save telephone relays. The mechanical memory of the Z4 was operational from 1950 to 1955 in Switzerland, where the machine was installed at the ETH Zurich (Bruderer 2012). As late as December 1944, Zuse wrote the following about mechanical components: "According to the present stage of development, it can be safely said that the future belongs to the mechanical gates, because of their mentioned advantages, and they must be preferred also because of the present war conditions. A series of 12 complete mechanical devices (i.e. Z4s) could be built with 100,000 to 150,000 man-hours. Their computing power could replace up to one million man-hours per year" (Zuse 1944).

What I find most surprising is how the young Konrad Zuse could come up with such an elegant design for a computing engine. He even calculated the numerical accuracy that his machine would have (Zuse 1936c). While the ENIAC or Mark I teams in the USA consisted of seasoned scientists and electronics experts, Zuse was working in isolation and without previous experience. However, from an architectural point of view, we compute today as Zuse did in 1938, not as the ENIAC did in 1945 (Goldstine and Goldstine 1996). More elegant architectures appeared later with the EDVAC report and the bit-serial machines developed by von Neumann and Turing.

## Appendix: The Arithmetic Instructions

As explained above, the Z1 could perform the four basic arithmetic operations. In the tables discussed below, the convention has been used of representing a binary one with the letter "L." The tables show the sequence of microinstructions required for each operation and how this affects the data flow between the registers in the processor. One table summarizes addition and subtraction (using two's complement), one table summarizes multiplication, and one table is for division. There is also one table for each of the two I/O operations: decimal-binary and binary-decimal conversion. The tables are divided into Part A for the exponent, and Part B for the mantissa. The registers Aa, Ab, Ba, and Bb are loaded as shown in each row of a table. The phase of the operation is given by the column labeled "Ph." Conditions can trigger an operation or inhibit it from starting. When a row is executed, condition bits can be set, or the next phase (Ph) can be computed by the incrementer.

### *Addition/Subtraction*

The table of microinstructions (Fig. 3.27) covers both addition of numbers and subtraction. The main problem for both operations is to scale the two numbers to be added/subtracted so that the binary exponent is the same. Assume that the numbers $m_1 \times 2^a$ and $m_2 \times 2^b$ are to be added. If $a = b$ the two mantissas can be added immediately. If $a > b$ then the smaller number is rewritten as $m_2 \times 2^{b-a} \times 2^a$.

| Part A (Exponent) | | | | | | | | | | | Notification->B | Part B (Mantissa) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Block 17 (left Part) | | | | | | | Block 15a | | | | | Block 17 (right part) | | | | | Block 15b | | |
| Operating sequence | | Criterion | | | | to operate | Condition | to operate | Aa | Ab | | Criterion | | | | to operate | Condition | Ba | Bb |
| Symb | Description | No. | S0 | S1 | Ph | | | | | | | No. | S0 | S1 | Ph | | | | |
| ADDITION (Σ) | Difference of the Exponents Δα | 1 | | | 0 | Ph | | | Af | | | | | | | | | | |
| | | 1 | | | 0 | | | | | -Ag | | | | | | | | | |
| | S1 ? | 2 | | | 1 | Ph, (Ae) | Ae ≥ 0 | S1 | Ae | | | | | | | | | | |
| | \|Δα\| | 3 | | 0 | 2 | Ph | | | -Ae | | | 3 | | 0 | 2 | | | | Bf |
| | | 4 | | L | 2 | (Ae) | | | Ae | | | 4 | | L | 2 | Ph | | | Bg |
| | Align → | 5 | | | 3 | Ae | | | | | | | | | | | | | |
| | | | | | | | → Ae = 0 | Ph | | | →B | (5) | | | 3 | A → | (Ae = 0) | Be | |
| | | | | | | | → Ae ≠ 0 | | | -1 | →B | (5) | | | 3 | A → | (Ae ≠ 0) | | ½ Be |
| | Actual Addition (Σ) | | | | | | | | | | | 6 | 0 | | 4 | | | -Be | |
| | | | | | | | | | | | | 7 | L | | 4 | | | Be | |
| | | 8 | | L | 4 | Ph | | | Af | | | 8 | | L | 4 | | | | Bf |
| | | 9 | | 0 | 4 | Ph | | | | Ag | | 9 | | 0 | 4 | (Ph) | | | Bg |
| | Align → | 10 | L | | 5 | Lz, Ae | | | Ae | | | 10 | L | | 5 | | $Be_{+1} = 0$ | Be | |
| | | 10 | L | | 5 | | $Be_{+1} = L$ | | Ae | +1 | | 10 | L | | 5 | | $Be_{+1} = L$ | | ½ Be |
| | Complement by real Substraction | 11 | 0 | | 5 | Ph, Ae | $Be_{+1} = L$ | S3 | Ae | | | 11 | 0 | | 5 | | $Be_{+1} = L$ | -Be | |
| | | | | | | | | | Ae | | | 11 | 0 | | 5 | | $Be_{+1} = 0$ | | Be |
| | Alignment by real Substraction | 12 | 0 | | 6 | Ae | → $Be_0 = 0$ | | Ae | -1 | | 12 | 0 | | 6 | | $Be_0 = 0$ | 2Be | |
| | | | | | | | → $Be_0 = L$ | Lz | Ae | | | | | | | | $Be_0 = L$ | | Be |

**Fig. 3.27** The microinstructions for addition and subtraction. An addition is finished in 5 cycles, a subtraction in 6

The first multiplication is equivalent to shifting the mantissa $m_2$ by $(a - b)$ places to the right (making the mantissa smaller). Let us call $m_2' = m_2 \times 2^{b-a}$. The two mantissas to be added are now $m_1$ and $m_2'$. The common binary exponent is $2^a$. A similar procedure is used when $a < b$.

Once the numbers have been aligned, condition bit S0 is tested (cycle 4). If S0 is 1, the mantissas are added. If S0 is 0, the mantissas are subtracted in that cycle.

In the table (Fig. 3.27), the maximum binary exponent of the two numbers is found first, and the mantissa of the smaller number is shifted to the right as many places as necessary, until the two binary exponents are equal. The actual addition starts in cycle 4 and is performed by the ALU in just one cycle. In cycle 5, it is tested to check if the new result mantissa is normalized, and if not, it is shifted to normalize it. It can happen (after a subtraction) that the result mantissa is negative, in which case the result is negated to make it positive. This change of sign is stored in the condition bit S3 to make the necessary adjustment to the sign of the final result. At the end, the result is normalized.

The sign unit near the tape reader (see Fig. 3.5, Block 16) computes the sign of the result and the type of operation in advance. If we assume that the mantissas $x$ and $y$ are positive, then we have the following four cases for addition and subtraction (after having assigned the sign of each number). We call the result $z$:

(a) $z = +x + y$
(b) $z = +x - y$
(c) $z = -x + y$
(d) $z = -x - y$

Cases (a) and (d) can be handled with an addition in the ALU. In case (a), the result will be positive. In case (d), it will be negative. Cases (b) and (c) require a subtraction. The sign of the subtraction is computed in phase 5 (Fig. 3.27).

An addition runs in the following steps:

- Determine the absolute difference of exponents $D$ in the exponent unit
- Select the largest exponent
- Shift the mantissa of the smaller number $D$ times to the right
- Add the mantissas
- Normalize the result
- The sign of the result is the sign of both arguments.

A subtraction runs in the following steps:

- Determine the absolute difference of exponents $D$ in the exponent unit
- Select the largest exponent
- Shift the mantissa of the smaller number D times to the right
- Subtract the mantissas
- Normalize the result
- The sign of the result is the sign of the largest number (in absolute value).

The final sign of the result is determined through the sign unit, which has a preliminary sign for the operation.

The following table may seem redundant, but it is a symbolic translation of the table in Fig. 3.27. It is useful because it can show the reader how to interpret the algorithmic tables for the arithmetic operations in the Z1.

| Phase | Operations in the ALUs | Comments |
|---|---|---|
| 0 | Ae=Af−Ag | Difference of exponents |
| 1 | IF(Ae $\geq$ 0) S1=1, Aa=Ae | Register F has larger number, recycle Ae |
| 2 | IF(S1=0) Aa=−Ae | Make Ae positive |
|  | ELSE Aa=Ae | Ae is positive |
|  | IF (S1=0) Bb=Bg ELSE Bb=Bf | Put smaller number's mantissa in Bb |
| 3 | WHILE (Ae$\neq$0) |  |
|  | Ba=Be/2; Ae=Ae−1 | Shift right smaller mantissa by exponent difference |
|  | ELSE Ba=Be |  |
| 4 | IF (S1=1) | Select exponent and mantissa of largest number |
|  | Aa=Af; Bb=Bf |  |
|  | ELSE |  |
|  | Aa=Ag; Bb=Bg |  |
| 5 | IF (S0=1) | It is addition |
|  | IF (Ae<2) Ba=Be | If result $<$ 2, leave unchanged |
|  | ELSE Bb=Be/2, Ae=Ae+1 | If result $\geq$ 2, shift right, increase exponent) |
|  | ELSE | it is subtraction |
|  | IF (diff. negative) Ba=−Be; S3=1 | If difference negative, make diff. positive, set flag |
|  | ELSE Bb=Be |  |
| 6 | WHILE (Be$\neq$ 0) Ba=2Be; Ae=Ae−1 | Shift left until leading bit is 1, decrease exponent |
|  |  | final result in (Ae,Be), set sign |

The flag S3 is set when the difference of the two numbers is negative, after a subtraction. This flag is used by the sign unit so that the final sign of the number can be determined. This case can happen when the exponents of the two numbers are equal, but the mantissas are different.

## *The Problem of Zero*

Zuse used normalized floating-point numbers, i.e., a representation where the leading bit of the mantissa, before the point, is 1. A binary normalized mantissa can be something like 1.001, but not 10.01, nor 0.1001. This is a simplification that is useful for the numerical algorithms and saves one bit of memory since the leading bit of the mantissa does not have to be stored (it is always 1). However, the number zero cannot be represented in the computer if the mantissa must be normalized! The solution adopted by Zuse in the Z3 was that the lowest possible exponent (−64) represented zero, regardless of the mantissa. Before an arithmetic operation is started, the computer has to check if zero is involved. If so, the results are trivial, and the ALU does not have to go through the entire numerical algorithm. Multiplication by zero, for example, is zero. Division by zero produces an error, and so on.

The tables in this appendix do not deal with the case where one or both arguments are zero. The implicit assumption is that checking for zero is done before the arithmetic algorithms are started. The Z3 had the necessary logic for this check. The Z1, alas, did not! The reason is that Zuse built the Z1 as a mechanical proof of concept leaving such implementation details for the next machine. He was aware of the necessary logic as his technical descriptions show (Zuse 1938).

It is interesting to know that the Z1 reconstruction of 1989 was accurate in that respect: it could not compute with a zero argument! The tables do not check for intermediate results that could be zero (for example, after a subtraction either). This would have been handled by a runtime exception, as in the Z3.

## Multiplication

For multiplication, first, the exponents of the two numbers are added in cycle 0 (criterion 21, exponent part, Fig. 3.28). Then 17 cycles are used to examine every bit of the mantissa in Bf, starting from the lowest power, all the way to the highest binary power in the mantissa (from the bit at position −16 to position 0). The register

| Part A (Exponent) | | | | | | | | | | Part B (Mantissa) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Block 17 (left Part) | | | | | | | Block 15a | | | Block 17 (right part) | | | | | Block 15b | | | |
| Operating sequence | | Criterion | | | | to operate | Condition | Aa | Ab | Criterion | | | | to operate | Condition | to operate | Ba | Bb |
| Symb | Description | No. | S0 | S1 | Ph | | | | | No. | S0 | S1 | Ph | | | | | |
| Multiplication (×) | Sum of the Exponents Δα | 21 | 0 | 0 | 0 | Ph | | Af | | | | | | | | | | |
| | | 21 | 0 | 0 | 0 | "R4. | | | Ag | | | | | | | | | |
| | | 22 | L 0 | 0 L | 0 | Ph, Ae, b($10^{-6}$) | | | a($10^{-6}$) | | | | | | | | | |
| | actual Multiplication (×) | -24 | | | 1 ⋮ 17 | Ph, "R4, Ae | | Ae | | 24 | | | 1 ⋮ 17 | | | ½ Be | | |
| | | | | | | | | | | 24 ↓ | | | 1 ⋮ 17 | | mm = L | | Bg | |
| | | | | | | | | | | | | | | | | | | |
| | Align → | 26 | | | 18 | Ph, Ae | | Ae | | 26 | | | 18 | | $Be_{+1}$ = 0 | | Be | |
| | | 26 | | | 18 | | $Be_{+1}$ = L | Ae | 1 | 26 | | | 18 | | $Be_{+1}$ = L | | | ½ Be |
| | Finish | 27 | | | 19 | Ae | | Ae | | 27 | | | 19 | | | | Be | |
| | | 28 | | 0 | 19 | Lz, (Ae) | | | | | | | | | | | | |
| | Setting ↘ | | | | | | | | | 29 | | L | 19 | ↘ , Ph1 | | | | |
| | Setting b($10^{-6}$) | | | | | | | | | 30 | L | | 0 ⋮ 5 | u5 | | | | |
| | | | | | | | | | | 31 | | L | 0 ⋮ 5 | d4 | | | | |

**Fig. 3.28** The microinstructions for multiplication. The multiplier-mantissa Bf is stored in a shift register (shift right). The multiplicand-mantissa is stored in register Bg

Bf is shifted to the right once at each step. The bit *mm* in the table contains the previously shifted-out bit at position $-16$. If the shifted-out bit is 1, then Bg is added to the partial result (which has been shifted previously one position to the right); otherwise, zero is added. Therefore, this algorithm computes the result

$$B_e = Bf_0 \times 2^0 \times B_g + Bf_{-1} \times 2^{-1} \times B_g + \cdots + Bf_{-16} \times 2^{-16} \times B_g$$

If the mantissa after the multiplication is greater than or equal to 2, the result is normalized in cycle 18 by shifting it one position to the right. Cycle 19 puts the final result on the data bus.

## *Division*

The division algorithm takes 21 cycles and is based on so-called "non-restoring floating-point division" (Fig. 3.29). The bits of the quotient are calculated one by one, starting with the most significant bit and going down to the less significant bits.

First, the difference of the exponents is computed in cycle 0, and then the division of the mantissas is executed. The divisor mantissa is stored in Register Bg and the dividend mantissa in Register Bf. The remainder is initialized to Bf in cycle 0. In each subsequent cycle, the divisor is subtracted from the remainder. If the result is

| Part A (Exponent) | | | | | | | | Part B (Mantissa) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Block 17 (left Part) | | | | | Block 15a | | | Block 17 (right part) | | Block 15b | | |
| Operating sequence | | Criterion | | to operate | Condition | Aa | Ab | Criterion | | Condition | Ba | Bb |
| Symb | Description | No. | Ph | | | | | No. | Ph | | | |
| DIVISION (··) | Diffrence of the Exponents Δα | 40 | 0 | Ph | | Af | | 40 | 0 | | | Bf |
| | | 40 | 0 | | | | -Ag | | | | | |
| | actual Division | 41 | 1 | | | Ae | | 41 | 1 | | Be | |
| | | | | | | | | | | | | -Bg |
| | | 43 | 1 ⋮ 18 | Ph, Ae, "R5 | | Ae | | 42 | 2 ⋮ 17 | u+2=0 | | Bg |
| | | | | | | Ae | | 42 | 2 ⋮ 17 | u+2=L | | -Bg |
| | | | | | | Ae | | 42 | 2 ⋮ 17 | | 2Be | |
| | Bf → Bb | 44 | 19 | Ph , Ae | | Ae | | 44 | 19 | | | Bf |
| | Align ← | 45 | 20 | | $Be_0 = 0$ | Ae | -1 | 45 | 20 | $Be_0 = 0$ | 2Be | |
| | | 45 | 20 | Lz, Ae | | Ae | | 45 | 20 | $Be_0 = L$ | | Be |

**Fig. 3.29** The microinstructions for division. The dividend-mantissa in Bf is pushed bit by bit in a shift register (shift left). The divisor-mantissa is kept in register Bg

positive, the corresponding bit in the mantissa of the result is set to 1. If the result is negative, the bit in the mantissa of the result is set to 0. The bits of the quotient are computed one by one, from bit 0 to bit $-16$. There is a mechanism in the Z1 to set the bits of register Bf one by one, as needed.

If the remainder becomes negative, there are two possible strategies for continuing. In "restoring division," the divisor D is added back to the remainder (R-D) to reproduce the positive remainder R. Then the remainder is shifted one position to the left (which is equivalent to shifting the divisor to the right) and the algorithm continues. In "non-restoring division," the remainder (R-D) is shifted one position to the left and then the divisor D is added. Since, in the previous step, (R-D) was negative, the shift to the left transforms this quantity into (2R-2D). If we now add the divisor, we obtain (2R-D), which is the subtraction of D from the left-shifted positive R, as we want to have in the next step of the division algorithm. The algorithm can continue in this way until the remainder becomes positive, and then we continue by subtracting the divisor D again. In the table below $u$ refers to the carry bit for the binary power at position 2. If this bit is set, the result of the addition was negative (in two's-complement arithmetic). Non-restoring division is a very elegant way to compute the quotient of two floating-point mantissas, since the restoring step (an additional cycle) is avoided.

Somewhat puzzling is the fact that the Z3 first tested whether the subtraction of Ba and Bb could become negative during a division, in which case the subtraction was "undone" by using a shortcut bus from Ba to Be (eliminating the result of the subtraction). This extra hardware was not used in the reconstructed Z1, and the non-restoring algorithm seems more elegant than the solution used in the Z3.

## *Input and Output*

The input panel consists of four columns of 10 small plates each. In each column (called Za3, Za2, Za1, and Za0, in that order, from left to right) the operator can pull out any of the digits 0 to 9. He or she can thus enter any four-digit decimal number using the four columns. Pulling a digit plate just generates its binary equivalent in the input console (using four bits). The input console is therefore just a 4-by-10 table of the 10 binary equivalents of the digits 0 to 9.

The microcontroller of the Z1 then controls passing each decimal digit Za3, Za2, Za1, and Za0 to the datapath through register Ba (at position $Ba_{-13}$ corresponding to the power $2^{-13}$). Za3 is entered first (in register Ba), and then a multiplication by 10 is performed. Digit Za2 is next, and another multiplication by 10 ensues. This is repeated for all four digits. The binary equivalent of the four decimal digits is contained in the Be after cycle 7. In cycle 8, the mantissa is normalized, if necessary. The constant 13 (LL0L in binary) is added once to the exponent (phase 7) to account for the fact that the digits are entered at the mantissa bit $-13$ (Fig. 3.30).

| Part A (Exponent) | | | | | | | | | Part B (Mantissa) | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Block 17 (left Part) | | | | | | Block 15a | | | Block 17 (right part) | | | | Block 15b | | |
| Operating sequence | | Criterion | | Cond. | to operate | Condition | to operate | Ab | Criterion | | Condition | to operate | Condition | Ba | Bb |
| Symb | Description | No. | Ph | | | | | | No. | Ph | | | | | |
| Transform (decimal → dual) (↘) | Ready? 1. Digit x L0L0 2. Digit x L0L0 3. Digit x L0L0 4. Digit | | | | | | | | 50 | 0 | u8 | Ph | | | |
| | | | | | | | | | 51 | 1 | | u2, Ph | | za3 | Be |
| | | | | | | | | | 52 | 2 | | Ph | | 2Be | 8Be |
| | | | | | | | | | 53 | 3 | | u2, Ph | | za2 | Be |
| | | | | | | | | | 54 | 4 | | Ph | | 2Be | 8Be |
| | | | | | | | | | 55 | 5 | | u2, Ph | | za1 | Be |
| | | | | | | | | | 565 | 6 | | Ph | | 2Be | 8Be |
| | | 57 | 7 | | | | | LL0L | 57 | 7 | | u2, Ph | | za0 | Be |
| | Align ← | 58 | 8 | | Ae | | | | 58 | 8 | | | $Be_0 = 0$ | 2Be | |
| | | 58 | 8 | | | $Be_0 = 0$ | | -1 | 58 | 8 | | | $Be_0 = L$ | | Be |
| | | 58 | 8 | | | $Be_0 = L$ | Ph | | | | | | | | |
| | Muliplication with L, 0L | 59 | 9 | | Ae | | | | 59 | 9 | | | | Be | |
| | | 59 | 9 | u6 = L | | | | LL | 59 | 9 | u6 | u4 | | | ¼ Be |
| | | 59 | 9 | u6 = 0 | Ph | | | | | | | | | | |
| | Align → | 60 | 10 | | Ph, Ae | | | | 60 | 10 | | | $Be_{+1} = 0$ | Be | |
| | | 60 | 10 | | | $Be_{+1} = L$ | | +1 | 60 | 10 | | | $Be_{+1} = L$ | | ½ Be |
| | Test $10^{-6}$ | 61 | 10 | u7 = 0 | Lz | | | | 61 | 10 | | u1 | | | |
| | | 61 | 10 | u7 = L | Ph, Ae Be=>Bg | | | | | | | (delete u8) | | | |
| | Setting $10^{-6}$ | 62 | 11 | | Ph0, S0 (×) | | | | | | | | | | |

**Fig. 3.30** The microinstructions for decimal-binary conversion. Four decimal digits are entered through a mechanical device

| Part A (Exponent) | | | | | | | | | | Notification->B | Part B (Mantissa) | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Block 17 (left Part) | | | | | | Block 15a | | | | | Block 17 (right part) | | | | Block 15b | | |
| Operating sequence | | Criterion | | Cond. | to operate | Condition | to operate | Aa | Ab | | Criterion | | Condition | to operate | Condition | Ba | Bb |
| Symb | Description | No. | Ph | | | | | | | | No. | Ph | | | | | |
| Re-Transform (dual → decimal) (↙) | Setup Operands | 70 | 0 | d0 | Ph | | | Af | | | 70 | 0 | d0 | | | | Bf |
| | Setting of $10^{-6}$ | 71 | 1 | | Ae | Ae ≤ 3 | Ph | | | B | 71 | 1 | | A | (Ae ≤ 3) | Be | |
| | | | | | | Ae > 3 | Ph0, S1, Be (×) => $B_8$ | | | | | | | | | | |
| | b - Value 2 Positions → | 72 | 2 | | Ph, Ae | | | | | | 72 | 2 | | | | | ¼ Be |
| | Muliplication with 10 | 73 | 3 | | Ae | Ae < 0 | | | LL | B | 73 | 3 | | | A → (Ae < 0) | | ¼ Be |
| | | | | | | Ae ≥ 0 | Ph | | | | (73) | 3 | | d3 | | Be | |
| | Align → | 74 | 4 | | | Ae ≠ 0 | | | | B | (74) | 4 | | | A → (Ae ≠ 0) | 2Be | |
| | | 74 | 4 | | Ae | Ae ≠ 0 | | | -1 | | | | | | | | |
| | | | | | | Ae ≠ 0 | Ph | | | B | 74 | 4 | | | A → (Ae ≠ 0) | | Be |
| | actual Re-Compile | | | | | | | | | | 75 | 5 | | µPh ,d1 | | 2Be' | 8Be' |
| | | | | | | | | | | | 76 | 6 | | µPh ,d1 | | 2Be' | 8Be' |
| | | | | | | | | | | | 77 | 7 | | µPh ,d1 | | 2Be' | 8Be' |
| | | 78 | 8 | | Lz | | | | | | 78 | 8 | | d1 (Del → do) | | | |

**Fig. 3.31** The microinstructions for binary-decimal conversion. Four decimal digits are displayed in a mechanical device

The decimal exponent of the number is set with a lever. In cycle 9, as many multiplications with the factor 10 are performed as indicated by the position of the lever for the decimal exponent.

The table in Fig. 3.31 shows how a binary number contained in register Bf is transformed into a decimal number to be displayed in the decimal output field.

To avoid having to deal with negative decimal exponents, the number in register Bf is first multiplied by $10^{-6}$ (Zuse limited the operating range of the machine to handle only numbers larger than $10^{-6}$ as result, although partial results in the ALU could be smaller than this). This happens after phase 1. This multiplication is done by the multiplication operation in the Z1, and the decimal-binary conversion remains "suspended" during the cycles needed for the multiplication.

Then the mantissa is shifted two places to the right (to set the binary point so that we can now have four bits to its left). The mantissa is shifted until the exponent is positive, and then three multiplications by 10 follow. After each multiplication, the integer part of the mantissa is copied, erased from the mantissa, and the copy (four bits) is transformed into a decimal digit using a table (that is the 2B'-8B' operation in phase 4 to 7). Each decimal digit is displayed in the output panel (starting with the most significant decimal digit). Each time a multiplication by 10 is performed, the exponent arrow in the decimal display moves one place to the left.

## References

Bruderer, H. 2012. *Konrad Zuse und die Schweiz: Wer hat den Computer erfunden?* Munich: Oldenbourg Wissenschaftsverlag. https://doi.org/10.1524/9783486716658

Goldstine, H.H., and A. Goldstine. 1996. The Electronic Numerical Integrator and Computer (ENIAC). *Annals of the History of Computing* 18 (1): 10–16. https://doi.org/10.1109/85.476557

Materna, H. 2010. *Die Geschichte der Henschel Flugzeug-Werke in Schönefeld bei Berlin 1933-1945*. Bad Langensalza: Verlag Rockstuhl.

Rojas, R. 1997. Konrad Zuse's Legacy: The Architecture of the Z1 and Z3. *IEEE Annals of the History of Computing* 19 (2): 5–16. https://doi.org/10.1109/85.586067

Rojas, R. 1998. Die Rechenmaschinen von Konrad Zuse. Springer-Verlag, Berlin, DOI 10.1007/978-3-642-71944-8

Rojas, R. 2014. The Z1: Architecture and Algorithms of Konrad Zuse's First Computer. https://arxiv.org/abs/1406.1886, arXiv eprint

Rojas, R. 2016. The Design Principles of Konrad Zuse's Mechanical Computers. https://arxiv.org/abs/1603.02396, arXiv eprint

Schweier, U., and D. Saupe. 1988. Funktions- und Konstruktionsprinzipien der Programmgesteuerten Rechenmaschine "Z1". Arbeitspapiere der Gesellschaft für Mathematik und Datenverarbeitung 321

Zuse, K. 1936a. Die Aufstellung der Rechenpläne. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0208/

Zuse, K. 1936b. Die Rechenmaschine des Ingenieurs. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0209/

Zuse, K. 1936c. Die Rechenmaschine des Ingenieurs. Mathematische Probleme. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0214/

Zuse, K. 1937. Einführung in die allgemeine Dyadik. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0211/

Zuse, K. 1938. Die Rechenmaschine des Ingenieurs. Available online at the Zuse Internet Archive

Zuse, K. 1944. Mechanische Schaltgliedtechnik. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0226/

Zuse, K. 1952a. Rechenvorrichtung aus mechanischen Schaltgliedern. http://zuse.zib.de/

Zuse, K. 1952b. Rechenvorrichtungen aus mechanischen Schaltgliedern. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0961/

Zuse, K. 1952c. Theorie der mechanischen Schaltglieder. Available online at the Zuse Internet Archive

Zuse, K. 1954. Patentschrift nr. 907948: Mechanisches Schaltglied (für die Zuse KG). Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0999/

Zuse, K. 1955. Patentschrift nr. 924107: Aus mechanischen Schaltgliedern aufgebautes Speicherwerk (für Zuse). Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-1001/

Zuse, K. 1970. *Der Computer – Mein Lebenswerk*. Landsberg: Verlag Moderne Industrie.

# Chapter 4
# The Z2 and the Cipher Machine

*The Z2 was a prototype that proved that telephone relays could be used to substitute the mechanical relays of the Z1. The Z2 used the mechanical memory of the Z1. The processor operated with 16-bit integers and could compute only additions, subtractions, and multiplications. The processor was built with just 200 telephone relays.*

## 4.1 Architecture of the Z2

Finishing the mechanical Z1 was important to Konrad Zuse because this machine proved that the overall logical design of his computer was sound. However, the Z1 was not reliable enough; its mechanical components frequently jammed. Therefore, in 1939, Zuse decided to build a small computer prototype using telephone relays, as a proof of concept, before migrating the complete calculating machine to relay technology. However, he was drafted to the front in 1939 but was discharged in 1940 so that he could continue working for the Henschel company. The Z2 was built with funding from Kurt Pannke, a manufacturer of specialized calculating devices. Zuse built the prototype in the free time that his daily work allowed, during the evenings and weekends.

According to Konrad Zuse, the Z2 was shown in 1940 to Prof. Alwin Teichmann from the German Aircraft Research Unit (DVL). The successful demo convinced the DVL to provide partial funding for the next machine, the Z3, which Zuse had already started to build.

Friends of Zuse reported after the war that they witnessed the Z2 computing $3\times3$ determinants in 1940. The machine did not use floating point, as did the Z1. It was a fixed-point machine that could handle 16-bit integers. The complete processor had only 200 relays.

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_4

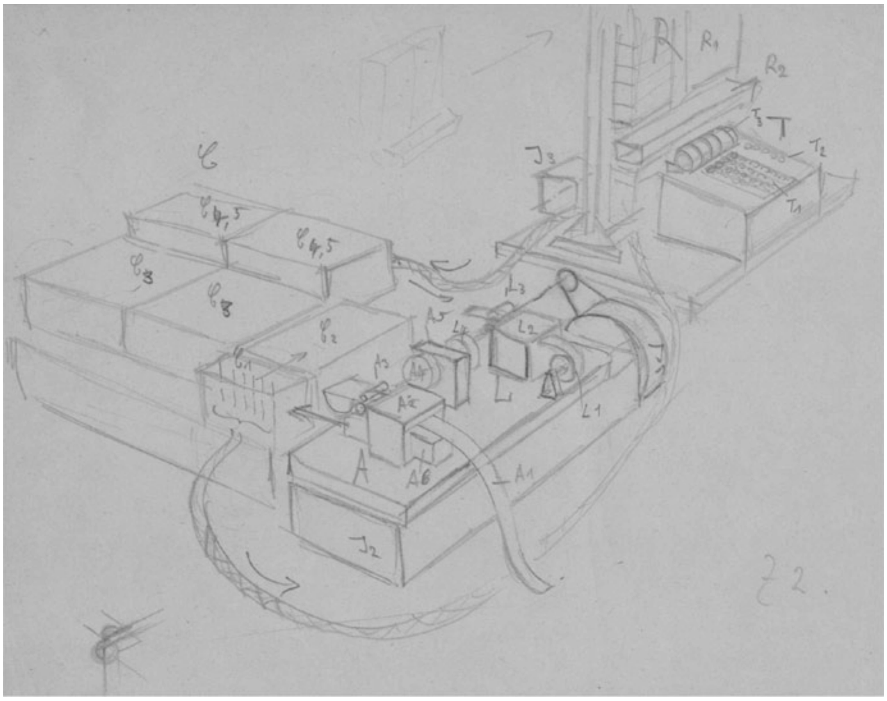


**Fig. 4.1** Sketch of the relay computer Z2. The processor relays and the input-output console are visible in the background (parts labeled T and R). The punched tape reader and the mechanical memory (labeled with C) are visible in the foreground (http://zuse.zib.de)

There is no written record from Zuse about the full range of capabilities of the machine, only the sketch shown in Fig. 4.1, which nevertheless shows some important facts about the machine. First, the output was displayed in binary, that is, there was no binary-decimal conversion in the machine. The input was also probably binary, through a simple keyboard. The memory and punched tape reader were taken from the Z1. The small total number of relays suggests to me that the machine could only add, subtract, and multiply integers. For the division algorithm, the processor would have required more than the 200 relays allocated to the processor. There are some cylinders visible below the relays for the processor. These are probably rotary dials like those used for sequencing the microinstructions of the Z3.

The Z2 was a minimal computer, and it shows, in retrospect, that the necessary technology for building binary computers was available in the early 20th century. What was lacking was a sound architectural design, which was provided by Zuse.

Before the Z2 was finished in 1940, there was another minimal machine, which was never built. It was the proposal for a cryptographic machine that Zuse authored, trying to be discharged from the front so that he could continue working on computing machines.

## 4.2 Konrad Zuse's Proposal for a Cipher Machine

Konrad Zuse wrote a proposal for a cipher machine during the winter of 1939/1940. The document was prepared at the Eastern Front during World War II and reached the German military authorities, possibly with the help of Kurt Pannke. Zuse's unpublished manuscript was found in his estate. Zuse's proposal was rejected by the military and the software-based cipher machine described in the letter never materialized.

### *4.2.1 The Context of the Invention*

It had been known for years that Konrad Zuse designed a cipher machine around 1939/1940 (one page of a handwritten description is reproduced in Zuse 1970), but until recently no further details had been available. This section provides an overview of Zuse's idea, the context of the invention, and its eventual rejection by the German military.

Konrad Zuse was called to the Eastern Front in August 1939. He was 29 years old and World War II proved to be a critical interruption of his work on computer machinery. He had started to tinker with mechanical binary elements as early as 1934, so that in February 1936, he was confident that he could build a complete calculating machine composed of such mechanical components. He built the Z1, a mechanical programmable calculator, between 1936 and 1938 in his parent's living room.

Zuse was forced to adapt his creation to the needs of the military. He had to "sell" his idea to be sent back to Berlin from the front so he could continue working on his machines. Previously, in 1937, Zuse had made contact with Kurt Pannke, who partially financed the construction of the Z1 (Zuse 1970). Pannke had worked in the past on machines for artillery calculations and even had a patent for such a device (Pannke 1927). It was probably Pannke's idea to modify the Z1 so that letters were exchanged between him and Zuse, discussing the possible adaptation of the machine to cryptography (Zuse 1940). The result was the document we present here, where the young soldier Konrad Zuse proposes to the military to build a binary programmable mechanical device for cryptography (Zuse 1939/1940). Potentially, any encoding algorithm could be implemented in the machine. In the document, Zuse gives an example of one possible encoding, making clear that the machine itself was not constrained by this choice.

As is evident from Zuse's pitch, he was not an expert cryptographer. He might have learned the basics before submitting his letter. It was written at the front (where Zuse was not involved in direct combat) and mailed to Berlin. A second proposal, with a more complex method for mixing bits, was prepared later by Zuse, and was also sent to the authorities. That second "algorithm" is not available in complete form, but the handwritten first page was reproduced by Zuse in his autobiography

(Zuse 1970). Nevertheless, the core of Zuse's argument was that, in principle, the concrete ciphering algorithm is just a matter of choice, once a computing machine is available that can carry out any desired computation. Random addressing of a table of random bits was the main new computational feature that Zuse could offer to cryptographers.

### 4.2.2 *Konrad Zuse's Letter*

The following is a translation of Konrad Zuse's proposal for a cipher machine (Zuse 1939/1940). The original letter (in German) is kept at Deutsches Museum in Munich, being part of the documents provided by Zuse's family to the museum after his death. The letter was typed. We find the date 1939/1940 handwritten on the first page (probably by Zuse).

Dipl. Ing. Konrad Zuse
Berlin SW 61, Methfesselstr. 10
Soldier
Field Postal Number 24 976
Post Office Berlin
Cipher Machine

My work on the machine I invented for technical computations has led to the development of mechanical switches and also of a mechanical storage composed of such elements. The disposition of the parts can be redesigned so that the machine can be adapted for cryptographic use. The advantage is, firstly, the spatial concentration and simplicity of the construction, secondly, the cipher can be built using functional laws as complex as desired. This is because the circuits can be modified with ease, as can be done with circuits made out of relays. In the following, we illustrate an encoding method that can be executed completely automatically by mechanical elements. The construction is independent of this specific method.

The text to be encoded consists of a sequence of groups of five binary symbols. Each combination (a letter) must be substituted by another one. The result R (the letter to be sent) is a function of the text T (the letters in the given text) and of an encoding combination S that we assume to consist of five binary symbols.

The function R=F(T,S) must be invertible in the function S=F(T,R), to make decoding possible. Let us consider two encoding functions as an example. The groups of five symbols are binary numbers with five digits each.

(1) The inversion method.

The digit $t_i$ of T is inverted when the digit $s_i$ of S is one. Example:

$$\mathrm{T} = 01101$$

$$\mathrm{S} = 11011$$

$$\mathrm{R} = 10110$$

(2) The addition method R = S + T.

If R becomes a binary number of six digits, we suppress the sixth digit. Example:

$$
\begin{array}{rl}
\mathrm{T} = & 01101 \\
\mathrm{S} = & 11011 \\
& 101000 \\
\mathrm{R} = & 01000
\end{array}
$$

Both solutions are straightforward to implement. The binary adder has been tested in a prototype. The machine can also change the encoding combination S continuously, following certain rules and maintaining the calculation laws determined by the initial configuration. We can store in the machine several initial configurations, for example seven, of five binary digits. They produce a binary number of 35 digits, from which we only need to use 32 positions. Using a decoder, we can select from the binary sequence the $n$-th and as many binary digits as needed for the key S. From S and T we determine R. R is equal to the next N.

| key N | N binary | random |
|---|---|---|
| 0 | 00000 | 0 |
| 1 | 00001 | 0 |
| 2 | 00010 | 1 |
| 3 | 00011 | 1 |
| 4 | 00100 | 1 |
| 5 | 00101 | 0 |
| 6 | 00110 | 1 |
| 7 | 00111 | 0 |
| 8 | 01000 | 1 |
| 9 | 01001 | 0 |
| 10 | 01010 | 1 |
| 11 | 01011 | 1 |
| 12 | ... | 0 |
| 13 | ... | 1 |
| 14 | | 1 |
| 15 | ... | 1 |
| 16 | | 0 |
| 17 | | 1 |
| 18 | | 1 |
| 19 | | 1 |
| 20 | | 0 |
| 21 | | 0 |
| 22 | | 1 |
| 23 | | 0 |
| 24 | | 0 |
| 25 | | 0 |
| 26 | | 1 |
| 27 | | 1 |
| 28 | | 0 |
| 29 | | 0 |
| 30 | | 0 |
| 31 | | 1 |

Start number: 25 31 1

| | T | N | S | T+N | R |
|---|---|---|---|---|---|
| 1 | 00010 2 | 25 | 01000 12 | 14 | 14 01110 |
| 2 | 00110 6 | 14 | 11011 27 | 33 | 1 00001 |
| 3 | 00110 6 | 1 | 01110 14 | 20 | 20 10100 |
| 4 | 01000 8 | 20 | 00100 4 | 12 | 12 01100 |
| 5 | 01001 9 | 12 | 01110 14 | 23 | 23 10111 |
| 6 | 11010 26 | 23 | 00011 3 | 29 | 29 11101 |
| 7 | 00110 6 | 29 | 00100 4 | 10 | 10 01010 |
| 8 | 00000 0 | 10 | 11011 27 | 27 | 27 11011 |
| 9 | 11110 30 | 27 | 10001 47 | 47 | 15 01110 |

The process can be inverted easily to decode.

### *4.2.3 Discussion*

There are several interesting aspects in Zuse's letter. First, he proposes to use a purely binary approach. Machines such as the Enigma and later the Lorenz SZ40 were mechanical devices based on rotors and some wiring. They were provided to the users as black boxes. The user could only adjust the start position of the rotors but not completely reprogram the machine. In Zuse's proposal, the pseudorandom stream of binary digits is provided in advance (it can be generated by any means), can be kept in memory, and the "scrambling," that is, the selection of the bits needed for a substitution, could be done with any good algorithm. Zuse does not compare his machine to those already known, simply because he was unaware of the cryptographic methods they implemented.

Zuse's method consists in taking 35 random bits and then generating a random pointer to any position between 0 and 31. The five bits starting at the pointer would be taken and used for the encoding (the last position, 31, can use the additional bits at positions 32 to 34). A random pointer to a random position would seem to be a good way of scrambling text. However, the reuse of the random bits would be significant in an encoded text of any useful length, so that the cipher could have been easily broken.

For the combination of a letter with five random bits, Zuse proposes to use XOR (the "inversion method") or addition. The expressions R=F(T,S) and S=F(T,R) are only valid for the XOR case (since S=T XOR (T XOR S)). For the addition case, subtraction would be needed to recover S. To recover the plain text from the cipher, the key and the starting position are needed. Given S and R, T can be recovered for the first line of the table in Zuse's letter ($14 - 12 = 2$). R provides the next pointer into the key, for recovering five bits of the key, and for proceeding to the next line of decoding ($33 - 27 = 6$).

Zuse's letter has only historical value today. It shows that very early during the development of the computer, he was aware of the full range of applications that a fast-calculating machine could have.

## References

Pannke, K. 1927. Rechenvorrichtung. German Patent Office, Patent 447780
Zuse, K. 1939/1940. Chiffriermaschine. Zuse Archive at Deutsches Museum, images 202_4-01, 202_4-02, 202_4-03
Zuse, K. 1940. Draft of a letter to Kurt Pannke. Zuse Archive at Deutsches Museum
Zuse, K. 1970. *Der Computer – Mein Lebenswerk*. Landsberg: Verlag Moderne Industrie.

# Chapter 5
# The Architecture of the Z3

*This chapter reviews the first detailed description (published in 1997) of the architecture of the Z1 and Z3 computing machines designed by Konrad Zuse in Berlin between 1936 and 1941. The necessary information was obtained from a careful evaluation of the patent application filed by Zuse in 1941. Additional insight was gained from a software simulation of the machine's logic.*[1]

The Z1 was constructed using purely mechanical components, while the Z3 used electromagnetic relays. Despite these differences, both machines shared a common logical structure and utilized the same programming architecture. We argue that both the Z1 and the Z3 possessed features comparable to those of modern computers. They consisted of separate memory and processor units, and the processor was capable of handling floating-point numbers, performing the four basic arithmetic operations, and computing square roots. Programs were stored on punched tape and read sequentially.

## 5.1 Early Computing Machines

Konrad Zuse's Z1, a programmable automaton built between 1936 and 1938, is one of the "first computers" in the world. Zuse decided to build his first experimental calculating machine around two main ideas: (a) the machine would work with binary numbers; (b) the computing and control unit would be separated from the storage. In 1936, the memory[2] of the planned machine was completed. The processor of the Z1 was completed a few months after the storage unit, using the same kind

[1] This chapter is based on Rojas (1997). The complete set of numerical algorithms for the Z3 was published in Rojas (1998).

[2] Zuse called it the "Speicherwerk" (storage mechanism). The term "Speicher" is still used in German instead of the anthropomorphic term "memory" introduced by John von Neumann.

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_5

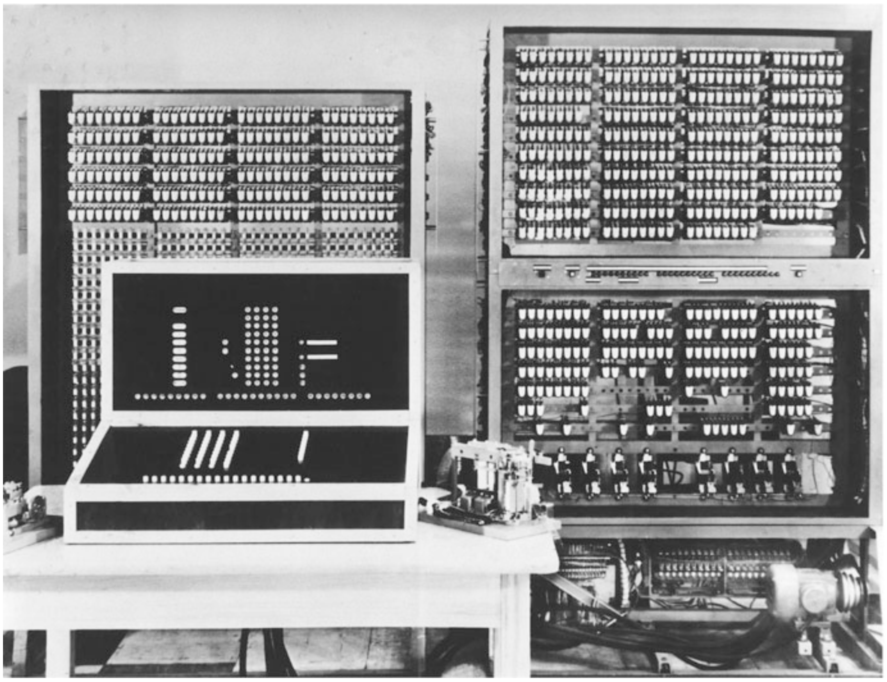


**Fig. 5.1** Reconstruction of the Z3 in Deutsches Museum in Munich (Image: Konrad Zuse Internet Archive, http://zuse.zib.de)

of technology. Despite its unreliability, the Z1 demonstrated the soundness of its architectural design and motivated Zuse to explore alternative technologies. As an interim step, Zuse constructed the Z2, a simpler model that used a hybrid approach, combining a relay-based processor with a mechanical memory. Subsequently, Zuse embarked on the construction of the Z3, a machine that relied solely on relays, while retaining the same logical design as the Z1. It was completed and operational in 1941, 4 years before the ENIAC (Fig. 5.1).

The Z3 was documented by Zuse in his patent application Z391 of 1941, which is rather difficult to decipher due to the non-standard notation and terminology used (Zuse 1941). Czauderna's book about the Z3 is a good source for understanding the historical context of Zuse's inventions but does not describe the Z3 in detail (Czauderna 1979). The main architectural difference between the Z1 and Z3 is the absence of the square root operation in the Z1. Additionally, there are minor differences in the number of bits used for arithmetic operations within the processor. Specifically, the Z1 used one bit less for the mantissa of floating-point numbers. Furthermore, differences exist in the number of cycles required for each instruction and the implementation of numerical exceptions.

## 5.2 Architectural Overview of the Z3

This section provides a concise summary of the most relevant architectural features of the Z3. We will start with an overview of the architecture and gradually move into more detailed explanations. For clarity and consistency, the Z3 is described in the present tense.

### *5.2.1 Block Structure*

The Z3 is a floating-point machine. While other early computing automata such as the Mark I, the ABC, and the ENIAC worked with fixed-point numbers, Zuse decided early on to adopt what he called the "semi-logarithmic" notation, which corresponds to the modern floating-point representation.

Figure 5.2 shows an overview of the main building blocks of the Z3. A key aspect is the separation between the processor and memory. The Z3 consists of a binary memory unit (capable of storing 64 floating-point numbers), a binary floating-point processor, a control unit, and input/output devices. The memory and the arithmetic units are connected by a data bus that carries the exponent and mantissa of the floating-point representation. The control unit contains the microsequencers required for each instruction. Control lines going from the control unit to the processor, the memory, and the I/O devices enforce proper synchronization of all units. The tape reader provides the opcode of each instruction and the addresses for

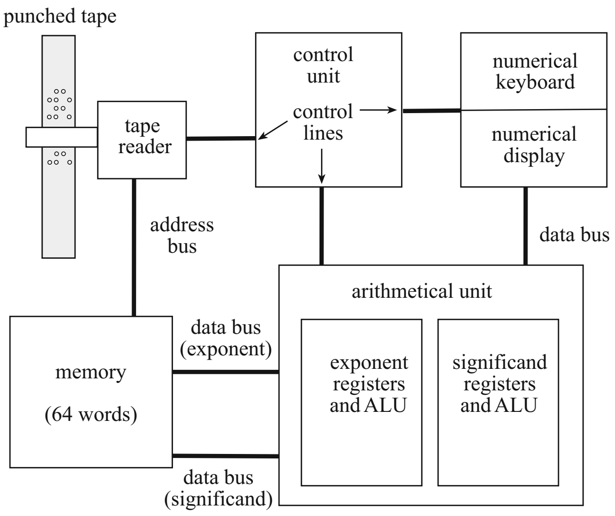


**Fig. 5.2** The building blocks of the Z3

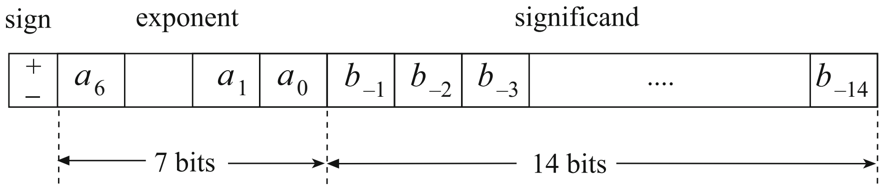


**Fig. 5.3** The floating-point representation in memory

memory accesses. The I/O devices are connected to the computing unit via a data bus.

## 5.2.2 *Floating-Point Representation*

Figure 5.3 shows the representation used in the memory of the Z3. The first bit is used to store the sign of the number, the next 7 bits are used for the exponent, and the last 14 bits for the mantissa (only the 14 places to the right of the binary point). The bits of the exponent are called part "A" of the number and are denoted by $a_6, \ldots, a_0$. The bits of the mantissa are called part "B" of the number and are denoted by $b_0, b_{-1}, \ldots, b_{-14}$. The exponent is encoded as a two's complement number. The range of possible values runs therefore from $-64$ to 63. The mantissa is stored in *normalized* form,[3] that is, the first digit before the binary point ($b_0$) must always be a 1. This digit does not need to be stored (and therefore does not appear in Fig. 5.3) so that the effective range of the numbers in the memory unit corresponds to a mantissa (significand) of 15 bits. However, there is a problem with the number zero, which cannot be expressed with a normalized mantissa. The Z3 uses the convention that any FP number with an exponent of $-64$ is considered equal to zero. Any number with an exponent of 63 is considered infinitely large. Operations involving zero and infinity are treated as exceptions and special hardware monitors the numbers loaded into the processor in order to set the exception flags (see Sect. 5.4).

With this convention, the smallest number representable in the memory of the Z3 is $2^{-63} = 1.08 \times 10^{-19}$ and the largest is a little less than $2^{63}$. The arguments for computations can be entered as decimal numbers on the keyboard of the Z3 (four digits). The exponent of the decimal representation is entered by pressing one of the keys labeled $-8, -7, \ldots, 7, 8$. The original Z3 could only accept input between $1 \times 10^{-8}$ and $9999 \times 10^{8}$. The reconstruction of the Z3 built by Zuse for Deutsches Museum in Munich provides enough keys for larger exponents. With this arrangement, the numerical capacity of the machine corresponds to the acceptable input. However, the Z3 does not print the numerical results produced by

[3] Donald Knuth attributes the invention of normalized floating-point numbers to Zuse (Knuth 1981).

the program. A result is displayed on an array of lamps that represent the digits from 0 to 9 (for each decimal place). The largest number that can be displayed is 19999. The smallest is 00001. The largest exponent that can be displayed is +8, and the smallest −8.

### 5.2.3 Instruction Set

The Z3's program is stored on punched tape. Each instruction is encoded using 8 bits for each row of the tape. The instruction set of the Z3 consists of the nine instructions shown in Table 5.1. There are three types of instruction: I/O, memory, and arithmetic operators. The opcode has a variable length of 2 or 5 bits. Memory operations encode the address of a word in the lower six bits, that is, the addressing space has a maximum size of 64 words, as we mentioned before.

The instructions on the punched tape can be combined in any order. The instructions Lu and Ld (read from the keyboard, display result) stop the machine so that the operator has enough time to enter a number or write down a result. The machine is then restarted and continues processing the program.

The instruction most conspicuously missing from the instruction set of the Z3 is conditional branching. Loops can be implemented by the simple expedient of joining the two ends of the punched tape, but there is no way to implement conditional sequences of instructions. Therefore, the Z3 is not a universal computer in the sense of Turing.

### 5.2.4 Number of Cycles

The Z3 is a clocked machine. Each cycle is divided into five "stages" called I, II, III, IV, and V. The instruction in the punched tape is decoded in stage I of a cycle. The two basic arithmetic operations of the machine are addition and subtraction of the

**Table 5.1** Instruction set and opcodes of the Z3

| Type | Instruction | Description | Opcode |
|---|---|---|---|
| I/O | Lu | Read keyboard | 01 110000 |
| | Ld | Display result | 01 111000 |
| Memory | Pr z | Load address z | 11 $z_6z_5z_4z_3z_2z_1$ |
| | Ps z | Store address z | 10 $z_6z_5z_4z_3z_2z_1$ |
| Arithmetic | Lm | Multiplication | 01 001000 |
| | Li | Division | 01 010000 |
| | Lw | Square root | 01 011000 |
| | $Ls_1$ | Addition | 01 100000 |
| | $Ls_2$ | Subtraction | 01 101000 |

exponents and mantissas. These operations can be executed in the first three stages of each cycle. Stages IV and V are used to prepare arguments for the next operation or to write back results.

The instructions implemented in the Z3 require the following number of cycles:

| | |
|---|---|
| Multiplication: | 16 cycles |
| Division: | 18 cycles |
| Square root: | 20 cycles |
| Addition: | 3 cycles |
| Subtraction: | 4 or 5 cycles, depending on the result |
| Read keyboard: | 9 to 41 cycles, depending on the exponent |
| Display output: | 9 to 41 cycles, depending on the exponent |
| Load from memory: | 1 cycle |
| Store to memory: | 0 or 1 cycle |

According to Zuse, the time required for a multiplication was 3 seconds. Considering that a multiplication operation takes 16 cycles, we can estimate that the operating frequency of the Z3 was $16/3 \approx 5.33\,\text{Hz}$.[4]

The number of cycles needed for the *read* and *display* instructions is variable because it depends on the exponent of the argument. Since the input must be converted from decimal to binary representation, the number of multiplications required with a factor of 10 or 0.1 is dictated by the decimal exponent (see Sect. 5.4).

Addition and subtraction require more than one cycle because in the case of floating-point numbers, care must be taken to set the size of the exponent of both arguments to the same value. This requires some additional comparisons and shifting.

A number can be stored in memory in *zero* cycles if the result of the last arithmetic operation can be redirected to the desired memory address. In this case, the cycle required for the store instruction overlaps with the last cycle of the arithmetic operation.

### 5.2.5 Programming Model

It is very important to describe the programming model of the Z3, that is, the part of the machine visible to the programmer. From the point of view of the software, the Z3 consists of 64 memory words that can be loaded into two floating-point registers, which we will simply call R1 and R2. These two registers contain the two arguments for arithmetic operations (the square root operation uses only R1). The programmer can write any desired sequence of instructions but must take into account the state of the machine's registers.

[4] It is a curious fact of life that the gate-level animated simulation of the Z3 implemented by my students in 1994 for the first browsers also required about 3 seconds for a multiplication!

The important thing to remember is that the first load operation in a program (Pr $z$) transfers the contents of address $z$ to R1. A second load operation transfers a word from memory to R2. A read keyboard instruction loads the numerical input into register R1 and *destroys* register R2.

Arithmetic operations do not specify their arguments in the opcode. Their implicit semantics is the following:

| | |
|---|---|
| Multiplication: | R1:=R1×R2 |
| Division: | R1:=R1/R2 |
| Addition: | R1:=R1+R2 |
| Subtraction: | R1:=R1−R2 |
| Square root: | R1:=sqrt(R1) |

In the Z3, after executing an arithmetic instruction, register R2 is set to zero, and the result of the operation is stored in register R1. Subsequent load operations refer to R2. The store and display instructions always refer to register R1, which contains the result of the previous arithmetic operation. Following a store or display operation, R1 is reset to zero. The next load operation then refers to R1.

An example is better than many additional remarks to illustrate the programming model of the Z3. Suppose we want to compute a polynomial using Horn's method:

$$x(a_2 + x(a_3 + xa_4))) + a_1.$$

Suppose we also want to store the constants $a_4, a_3, a_2, a_1$ in the addresses 4, 3, 2, and 1 of the memory unit. The value $x$ is stored at address 5. The program that performs the desired computation is the following:

| | |
|---|---|
| Pr 4 | load $a_4$ in R1 |
| Pr 5 | load $x$ in R2 |
| Lm | multiply R1 and R2, result in R1 |
| Pr 3 | load $a_3$ in R2 |
| $Ls_1$ | add R1 and R2, result in R1 |
| Pr 5 | load $x$ in R2 |
| Lm | multiply R1 and R2, result in R1 |
| Pr 2 | load $a_2$ in R2 |
| $Ls_1$ | add R1 and R2, result in R1 |
| Pr 5 | load $x$ in R2 |
| Lm | multiply R1 and R2, result in R1 |
| Pr 1 | load $a_1$ in R2 |
| $Ls_1$ | add R1 and R2, result in R1 |
| Ld | display result |

After the last instruction has been executed, the processor is reset to its initial state. A new program sequence can be started.

## 5.3 Block Diagram of the Z3

In this section, we take a closer look at the structure of the Z3 and describe its main building blocks in more detail. The main issue is how to enforce the correct synchronization of the available components.

### *5.3.1 The Processor*

Figure 5.4 shows a simplified representation of the arithmetic unit of the Z3. There are two parts: the left side is used for operations with the exponents of the floating-point numbers, and the right side is used for operations with the mantissas. Af and Bf are registers used to store the exponent and mantissa of the first register, from the programmer's point of view. We will refer to R1 as the register pair <Af,Bf>. The register pair <Ab,Bb> stores the exponent and mantissa of R2. The pair <Aa,Ba> contains the exponent and the mantissa of a third temporal floating-point register invisible to the programmer. The two ALUs A and B are used to add or subtract exponents and mantissas respectively. The result of the operation in the exponent part is placed in Ae. In the mantissa part, the result of the operation is put into Be. In part B, a multiplexer allows the selection of Ba or the output of the ALU as the

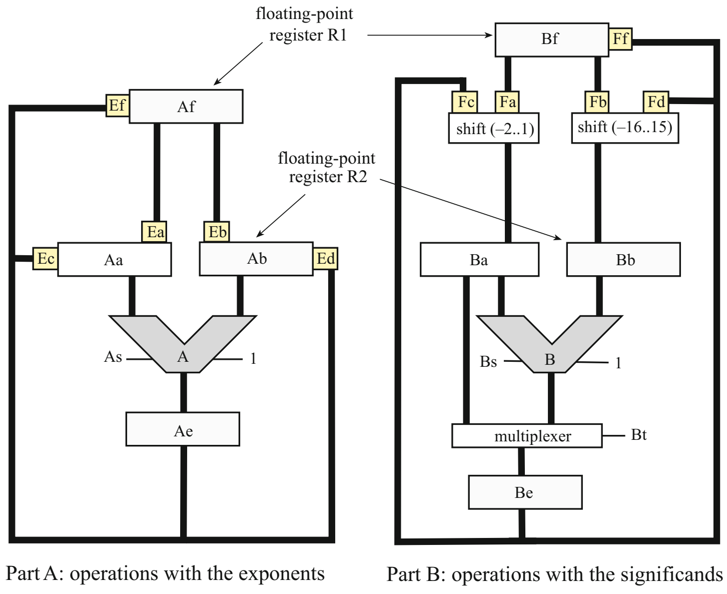

floating-point register R1
Bf
Ff
Ef
Af
Fc
Fa
Fb
Fd
shift (–2..1)
shift (–16..15)
floating-point register R2
Ea
Eb
Ec
Aa
Ab
Ed
Ba
Bb
As
A
1
Bs
B
1
Ae
multiplexer
Bt
Be
Part A: operations with the exponents
Part B: operations with the significands

**Fig. 5.4** The registers and datapath

result of the operation. The multiplexer is controlled by a relay Bt (if Bt=0 then Be is set equal to Ba).

The small boxes labeled Ea, Eb, Ec, Ed, Ef, Fa, Fb, Fc, Fd, and Ff are switches that open or close the data bus. If the contents of register Af are to be transferred to Aa, for example, the box of relays Ea is set to 1 and the result is Aa:=Af. As can be seen from the diagram, the contents of Af can be transferred to Aa or Ab, while the contents of Ae can be transferred to any of Aa, Ab, or Af, according to the states of the switches. The structure of part B of the arithmetic unit is very similar, but in addition to the multiplexer controlled by the relay Bt, there is also a shifter between Bf and Ba and a shifter between Bf and Bb. The first shifter can shift the mantissa up to two positions to the right and one position to the left. This is equivalent to a division of Bf by 4 or to multiplication by the constant 2. The second shifter can shift the mantissa in Af from 1 to 16 positions to the left and from 1 to 15 positions to the right. These shifts are necessary for addition and subtraction of floating-point numbers. Multiplication and division with powers of 2 can therefore be performed when the operands are fetched for the next arithmetic operation and thus do not consume any time.

The number of bits used in the registers is the following:

| | | | |
|---|---|---|---|
| Af | 7 bits | Bf | 17 bits |
| Aa | 7 | Ba | 19 |
| Ab | 7 | Bb | 18 |
| Ae | 8 | Be | 18 |

As can be seen from this list, Ae uses an extra bit to handle the addition of the exponents of the arguments. Part B of the processor uses two additional bits for the mantissas ($b_{-15}, b_{-16}$) and makes explicit $b_0$, which is not stored in memory. The extra bits at positions $-15$ and $-16$ are included to increase the precision of the computations. The total number of bits required to store the result of an arithmetic operation in Bf is therefore 17 bits. The registers Ba and Bb need more extra bits ($ba_2, ba_1$, and $bb_1$) to store intermediate results of some of the numerical algorithms. The square root algorithm, in particular, can result in partial computations in Ba that require three bits to the left of the binary point.

The basic primitive operation of the datapath is the addition or subtraction of exponents or mantissas. When the relays As or Bs are set, the negation of the second argument (Ab or Bb) is fed into the ALU. Thus, if the As relay is set to 1, the ALU in part A subtracts its arguments, otherwise, they are added. The same is true for part B and the relay Bs. The constant 1 is needed to build the two's complement of a number.

Suppose that two numbers with the same exponent are to be added. The first exponent is stored in Af and the second in Ab. Since they are equal, no operation has to be performed on this side of the machine. In part B, the mantissa of the first number is stored in Bf and the mantissa of the second in Bb. The first step is to load Bf into Ba by setting the relay box Fa to 1. The addition is performed next: the relay Bt is set to zero, and the result Ba+Bb is assigned to Be. The relay box Ff is now set to 1, and the result is stored in Bf. As we can see, the information can move between registers and flow through the datapath. The computer architect must determine the correct sequence of activations of the relay boxes in order to get the desired operation. This is done in the Z3 using a technique very similar to microprogramming.

## 5.3.2 *The Control Unit*

Figure 5.5 shows a more detailed diagram of the control unit and the I/O panels. The circuit Pb decodes the opcode of the instruction read from the punched tape. If it is a memory instruction, circuit Pb sets the address bus to the value of the lower six bits of the opcode. The control unit determines the correct microsequencing of the instructions. There are special circuits for each of the operations in the instruction set.

Circuit Z represents the panel of buttons used to enter a decimal number into the machine. Only one key in each of the four columns can be activated. The exponent is set by pressing one of the keys labeled $-8$ to 8 in circuit K. The output display is very similar to the input panel, but here lamps illuminate the appropriate decimal digits, the exponent of the number (circuit Q), as well as its sign. Note that there is a fifth digit for the output (which can only be 1 or 0).

Once a decimal number has been set, a data bus transmits the digits to register Ba and a complex series of operations is started. The decimal input must be transformed into a binary number. This requires a chain of multiplications, which is longer depending on the absolute magnitude of the exponent. If the exponent is zero, the whole transformation takes 9 cycles, but if the exponent is 8, the operation requires $9 + 4 \times 8 = 41$ cycles.

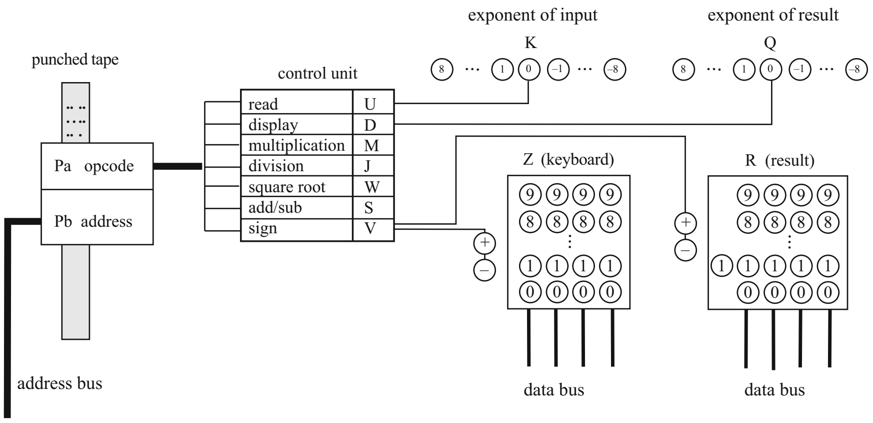


**Fig. 5.5** The control unit and I/O panels

### 5.3.3 Microcontrol of the Z3

The heart of the control unit are its microsequencers. Before we describe how they work, it is necessary to take a closer look at the chaining of arithmetic instructions in the Z3. Figure 5.6 shows the main idea. Each cycle of the Z3 is divided into five stages. Stages IV and V are used to move information in the machine. During stages I, II, and III an addition/subtraction is computed in part A and another in part B. We call this the "execute" phase of an instruction. A typical instruction fetches its arguments, executes, and writes back the result. Zuse took great care to save execution time by overlapping the fetch phase of the next instruction with the write-back phase of the current one. We can think of an execution cycle as consisting of just two phases, as shown in Fig. 5.6 where the first two cycles of a series of instructions have been labeled. We have adopted this convention in the tabular diagrams of the numerical algorithms discussed later on.

The microsequencing is done by special control wheels. There is one for the multiplication algorithm, another to control the division, and another for the square root instruction. The moving arm shown in Fig. 5.7 starts to move clockwise as soon as the control unit decodes the corresponding instruction. In each cycle, the arm

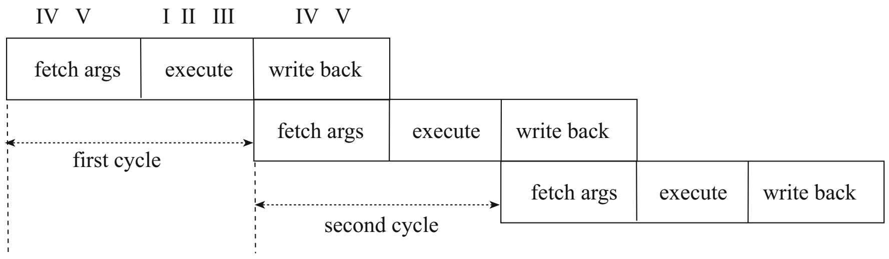


**Fig. 5.6** The execution pipeline of the Z3

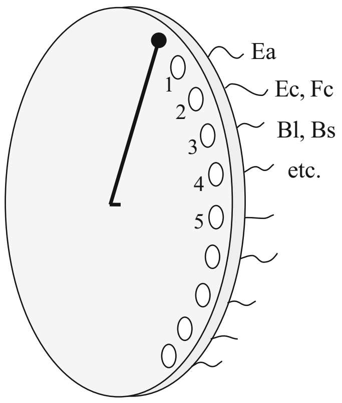


**Fig. 5.7** Control wheels for microsequencing

moves from one position to the next. The arm conducts electricity and activates the circuits it comes into contact with. In the example shown in the figure, the moving arm sets the relay box Ea to 1 in the first cycle. This leads to the transfer of the contents of register Af into Aa. In the next cycle, the relay boxes Ec and Fc are activated. In this way, the results of the operations in parts A and B are written back to the registers Aa and Ba, respectively. As can be seen, such control wheels provide a convenient platform for modifying the exact sequence of events during an operation. They correspond to the microsequencers used today in modern microprocessors. I stop short of calling them a form of microprogramming, because in this case the microsequence is hardwired, but it is obvious that microsequencing and microprogramming are closely related.

Extensive use of microsequencing allowed Zuse to simplify the Z3. Once the basic circuits had been laid out, it was just a matter of refining the control until optimal sequences of events could be found. There are a lot of details that need to be kept in mind by the engineer designing the "microprogram." Otherwise, short circuits could destroy the hardware. The Z1 with its mechanical design was even more sensitive in this respect than the Z3. Even after it was finished, there were sequences of instructions that the programmer had to avoid in order not to damage the hardware. One of those sequences was inadvertently tried at the German Museum of Technology in Berlin in 1994, causing minor damage to the reconstructed Z1.

### *5.3.4 The Adders*

An important feature of the Z3 is the design of the adders, which compute additions and subtractions using a method called *carry look-ahead*. When binary addition is implemented in a straightforward way, carries have to be passed from one bit position to the next. In the case of the mantissa, we would need 16 cycles just for the transmission of the carry bits.

The adders designed by Zuse are much faster than this—they perform an addition or subtraction in stages I, II, and III of a single cycle. Subtraction is computed by complementing the second argument and adding an extra 1 at the lowest bit position.

Consider the addition of the registers Ba and Bb. First of all, a partial result Bc is computed, which is the bitwise XOR of both registers, i.e., $bc_i = ba_i$ XOR $bb_i$. We will refer to bit $i$-th of register Bb by $bb_i$ or Bb[$i$], whichever form seems more convenient. The same notation will be used in the case of other registers. A second partial result is the bitwise AND operation applied to both registers, i.e., $bh_i = ba_i$ AND $bb_i$. This last operation locates the bit positions where a carry is needed. The intermediate result Bd is computed by using the circuit shown in Fig. 5.8. The input to the circuit consists of the bits $bh_1, \ldots, bh_{-16}$ computed previously. When a bit is 1, the corresponding line carries a current. Otherwise, the line is disconnected from the power source (three-state). The rest position of the relays $bc_1, \ldots, bc_{-16}$ is the one shown in the figure. If bit $bc_i$ is equal to 1 the corresponding relay is

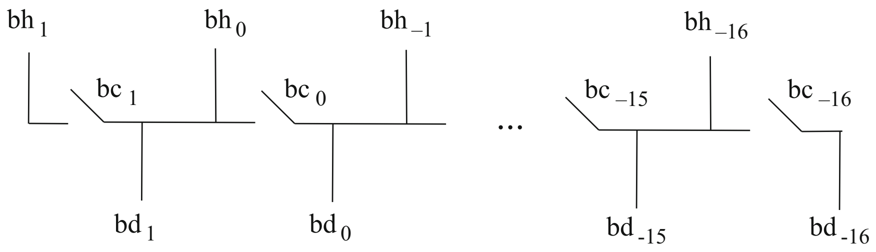


**Fig. 5.8** Circuit for carry look-ahead

closed. The final result is $be_i = bd_i$ XOR $bc_i$. Note that the use of relays simplifies the propagation of the carries up to the last required bit position. Since all relays are activated simultaneously, the carry is not delayed in going from one bit position to the next.

## 5.4 Numerical Algorithms

In this section, we describe the floating-point algorithms used by the Z3. They are, without exception, the same as those normally used in small sequential floating-point processors (Koren 1993). The description of the algorithms has been verified by Julius Range using a simulation of the Z3 (Range 2016).

### *5.4.1 Floating-Point Exceptions*

The problem with floating-point notation is that special conventions must be used to deal with the number zero. The Z3 solves this problem and handles other exceptions (overflow, underflow) by monitoring the value of the exponent after every arithmetic operation or a load from memory. A special circuit looks at the state of the bus Ae and catches exceptions. Any number with exponent −64 is flagged as zero: a relay denoted $Nn_1$ is set to 1 if the number is stored in the register pair <Af,Bf>. If the number is stored in the register pair <Ab,Bb>, the relay $Nn_2$ is set to 1. In this way, we always know if one or both arguments of an arithmetic operation are zero. Something similar is done for every exponent of value 63 (an infinite number, by convention). In this case, the relays $Ni_1$ or $Ni_2$ are set to 1 depending on the register pair where the number is stored.

Operations involving "exceptional" numbers (zero or infinity) are performed as usual, but the result is overwritten by the snooping circuit. Assume, for example, that a multiplication is computed and the first argument is zero ($Nn_1$ is set to 1). The computation proceeds as usual, but in each cycle the snooping circuit produces the result −64 at the output of the adder of part A. It does not matter what operations are performed with the mantissas because the exponent of the result is set to −64 and therefore the final result is zero. Division by infinity can be handled similarly. The Z3 can detect undefined operations like $0/0$, $\infty - \infty$, $\infty/\infty$ and $0 \times \infty$. In all these cases the corresponding exception lamp will light up on the output panel and the machine will stop. The Z3 always produces the correct result when one of the arguments is zero or $\infty$ and the other a number within bounds.[5]

An additional circuit looks at the exponent of the result at the output of the exponents' adder. If the exponent is greater than or equal to 63, an overflow has occurred, and the result must be set to $\infty$. If the exponent is less than $-63$, an underflow has occurred, and the result must be set to 0. To do this, the corresponding relay ($\mathrm{Nn}_1$ or $\mathrm{Ni}_1$) is set to 1.

Zuse managed to implement exception handling using just a few relays. This feature of the Z3 is one of the most elegant in the whole design. Many of the early microprocessors of the 1970s did not include exception handling and left this to the software. Zuse's approach is better, since it frees the programmer from the tedium of checking the bounds of his numbers before each operation.

### 5.4.2 *Addition and Subtraction*

To add or subtract two floating-point numbers $x$ and $y$, their representation must be reduced to the same exponent. After this has been done, only the mantissas need to be added or subtracted. If the exponents are different, the mantissa of the smaller number is shifted to the right by as many places as necessary (and its exponent is incremented accordingly to keep the number unchanged) until both exponents are equal. Of course, it can happen that the smaller number becomes zero after 17 shifts to the right.

The signs of the two numbers are compared before deciding on the type of operation to be executed. If an addition has been requested and the signs are equal, the addition is performed. If the signs are different, a subtraction is executed. If a subtraction has been requested and the signs are different, an addition is executed. If the signs are the same, the subtraction is executed. A special circuit sets the sign of the result according to the signs of the arguments and the sign of the result.

Addition and subtraction are controlled by a chain of relays (not by a control wheel) since the maximum number of cycles needed is low. Figure 5.9 shows the synchronization required for the addition of two numbers. Initially, the arguments for the addition are stored in the register pairs <Af,Bf> and <Ab,Bb>. In the first cycle, the exponents are subtracted. In cycle 2, the mantissa with the larger exponent is loaded into register Ba and the mantissa with the smaller exponent into register Bb. The mantissa in register Bb is shifted as many places to the right as the absolute value of the difference of the exponents (exception handling takes care of the case when the smaller number becomes zero after the shift). In stages I, II, and III of cycle

[5] This was not the case with the Z1. Zuse thought of, but did not implement, exception handling in the Z1. The machine could not correctly perform computations involving zero [Zuse, personal communication].

| cycle | stage | exponent | mantissa |
|---|---|---|---|
| 0 | I,II,III | | |
| 1 | IV,V | Aa:=Af | |
| | I,II,III | Ae:=Aa−Ab | Be:=0+Bb |
| 2 | IV,V | if (Ae≥0) then Ab:=0, Aa:=Af<br>else Aa:=0 | if (Ae≥0) then Ba:=Bf, Bb:=Be (shifted)<br>else Ba:=Be, Bb:=Bf (shifted)<br>(Be or Bf are shifted \|Ae\| places to the right) |
| | I,II,III | if (Be≥2) then Ae:=Aa+Ab+1<br>else Ae:=Aa+Ab | Be:=Ba+Bb |
| 3 | IV,V | Af:=Ae | if (Be≥2) then Bf:=Be/2<br>else Bf:=Be |

**Fig. 5.9** The 3 cycles needed for the addition algorithm. The arguments for the addition are stored in the register pairs <Af,Bf> and <Ab,Bb> before the operation is started

| cycle | stage | exponent | mantissa |
|---|---|---|---|
| 0 | I,II,III | | |
| 1 | IV,V | Aa:=Af | |
| | I,II,III | Ae:=Aa−Ab | Be:=0+Bb |
| 2 | IV,V | if (Ae≥0) then Ab:=0, Aa:=Af<br>else Aa:=0 | if (Ae≥0) then Ba:=Bf, Bb:=Be (shifted)<br>else Ba:=Be, Bb:=Bf (shifted)<br>(Be or Bf are shifted \|Ae\| places to the right) |
| | I,II,III | Ae:=Aa+Ab | Be:=Ba−Bb |
| 3 | IV,V | Aa:=Ae, Ab:=0 | Ba:=0, Bb:=Be |
| | I,II,III | Ae:=Aa+Ab | Be:=Ba−Bb |
| 4 | IV,V | Aa:=Ae<br>Ab:= number of shift positions | Bb:=Be (shifted)<br>(Be is normalized by shifting to the left) |
| | I,II,III | Ae:=Aa−Ab | Be:=0+Bb |
| 5 | IV,V | Af:=Ae | Bf:=Be |

**Fig. 5.10** The 4–5 cycles needed for the subtraction algorithm. The first argument is stored in the register pair <Af,Bf> and the second in <Ab,Bb> before the operation is started

2 the mantissas are added, and finally, the processor tests if the result is greater than 2. If this is the case, the mantissa is shifted one position to the right and the exponent is incremented by 1. Note that the test "if (Be≥2)" in part A of the arithmetic unit is done *after* Be has already been computed in part B during stages I, II, and III of cycle 2.

In the case of a subtraction four or five cycles are needed. Figure 5.10 shows the synchronization required for a subtraction. The first two cycles are almost identical to the first two cycles of the addition algorithm, but now the mantissas are subtracted. Cycle 3 is executed only when the difference of the mantissas is negative. The effect of cycle 3 is just to make the mantissa of the result positive. Cycle 4 is very important: the difference of two normalized mantissas can have many zeros in the first bit positions to the left. The result is normalized by shifting

Be to the left as many places as necessary (this is done with the shifter between the relay box Fd and register Bb). The number of one-bit shifts is subtracted from the exponent in part A of the processor. In cycle 5, the result is stored in the register pair <Af,Bf>.

## 5.4.3 *Multiplication*

The multiplication algorithm of the Z3 is like the one used for decimal multiplication by hand, that is, it is based on repeated additions of the multiplicand according to the individual binary digits of the multiplicand. At the beginning of the algorithm, the first argument is stored in the register pair <Af,Bf>. The second argument is stored in the register pair <Ab,Bb>. The temporal register pair <Aa,Ba> is set to zeroes. Figure 5.11 shows the microsequencing produced by the multiplication wheel of the control unit. The algorithm takes 16 cycles to run. Note that only the

| cycle | stage | exponent | mantissa |
|---|---|---|---|
| 0 | I,II,III | Ae:=Aa+Ab | |
| 1 | IV,V | Aa:=Ae, Ab:=Af | |
| | I,II,III | Ae:=Aa+Ab | if (Bf[−14]=1) then Be:=Ba+Bb<br>else Be:=Ba |
| 2 | IV,V | Aa:=Ae, Af:=0, Ab:=0 | Ba:=Be/2 |
| | I,II,III | Ae:=Aa+Ab | if (Bf[−13]=1) then Be:=Ba+Bb<br>else Be:=Ba |
| 3 | IV,V | Aa:=Ae | Ba:=Be/2 |
| | I,II,III | Ae:=Aa+Ab | if (Bf[−12]=1) then Be:=Ba+Bb<br>else Be:=Ba |
| 4 | ⋮ | ⋮ | ⋮ |
| *i* | IV,V | Aa:=Ae | Ba:=Be/2 |
| | I,II,III | Ae:=Aa+Ab | if (Bf[$i-15$]=1) then Be:=Ba+Bb<br>else Be:=Ba |
| 14 | ⋮ | ⋮ | ⋮ |
| 15 | IV,V | Aa:=Ae | Ba:=Be/2 |
| | I,II,III | if (Be $\geq$ 2) then Ab:=1<br>Ae:=Aa+Ab | if (Bf[0]=1) then Be:=Ba+Bb<br>else Be:=Ba |
| 16 | IV,V | Af:=Ae | if (Be $\geq$ 2) then Bf:=Be/2<br>else Bf:=Be<br>Bb:=0 |

**Fig. 5.11** The 16 cycles needed for the multiplication algorithm. The $i$-th bit of register Bf is denoted by Bf[$i$]. The first argument is stored in the register pair <Af,Bf> and the second in <Ab,Bb> before the operation is started

bits of the multiplicand from position −14 to position 0 are used. The exponents are added in the first cycle, and the result just loops afterward in part A of the arithmetic unit. The mantissas are handled in part B of the unit. Register Ba contains the partial result of the computation. The basic multiplication loop has the following form:

```
Ba:=Be/2
Be:=Ba + Bb×(i-th bit of Bf)
```

for $i = -14, \ldots, 0$. The partial result Be is shifted one position to the right, to produce Ba:=Be/2. This is done with the shifter connected to the relay box Fc.

The result of the multiplication is a number $1 \leq r < 4$ (for arguments within bounds). In the last cycle, there is a check to see if $r \geq 2$. If this is the case the result is shifted one position to the left and a 1 is added to the exponent of the result.

### *5.4.4 Division*

The division algorithm is similar to the multiplication algorithm, but subtraction is used repeatedly instead of addition. At the beginning of the algorithm, the dividend is stored in the register pair <Af,Bf>. The divisor is stored in the register pair <Ab,Bb>. The temporal register pair <Aa,Ba> is set to zeroes. Figure 5.12 shows the microsequencing produced by the division wheel of the control unit. The algorithm takes 18 cycles to run.

The main idea of the algorithm is very simple. The exponent of the result is obtained by subtracting the exponents of the dividend and divisor. Now for the mantissa: assume that we want to compute $x/y$ for the mantissas $x$ and $y$. Since we are dealing with normalized numbers, the first digit of the result is 1 if $x \geq y$ and zero if $x < y$. In the first case, we set the first digit of the result to 1 and compute the remainder, which is $x - y$. The remainder is divided recursively by $y$. To do this, it is shifted one position to the left, and the new result bit is stored at position $[-1]$ of register Bf (in this way nullifying the effect of the shift). If the result bit is zero, the remainder is just $x$, and the recursive division is continued as in the first case.

The basic division loop has the following form:

```
Ba:=2×Be
if (Ba−Bb ≥ 0) then Be:=Ba−Bb, Bf[i]:=1
                else  Be:=Ba         Bf[i]:=0
```

for $i = 0, \ldots, -14$. The partial result Be is shifted one position to the left to produce Ba:=2×Be. This is done with the shifter connected to the relay box Fc.

The result of the division of mantissas is a number $1/2 < r < 2$. This condition is tested in cycles 17 and 18. If $r < 1$, a 1 is subtracted from the exponent, and the result is shifted one position to the left to get a normalized number.

| cycle | stage | exponent | mantissa |
|---|---|---|---|
| 0 | I,II,III | | |
| 1 | IV,V | Aa:=Af | Ba:=Bf |
| | I,II,III | Ae:=Aa−Ab | if (Ba−Bb ≥ 0) then Be:=Ba−Bb, bt:=1<br>else Be:=Ba, bt:=0 |
| 2 | IV,V | Aa:=Ae<br>Ab:=0 | Bf:=0<br>if (bt=1) then Bf[0]:=1<br>Ba:=2×Be |
| | I,II,III | Ae:=Aa+Ab | if (Ba−Bb ≥ 0) then Be:=Ba−Bb, bt:=1<br>else Be:=Ba, bt:=0 |
| 3 | IV,V | Aa:=Ae | if (bt=1) then Bf[−1]:=1<br>Ba:=2×Be |
| | I,II,III | Ae:=Aa+Ab | if (Ba−Bb ≥ 0) then Be:=Ba−Bb, bt:=1<br>else Be:=Ba, bt:=0 |
| 4 | ⋮ | ⋮ | ⋮ |
| *i* | IV,V | Aa:=Ae | if (bt=1) then Bf[2−*i*]:=1<br>Ba:=2×Be |
| | I,II,III | Ae:=Aa+Ab | if (Ba−Bb ≥ 0) then Be:=Ba−Bb, bt:=1<br>else Be:=Ba, bt:=0 |
| 15 | ⋮ | ⋮ | ⋮ |
| 16 | IV,V | Aa:=Ae | if (bt=1) then Bf[−14]:=1<br>Ba:=2×Be |
| | I,II,III | Ae:=Aa+Ab | if (Ba−Bb ≥ 0) then Be:=Ba−Bb, bt:=1<br>else Be:=Ba, bt:=0 |
| 17 | IV,V | if (Bf[0]=0)<br>then Ab:=−1 | if (bt=1) then Bf[−15]:=1<br>Bb:=0<br>if (Bf[0]=0) then Ba:=2× Bf<br>else Ba:=Bf |
| | I,II,III | Ae:=Aa+Ab | Be:=Ba−Bb |
| 18 | IV,V | Af:=Ae | Bf=Be |

**Fig. 5.12** The 18 cycles needed for the division algorithm. The *i*-th bit of register Bf is denoted by Bf[*i*]. The dividend is stored in the register pair <Af,Bf> and the divisor in <Ab,Bb> before the operation is started

## *5.4.5 Square Root Extraction*

The square root algorithm is the jewel in the Z3's crown. Figure 5.13 shows the microsequencing required during the 20 cycles needed to compute the square root of a number. The argument for the operation is stored in the register pair <Af,Bf>. The register pair <Aa,Ba> is initialized to zeroes. The algorithm computes the square root of numbers with an even exponent. If the exponent is an odd number,

| cycle | stage | exponent | mantissa |
|---|---|---|---|
| 0 | I,II,III | | |
| 1 | IV,V | | If (Af[0]=1) then Ba:=2×Bf<br>else Ba:=Bf<br>Bb[0]:=1 |
| | I,II,III | | if (Ba−Bb ≥ 0) then Be:=Ba−Bb, bt:=1<br>else Be:=Ba, bt:=0 |
| 2 | IV,V | | Bf:=0<br>if (bt=1) then Bf[0]:=1<br>Ba:=2×Be, Bb:=2×Bf, Bf[−1]:=1 |
| | I,II,III | | if (Ba−Bb ≥ 0) then Be:=Ba−Bb, bt:=1<br>else Be:=Ba, bt:=0 |
| 3 | IV,V | | if (bt=1) then Bf[−1]:=1<br>Ba:=2×Be, Bb:=2×Bf, Bb[−2]:=1 |
| | I,II,III | | if (Ba−Bb ≥ 0) then Be:=Ba−Bb, bt:=1<br>else Be:=Ba, bt:=0 |
| 4 | ⋮ | ⋮ | ⋮ |
| $i$ | IV,V | | if (bt=1) then Bf[2−$i$]:=1<br>Ba:=2×Be, Bb:=2×Bf, Bf[1 − $i$]:=1 |
| | I,II,III | | if (Ba−Bb ≥ 0) then Be:=Ba−Bb, bt:=1<br>else Be:=Ba, bt:=0 |
| 17 | ⋮ | ⋮ | ⋮ |
| 18 | IV,V | | if (bt=1) then Bf[−16]:=1<br>Ba:=2×Be, Bb:=2×Bf |
| | I,II,III | | if (Ba−Bb ≥ 0) then Be:=Ba−Bb, bt:=1<br>else Be:=Ba, bt:=0 |
| 19 | IV,V | Aa:=Af/2 | Ba:=Bf, Bb:=0 |
| | I,II,III | Ae:=Aa+0 | Be:=Ba+Bb |
| 20 | IV,V | Af:=Ae | Bf:=Be |

**Fig. 5.13** The 20 cycles needed for the square root algorithm. The $i$-th bit of registers Bf and Af are denoted by Bf[$i$] and Af[$i$], respectively. The argument is stored in the register pair <Af,Bf> before the operation is started

the mantissa is shifted one place to the left, and the exponent is decremented by one. The final exponent (computed in cycle 19) is half this initial exponent.

The main idea of the algorithm is to reduce the square root operation to a division. If we want to compute the square root of $x$, we need a number $Q$ such that $x/Q = Q$. The result $Q$ is built sequentially by setting the $i$-th bit to 1 and then testing whether the condition $x > Q^2$ still holds. If this is not the case the $i$-th bit must be set to 0.

Assume that we have already computed from bit 0 to bit $-i+1$ of the final result. Denote by $Q_{-i+1}$ the mantissa

$$Q_{-i+1} = \mathrm{Bf}[0] \times 2^0 + \mathrm{Bf}[-1] \times 2^{-1} + \cdots + \mathrm{Bf}[-i+1]2^{-i+1}.$$

Bit $-i$ is then set to to $q_{-i}$ and it must hold that

$$x \geq Q^2_{-i} = (Q_{-i+1} + q_{-i}2^{-i})^2$$

This is true if

$$(x - Q^2_{-i+1}) - 2^{-i}q_{-i}(2Q_{-i+1} + 2^{-i}q_{-i}) \geq 0$$

Define $t_{-i}$ using the expression

$$2^{-i}t_{-i} = (x - Q^2_{-i+1})2^{-1}2^1 - 2^{-i}q_{-i}(2Q_{-i+1} + 2^{-i}q_{-i})$$

This can be written as

$$2^{-i}t_{-i} = t_{-i+1}2^{-i+1}2^{-1}2^1 - 2^{-i}q_{-i}(2Q_{-i+1} + 2^{-i}q_{-i})$$

where we have used the recursive definition $2^{-i+1}t_{-i+1} = (x - Q_{-i+1})^2$. Simplifying the last expression we finally get:

$$t_{-i} = 2t_{-i+1} - q_{-i}(2Q_{-i+1} + 2^{-i}q_{-i})$$

If $t_{-i}$ is positive for $q_{-i} = 1$, we set bit $-i$ of the final result to 1, i.e., Bf[$-i$]:=1. If $t_{-i}$ is negative, we set Bf[$-i$]:=0. The recursive computation is started with $t_0 = x$. $Q_{-i+1}$ represents at each step the partial result contained in register Bf. Bit $-i$ is tentatively set and the sign of $t_{-i}$ is tested.

The basic loop of the square root algorithm for bit $-i$ has the following form:

```
Ba:=2×Be
Bb:=2×Bf
Bb[−i]:=1
if (Ba−Bb ≥ 0) then Be:=Ba−Bb, Bf[−i]:=1
                else Be:=Ba,    Bf[−i]:=0
```

All bits of register Bf are used for the computation of the square root. If the original number lies within bounds, the result is also within bounds.

### 5.4.6 *Read and Display Instructions*

The two most complex instructions of the Z3 are those related to the input and output of decimal numbers. A decimal number of four digits entered via the keyboard is first converted into binary. This is done by reading each digit sequentially, transforming it into a binary number, and storing it in the bits Ba[−10], Ba[−11], Ba[−12], and Ba[−13] of register Ba. The number in register Ba is multiplied by 10 and the procedure is repeated for the other digits. After four iterations, the decimal input has been transformed to a binary number (the exponent is adjusted to the correct value). The difficult part is handling the exponent. If the exponent $e$ is positive, the mantissa has to be multiplied $e$ times with 10. If it is negative, it must be multiplied $|e|$ times with 0.1. Multiplying with 10 is relatively easy: the mantissa in Be can be shifted one bit to the left and then stored in Ba (that is Ba:=2×Be). At the same time, Be can be shifted 3 places to the left and can be stored in Bb (that is Bb:=8×Be). The addition of Ba and Bb then provides the desired result: the multiplication of the original number in Be with the constant 10. The process takes 4 cycles for multiplication, that is 32 cycles for the decimal exponent +8. Since a read operation needs a minimum of 9 cycles, this means that a decimal number with exponent +8 is read in 41 cycles.

In the case of negative exponents, multiplication with the constant 0.1 is performed using both shifters and the adders. This multiplication is somewhat more complex because 0.1 is a periodic number in the binary system. The description of the microsequencing used would take us too far away from the main topics, so we omit it here.

The display instruction works by multiplying or dividing iteratively by 10. If the binary exponent of the number in register R1 is positive, the number is multiplied with 0.1 as many times as needed to make the binary exponent equal to 2 and until the first left four bits of register Bf contain a number between 0 and 9 (0000 and 1001). This is the decimal digit that can be displayed in the next column of the output panel. The number is subtracted from the mantissa in Bf, and the process continues for the following digits. If the binary exponent of the number in register R1 is negative, the process is similar, but multiplications with the constant 10 are used.

The tables of microoperations needed for binary-decimal and decimal-binary conversion can be found in Rojas (1998).

## 5.5 Complete Architecture of the Z3

We are able to understand the detailed diagram of the Z3 shown in Fig. 5.14. We see some of the components that were discussed in the previous sections.

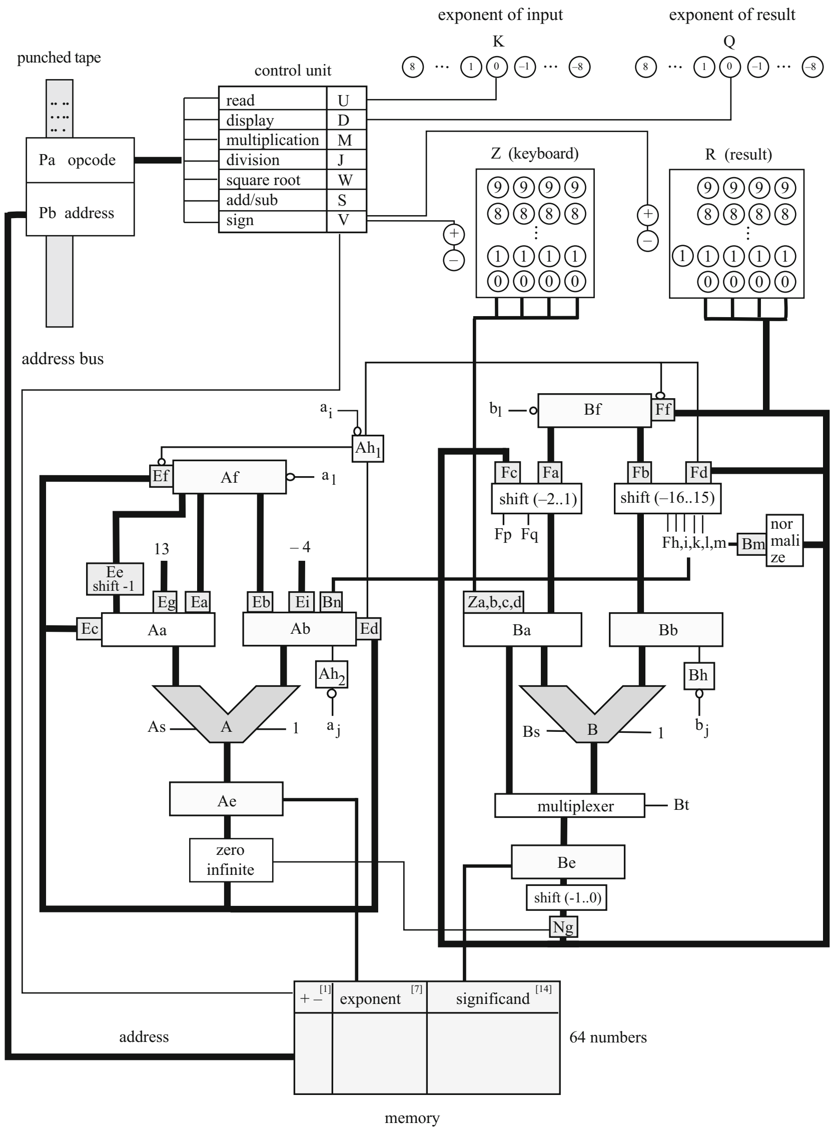


**Fig. 5.14** The complete architecture of the Z3

The control unit and I/O panels have been discussed already. Note that the four decimal digits of the input keyboard are transferred into register Ba using the relay boxes Za, Zb, Zc, and Zd, which are activated one after the other.

The relay boxes Eg and Ei are used to directly set some useful constants into the exponent registers (+13 and −4). The shifter Ee between register Af and register Aa is used for the square root algorithm. The exponent of the result (Aa) becomes half the exponent (Af) of the original number.

$\mathrm{Ah}_1$ is a relay acting as a flip-flop. When it is set to 0, the register pair <Af,Bf> is accessed by load operations. When it is set to 1, the register pair <Ab,Bb> is accessed. This relay is reset to 0 by the control line $\mathrm{a}_i$. The control lines $\mathrm{a}_l$, $\mathrm{a}_j$, $\mathrm{b}_l$, and $\mathrm{b}_j$ are used to clear the registers Af, Ab, Bf, and Bb when needed.

The box labeled "zero, infinite" below Ae represents the circuitry for exception handling. They snoop permanently on the data bus (results of operations and data from memory) and raise the corresponding exception flags when needed. The shifter below Be is used to displace the mantissa one bit to the right. This provides the normalization needed for the mantissa whenever Be$\geq$2.

Fp and Fq are the relays that control the number and direction of one-bit shifts in the shifter below the relay boxes Fc and Fa. Fh, Fi, Fk, Fl, and Fm have the same function relative to the other shifter. Using these five bits, the numbers between $-16$ and 15 can be represented, and this is also the range of the second shifter. When such a shift is performed, the number represented by the relays Fh to Fm is transferred through the relay box Bn to register Ab to modify the exponent of the result. If the number is shifted 10 positions to the left, then +10 is subtracted from the exponent of the result. Such drastic shifts are needed mostly after subtractions.

Take another look at the diagram of the Z3. It all makes sense now and looks as conventional as any modern small floating-point processor. It is indeed amazing how Konrad Zuse was able to find the adequate architecture right from the start, with one notable exception, the non-inclusion of the conditional branch. The Z3 processor uses only 600 relays; the memory required three times as many. By having to optimize the design, by having to save hardware everywhere, Zuse was *forced* to think and rethink the logic structure of his machine. He did not have the luxury of the almost unlimited funding provided by the US military for the development of the ENIAC or by IBM for the Mark I. He was all alone, and while this may have worked to his advantage from a conceptual standpoint, it may also have worked to his disadvantage in terms of the negligible impact that the Z1 and Z3 had on the emerging American computer industry after the war (Stern 1981).

## References

Czauderna, K.H. 1979. Konrad Zuse, der Weg zu seinem Computer Z3. In *Berichte der Gesellschaft für Mathematik und Datenverarbeitung*. Munich: Oldenbourg Verlag.

Knuth, D.E. 1981. *The Art of Computer Programming – Seminumerical Algorithms*, Vol. 2. Boston: Addison-Wesley Professional.

Koren, I. 1993. *Computer Arithmetic Algorithms*. Englewood Cliffs: Prentice Hall.

Range, J. 2016. Validierung und Visualisierung des Mikrocodes der Z3 von Konrad Zuse. Master's Thesis, Humboldt Universität Berlin.

Rojas, R. 1997. Konrad Zuse's legacy: The architecture of the Z1 and Z3. *IEEE Annals of the History of Computing* 19(2): 5–16. https://doi.org/10.1109/85.586067.

Rojas, R. 1998. *Die Rechenmaschinen von Konrad Zuse*. Berlin: Springer. https://doi.org/10.1007/978-3-642-71944-8.

Stern, N. 1981. *From ENIAC to UNIVAC*. Bedford: Digital Press.

Zuse, K. 1941. *Patentanmeldung Z-391*. Berlin: German Patent Office.

# Chapter 6
# How to Make Zuse's Z3 a Universal Computer

*The computing machine Z3, built by Konrad Zuse from 1938 to 1941, could only execute fixed sequences of floating-point arithmetical operations (addition, subtraction, multiplication, division, and square root) encoded in a punched tape. An interesting question to ask, from the viewpoint of the history of computing, is whether or not these operations are sufficient for universal computation.*[1]

In this chapter, we show that in fact a single program loop containing these arithmetic instructions can simulate any Turing machine whose tape has a a given finite size. This is done by simulating conditional branching and indirect addressing by purely arithmetic means. Zuse's Z3 is thus, at least in principle, as universal as today's computers that have a limited addressing space.

## 6.1 Universal Machines and Single Loops

*No One Has Ever Built a Universal Computer* The reason is that a universal computer consists, in theory, of a fixed processor and a memory of unlimited size. This is the case of Turing machines, which have infinitely long tapes. Also, in the theory of general recursive functions, there are a small set of rules and some predefined functions, but there is no upper bound on the size of intermediate results. Modern computers are therefore only *potentially* universal: they can perform any computation that a Turing machine with a tape of bounded length can perform. If more storage is required, more can be added without modifying the processor (provided that the additional memory is still addressable).

It is the purpose of this chapter to show that Konrad Zuse's Z3, a computing automaton built in Berlin between 1938 and 1941, could *in principle* be programmed

[1] This chapter is based on Rojas (1998b).

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_6

like any other modern computer. This is a rather curious result, since the Z3 can only compute sequences of arithmetic operations (addition, subtraction, multiplication, and division) stored in a punched tape. There is no conditional branching. Since both ends of the punched tape can be joined together, the Z3 is a machine that can repeatedly execute *a single loop* of arithmetic operations acting on numbers stored in memory.

It is well known that any computer program that contains conditional branches and the usual instructions of imperative languages can be programmed with a single WHILE loop (Harel 1980). Also, all conditional branches can be eliminated from the loop (Ibarra et al. 1983). I showed in Rojas (1996a) that if the Z3 is extended with indirect addressing, it can simulate a Turing machine. We will adopt the techniques used in those papers to show that complex computations can be simulated by a single program loop of a machine able to compute the four basic arithmetic operations.

Our computing model is the following: there are memory locations available that will be denoted by lowercase letters. We can only refer explicitly to memory addresses (there is no indirect addressing). Initially (for the sake of simplicity), we will limit our programs for the Z3 to a language containing only statements of the form

$$a = b \ \mathsf{op}\ c,$$

where op stands for one of the four basic arithmetic operations. Any statement of this form can be "compiled" using the two registers of the Z3 and four assembler instructions (which load the two argument registers in the appropriate order):

```
LOAD b
LOAD c
op
STORE a
```

The store operation implicitly refers to the first register (accumulator) of the processor. All computations are performed with floating-point numbers. The mantissa has a precision of 16 bits for its fractional part. The Z3 uses normalized floating-point numbers (i.e., with a mantissa $m$ such that $1 \leq m < 2$). The special case of a zero mantissa is handled with a special code (like in the IEEE standard). There is also a "halt" instruction in the Z3 (when a result is displayed on the console the machine stops).

## 6.2 Simulating Branches

Here we show how to simulate the operation of a CASE statement using a technique introduced in Ibarra et al. (1983) and used previously for the development of the theory of recursive functions (Péter 1967). Define the state of the machine as the state of its memory. Assume that in a program $P$, there are $n$ consecutive sections of code $P_1, \dots, P_n$ and that the variable $z \in \{1, 2, \dots, n\}$ is used to select the section that should perform the computation we are interested in. The general strategy is to execute *all n* sections of code, one after the other, but we will allow only the $z$-th section to modify the memory contents. To implement this idea, we transform each section of code $P_j$ into equivalent code $P_j'$ according to the following rule: At the beginning of each section $P_j$ a comparison is made and if $z = j$ the auxiliary variable $t$ is set to 0; otherwise, it is set to 1. The variable $t$ can be interpreted as a flag for the "selected section," since it will be only zero in $P_z$. Now all the original statements in the program $P_1, \dots, P_n$ of the form $a = b \text{ op } c$ are transformed into

$$a = a \cdot t + (b \text{ op } c) \cdot (1 - t)$$

and are compiled accordingly. Therefore, the state of variable $a$ will not be modified unless the computation occurs within the $z$-th code section. When all statements have been transformed in this way and the appropriate initialization of $t$ has been inserted at the beginning of each code section, we can execute the transformed program $P_1', \dots, P_n'$ from beginning to end. Most of the computations are superfluous, since we execute all sections of code, but only $P_z'$ modifies the memory, as is expected from a CASE statement.

All we need to do now is to show that it is indeed possible to do the calculation

$$\text{if } (z = j) \text{ then } t = 0 \text{ else } t = 1.$$

where $z$ and $j$ are integers. The simplest approach is to use the binary representation of $z$, which is stored using the auxiliary variables $z_1, z_2, \dots, z_m$. The number $m$ of bits used is fixed in advance according to the total number $n$ of sections of code that have to be selected. For each code section $j$, the *complement* of the binary representation of $j$ is stored in the variables $c_1^{(j)}, c_2^{(j)}, \dots, c_m^{(j)}$. The following arithmetic calculation at the beginning of each code section $j$ sets the variable $t$ to its correct value:

$$t = 1 - \left[\left(c_1^{(j)} - z_1\right)\left(c_2^{(j)} - z_2\right)\cdots\left(c_m^{(j)} - z_m\right)\right]^2$$

The variable $t$ is set to zero only if all factors in the expression are $\pm 1$, but this is only the case when $z = j$.

It should now be clear that an unconditional jump to code section $j$ can be programmed in a section of code $P_i'$ by setting the next value of $z$ to $j$ (i.e. their binary representations) at the end of $P_i'$ and going back to the beginning of the transformed program $P_1', \dots, P_n'$. This is done by storing the program in a single loop of punched tape that is used repeatedly.

In this and other programs, all necessary constants (the binary representations of the code section numbers) can be precomputed and stored before we start the CASE statement.

This proves that with the computing model of the Z3 we can, in principle, perform any computation that any other computer with a bounded memory can perform. In Rojas (1998b), I also show how to simulate a Turing machine using the approach described in this section.

## 6.3 Halting the Computation

The attentive reader will have noticed that the master loop of computations never stops. Algorithms, however, must stop after a finite number of steps. Fortunately, the Z3 has an additional feature that provides the solution to this problem.

Whenever an undefined operation is performed, the Z3 stops and a lamp lights up on the console. This is the case, for example, for the operation $0/0$. Thus we define a state $Q_0 = 0$ of the computation as the "halting state," with $Q_0 = 1$ as the initial value. The computation $0/Q_0$ is performed at the beginning of the master loop. If the simulation reaches state $Q_0 = 0$ the machine will stop. In the algorithm being executed, we set $Q_0$ to this value when we want to stop the master loop.

If Zuse had not thought of trapping undefined operations, we would have been unable to stop the master loop. One possible way out in that case would be to consider those cycles in which nothing is altered as the "halting state" of the machine, but the human operator would have some problems identifying this situation.

## 6.4 Conclusions

The main result discussed in this chapter is intriguing because it looks so artificial. From the theoretical point of view, it is interesting to see that limited precision arithmetic embedded in a WHILE loop can compute anything modern computers can. It could be argued that whenever we expand the memory (to accommodate more tape positions for a simulated Turing machine), the program in the punched tape has to be expanded as well (to cover the new memory addresses), and the number of bits ($m$) used to identify the code sections has to be increased. If we think of the punched tape as part of the processor (when simulating a Universal Turing Machine), then we are extending the processor when we enlarge the program in the punched tape. Although this is undesirable, in real computers, there is also a limit to the size of the memory we can manage (given by the addressable space, i.e., the number of bits in the address registers). If we expand the memory, we need more addressing bits and the processor may have to be expanded (going, for example, from 16-bit to 32-bit registers).

The result shown in this chapter seems counterintuitive until we realize that operations like multiplication and division are iterative computations in which branching decisions are taken by the hardware. The conditional branchings we need are *embedded* in these arithmetical operations, and the whole purpose of the transformations used is to lift the branches up from the hardware in which they are buried to the software level, so that we can control the program flow. The whole magic of the transformation consists of making the hardware branchings visible to the programmer.

A possible criticism of the approach discussed here could be that it greatly slows down the computations. From a purely theoretical point of view, this is irrelevant unless we introduce a complexity measure and demand a simulation of Turing machines capable of running without an exponential slowdown. From a practical point of view, obviously nobody would program the Z3 as we just described, in the same way that nobody solves industrial problems using Turing machines.

We can therefore say that, from an *abstract theoretical perspective*, the computing model of the Z3 is equivalent to the computing model of today's computers. From a practical perspective, and in the way the Z3 was really programmed, it was not equivalent to modern computers. However, it is clear to me from the study of Zuse's unpublished manuscripts (held in the archive of Deutsches Museum in Munich) that after completing the Z3, he realized (between 1943 and 1945) that he could "lift" the decisions taken in hardware to the software level, to give the programmer full control of the computation. His plans for a "logistic machine," so elementary that the instruction set consisted exclusively of Boolean operations is discussed in Chap. 11.

## References

Harel, D. 1980 On Folk Theorems. *Communications of the ACM* 23(7): 379–389. https://doi.org/10.1145/358886.358892.

Ibarra, O.H., S. Moran, L.E. Rosier. 1983. On the Control Power of Integer Division. *Theoretical Computer Science* 24(1): 35–52. https://doi.org/10.1016/0304-3975(83)90129-9.

Péter, R. 1967. *Recursive Functions*. New York: Academic Press.

Rojas, R. 1996a. Conditional Branching is not Necessary for Universal Computation in von Neumann Computers. *Journal of Universal Computer Science* 2(11): 756–767. https://doi.org/10.3217/jucs-002-11-0756.

Rojas, R. 1998b. How to make Konrad Zuse's Z3 a universal computer. *IEEE Annals of the History of Computing* 20(3): 51–54. https://doi.org/10.1109/85.707574.

# Chapter 7
# The S1 and S2: Zuse's Work for the German Military 1941–1945

*This chapter describes the architecture and operation of the S1 and S2 computing machines built by Konrad Zuse between 1941 and 1945. Both were special-purpose devices for computing aerodynamic corrections to the wings of radio-controlled flying bombs, that is, the Henschel Flugzeug-Werke's HS-293 and HS-294, precursors of modern cruise missiles.*

*The S1 and S2 were fed with the result of dozens of measurements of shape parameters from the wings of a bomb. The data were processed using an accumulating loop consisting of one iteration for each group of four measurements. Several additions/subtractions and a few multiplications were executed in each iteration. The final result consisted of just three numbers, which allowed the technicians to adjust the left, right, and rear wings in order to achieve smooth flight. All computations and constants were hardwired in the S1 and S2 using rotary switches and relays. The S2 was more advanced than the S1: it eliminated manual data entry by automatically reading the values provided by digitally controlled measuring instruments that transformed an analog into a digital value. This chapter concludes with a discussion of the effect of the war on Zuse's work and some related questions.*

## 7.1 Introduction

Konrad Zuse's contribution to the development of the first German computers has been extensively documented by now. The two most important early machines built by Zuse until 1941 in Berlin were the Z1 and the Z3, which have been reconstructed for German museums (Rojas et al. 2005). However, little is known about two other important machines designed by him, the S1 and S2, custom-made for the military. Zuse's official work during most of the war was, in fact, as an engineer for the Henschel Flugzeug-Werke, a company producing airplanes and other kinds of armament during World War II. During the years in which the S1 and S2 were

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_7

built, Zuse was jump-starting his computer company with a contract for developing the Z4, a more advanced variation of the Z3, which would later become the first computer in operation on the European continent (that is, without considering the British computers). From 1941 to 1945, Zuse's work for Henschel resembled more the activity of a consultant, who would provide solutions for pressing problems devoting only one and a half days a week to the company itself.

This chapter contains the first detailed description of the special-purpose S1 and S2 machines (Zuse 1942a, 1944a). They are interesting from the point of view of the history of computing since the S2 was one of the first attempts at implementing automatic measurement and control in a factory setting. Also, the exact relationship between Zuse's pioneering ideas and his work for the military establishment has been the source of many legends. This chapter provides some useful information about this aspect of Zuse's life. Although the existence of the S1 and S2 is well known—Zuse himself mentions them briefly in two pages of his memoirs (Zuse 1970)—few details about their architecture and mode of operation have been published. The information was acquired through a detailed study and reverse engineering of the only remaining information, that is, the circuit diagrams of the S1 (Zuse 1942b) and S2 (Zuse 1944a), and a patent application for the S2 analog-to-digital converter (Zuse 1953b).

When World War II started, Zuse had already built his first mechanical computing machine (the Z1) and was working on the relay version of the same architecture, the Z3. He was called to the Eastern Front immediately, but through his connections with the scientific establishment, and his friends at the *Technische Hochschule Berlin*, where he had studied civil engineering, he could arrange for a discharge that brought him back to Berlin in March 1940 as an engineer for the Henschel Flugzeug-Werke. His most important "business connection" was probably with this company, where he had worked as a static forces analyst for 1 year after finishing his studies in 1935. The Henschel Flugzeug-Werke produced locomotives, autos, and armaments. In 1933, they had started producing airplanes at a new location in the south of Berlin and, when the war started, flying bombs.

Back from the front in 1940, Zuse continued working on computing machines, albeit by night and during the weekends, until he could arrange to finance the Z3 that same year. The Z3 became operational in May 1941. One month earlier, Zuse had founded the *Zuse Ingenieurbüro und Apparatebau* company. However, during the summer of 1941, he was called to the front. Prof. Herbert Wagner asked immediately for his discharge and reassignment to the Henschel Flugzeug-Werke. A few days later, Konrad Zuse was back from the army, for the second time, and would remain in Berlin until the final months of the war.

Herbert Wagner, a professor at the Technical University until 1938, was a maverick engineer, who had worked at the Junkers factories in Dessau developing airplanes from 1938 to 1940. His unconventional research methods led to friction with the Junkers management. He moved to the Henschel Flugzeug-Werke in April 1940, where he became the head of Section F in charge of developing remote-controlled self-propelled flying bombs. This was the section Zuse was assigned to after each of his two discharges from the army.

## 7.2 The HS-293 Flying Bomb

The HS-293 flying bomb was developed by Wagner's team. The bomb was released by an airplane, and an operator would follow the bomb's trajectory by sight. The operator had a small joystick that allowed him to control ailerons and therefore the bomb's flight path. The commands were sent by radio. Other variations of the bomb used cables unrolling as the bomb fell, to avoid radio jamming. There were many types of HS-293 bomb, and, later on, the HS-294, but such details are not important for our topic.

Figure 7.1 shows a picture of a HS-293 being tested. The bomb weighed almost a ton and had a wingspan of 3.1 meters. The propulsion rocket burned liquid fuel. Figure 7.2 shows a drawing of the main components of the bomb. The rocket propulsion is located in the lower part. The rest of the bomb resembles a miniature airplane with one aileron in each wing and in the tail.

The first prototypes of the HS-293 were built with expensive wings manufactured to high precision. It became necessary to use simpler wings coated with aluminum skin, as airplanes are built today. However, the shape of those aluminum wings was not perfect, and deviations from the ideal form could lead to vibrations in flight, or unbalanced lift, making the bomb uncontrollable for the operator. Wagner's solution was to measure the deviation of the wing's shape from the ideal form at dozens of

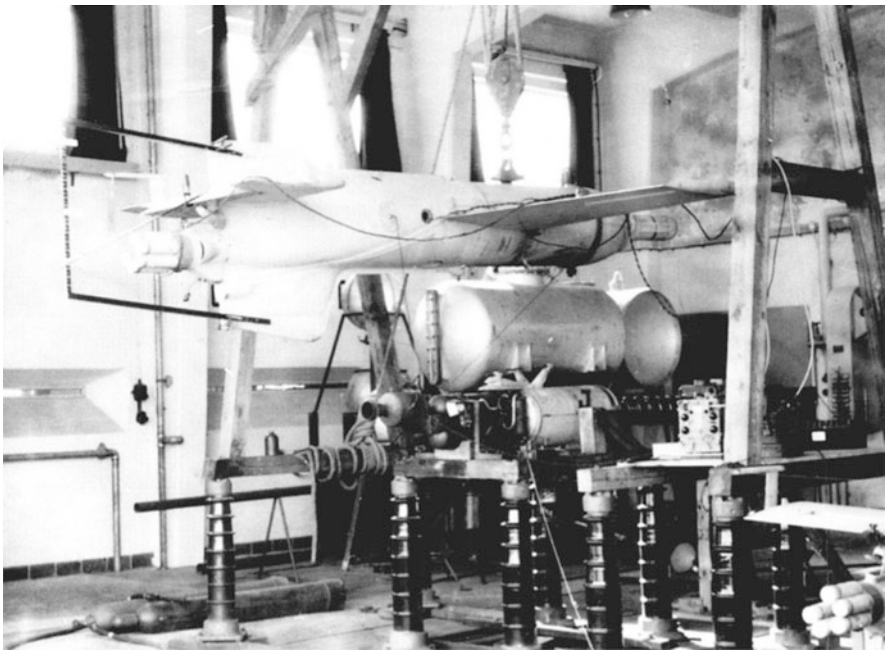


**Fig. 7.1** A HS-293 flying bomb in a production test bed

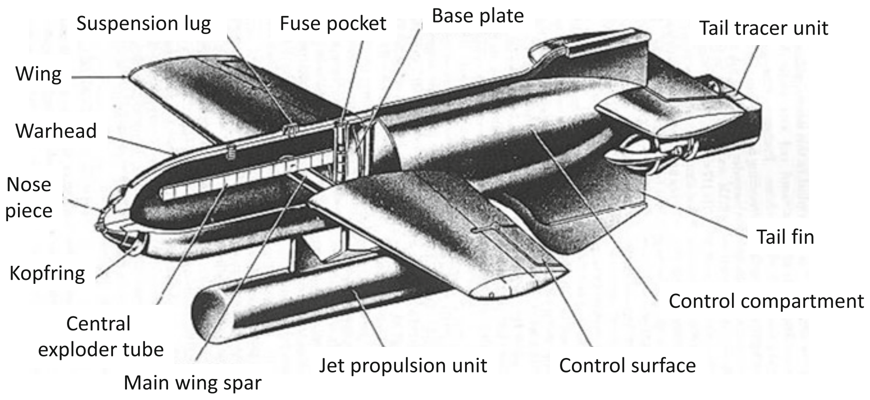


**Fig. 7.2** Drawing of the HS-293 flying bomb (Image: Wikimedia Commons)

points and, using this data, compute a repositioning of the wings' angle of attack to compensate for the fabrication errors. This allowed the Henschel Flugzeug-Werke to produce controllable flying bombs at lower cost.

After the wing shape measurements were taken, operators working in two shifts made all necessary computations using desktop mechanical calculators. The computations could consume dozens of man-hours. Therefore, computing was a major bottleneck for the production of flying bombs.

Around the end of 1941, Konrad Zuse proposed to Wagner to build a machine capable of computing the corrections automatically. For this, he derived the logic circuits from the completed Z3. The result was the HS-1 machine, which became operational in 1942 and was used without interruption until its destruction in 1944 during a bombing raid. Figure 7.3 shows a sketch, drawn by Zuse after the war, of the shape of the flying bomb and the approximate location of the places at which the shape measurements were made (above and below the wings). The positions were selected along lines parallel to the central axis and at symmetrical distances from it. The measurements were collected in groups of four, according to the following scheme:

| Measurement | Position |
|---|---|
| *a* | Left wing (above) |
| *b* | Left wing (below) |
| *c* | Right wing (above) |
| *d* | Right wing (below) |

In the case of the S1, 24 groups of four measurements were made manually, that is, 96 measurements in total. Later, for the S2, the number of measurement sites was increased since the machine itself could automatically control the measurement instruments.

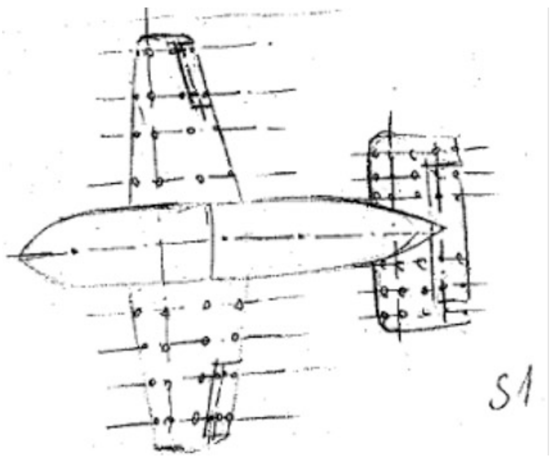


**Fig. 7.3** Sketch by Konrad Zuse of the HS-293 bomb and the position of control points for measuring the shape of the wings (Zuse 1942a, 1953a)

Once the measurements had been taken, three quantities were computed:

$$Hl = \sum_{i=1}^{n} (a_i + b_i - c_i - d_i) h_i$$

$$Fl_1 = \sum_{i=1}^{m} (a_i + b_i - c_i - d_i) f_i$$

$$Fl_2 = \sum_{i=1}^{m} (a_i - b_i - c_i + d_i) g_i$$

where the index $i$ runs from 1 to $m$ (the $m$ groups of four symmetrical measurements for the frontal wings), and from 1 to $n$ (the $n$ groups of four symmetrical measurements for the tail wing), where $n + m = 24$, and, $h_i$, $g_i$, and $f_i$ are constants providing the relative effect of a wing inaccuracy on the lift. The $h_i$, $g_i$, and $f_i$ constants are given for each type of wing and do not have to be recomputed every time. $Hl$, $Fl_1$, and $Fl_2$ represent the total inaccuracy and its effect on the rear, left, and right wings.

Once all the measurements had been processed, the final results were computed as follows:

$$\text{If} \quad (Hl < 1.0) \quad \text{then}$$

$$\begin{aligned} \delta_h &= 0 \\ \delta_l &= Fl_1 + Fl_2 - \delta - Hl \\ \delta_r &= Fl_1 - Fl_2 + \delta - Hl \end{aligned}$$

otherwise

$$\begin{aligned}\delta_h &= 0.53Hl\\ \delta_l &= Fl_1 + Fl_2 - \delta\\ \delta_r &= Fl_1 - Fl_2 + \delta\end{aligned}$$

where $\delta$ is a given constant for the measuring instruments. The values $\delta_h$, $\delta_l$, and $\delta_r$ were then used by the technicians to adjust the wings of the bomb. The only operations needed for the whole computation are addition, subtraction, and multiplication. The first two were hardwired in the addition unit. Multiplication was performed by shifting partial results and adding them.

The S1 computation consisted therefore of a main loop with 24 iterations. In each iteration, a summand in the summations for Fl1 and Fl2, or one in the summation for Hl, was computed and accumulated. Once data entry was finished, the user would request the computation of the final results and could read each of them on a decimal lamp array.

## 7.3 Block Architecture of the S1

As mentioned before, Konrad Zuse used a subset of the Z3 circuits to build the S1. The machine used binary logic and binary numbers, and it was built with relays and rotary switches. No program tape or any kind of "software" was used: all calculations were hardwired, and the user entered the data by by pressing a few buttons and reading the results at the end. The memory was rather limited: only seven memory cells with 15 bits (the memory cells 1 and 2, for temporary values, had only 10 bits). The numbers were handled as fixed-point numbers with five bits to the left of the decimal point and 10 bits to the right. A console was used for data entry, control, and to read out the results. Figure 7.4 shows a sketch of the S1 drawn by Zuse (left) and Fig. 7.5 is the only known photograph of the single machine built in 1942 (right).

In the photograph, a white rectangle is visible on top of the console, on the lower left. It is a lamp array used to keep track of the data entry (24 by 4 values) and prompt the operator for the next value needed. The data entry decimal keyboard is situated immediately to the right of the lamp array. The upper part, on the right, contains the decimal output display.

The S1 was a microprogrammed machine. No primitive assembler instructions were provided (in the sense of modern machine code). The data flow inside the machine was controlled by a few relays. Rotary switches advanced one position every cycle and opened or closed data paths as needed. Zuse hardwired all phases of the computation so that the machine had no other use than the intended one. The S1 could not even add two numbers entered ad-hoc through the keyboard. It could only execute its fundamental wing-correction loop, read decimal numbers from the

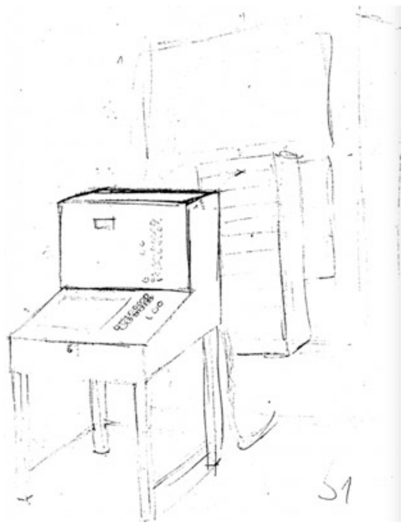


**Fig. 7.4** Sketch of the S1 by Konrad Zuse (Image: Konrad Zuse Internet Archive, zuse.zib.de)

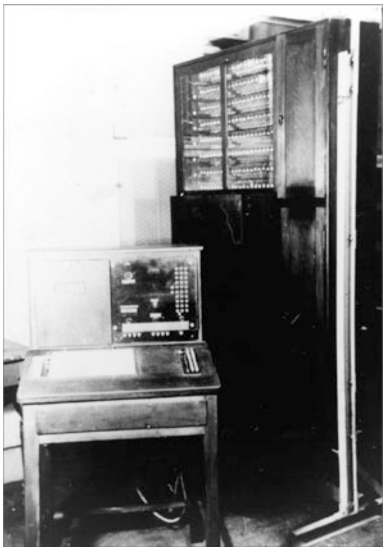


**Fig. 7.5** The only surviving photograph of the S1. The console is in front (Image: Konrad Zuse Internet Archive, zuse.zib.de)

keyboard, and store them in memory as binary numbers, or read binary numbers from memory and display them as decimal numbers in the console. According to Konrad Zuse, the machine was built using 600 relays.

Figure 7.6 shows the block architecture of the S1. The shape measurements were entered in decimal form, using the keyboard. The precision of the input was limited to three decimal digits, but a keyboard extension could be used to have two more decimal digits, for a total of five (around 14 bits of precision). If the input number was negative, the minus button was pressed. The output could be read from a lamp display shown on the right of Fig. 7.6. Each decimal digit of the data entry keyboard

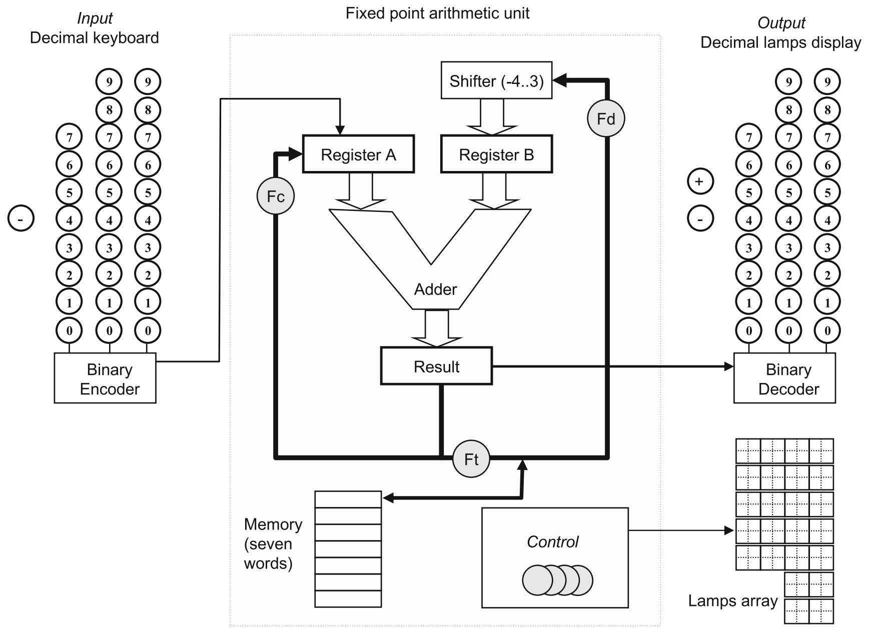


**Fig. 7.6** Block architecture of the S1

was encoded using four bits and passed to register A, one digit in one cycle. The highest significant digit was multiplied by 10 (as explained below), and the result was added to the next digit. The result was multiplied by 10, and so on.

The S1 had two registers, whose contents were added in one machine cycle. The addition circuit was identical to the circuit of the Z3, i.e., it was based on carry-look-ahead addition. An addition could be performed in one cycle. The result of an addition could be reloaded to register A or register B. A shifter allowed the machine to displace the number being loaded onto register B by up to three places to the left or up to four places to the right. Therefore, it was possible to multiply a number by 8, 4, or 2, just by shifting to the left, or divide it by 2, 4, 8, or 16, by shifting to the right. The Fc, Ft, and Fd relays closed or opened the data buses shown in the diagram. The memory contained seven words. A control unit (consisting of chains of relays and many rotary switches for the microprogramming) coordinated the whole machine. An array of 96 (four times 24) lamps was lit sequentially in order to prompt the operator for the next measurement needed in the calculation.

As can be seen, the main differences between the S1 and the Z3 were the lack of floating-point internal representation, the scale of the machine, and the fact that no program tape for software was used, since the S1 was special-purpose.

## 7.4 Operation of the S1

The architecture of the S1 can be more easily understood by looking at the sequence of steps followed by the operator of the machine. Before the operator started entering numbers, all measurements were taken and written on a report sheet, which the operator probably let stand on top of the console, to the left, a part of the device free of lamps and buttons.

The operator started the machine by pressing the button labeled Tgl. The array of lamps on the lower left side of the console consists of 24 fields subdivided into four rectangles (see Fig. 7.7 ). The 24 fields represent the 24 wing measurement positions (each with the four associated measurements $a$, $b$, $c$, $d$, as described above). If the "$a$" measurement for the first position was needed, the "$a$" subfield in the first field would be illuminated, the "$b$" subfield for the $b$ measurement, and so on. When the machine stopped, waiting for input, the operator keyed in the next requested

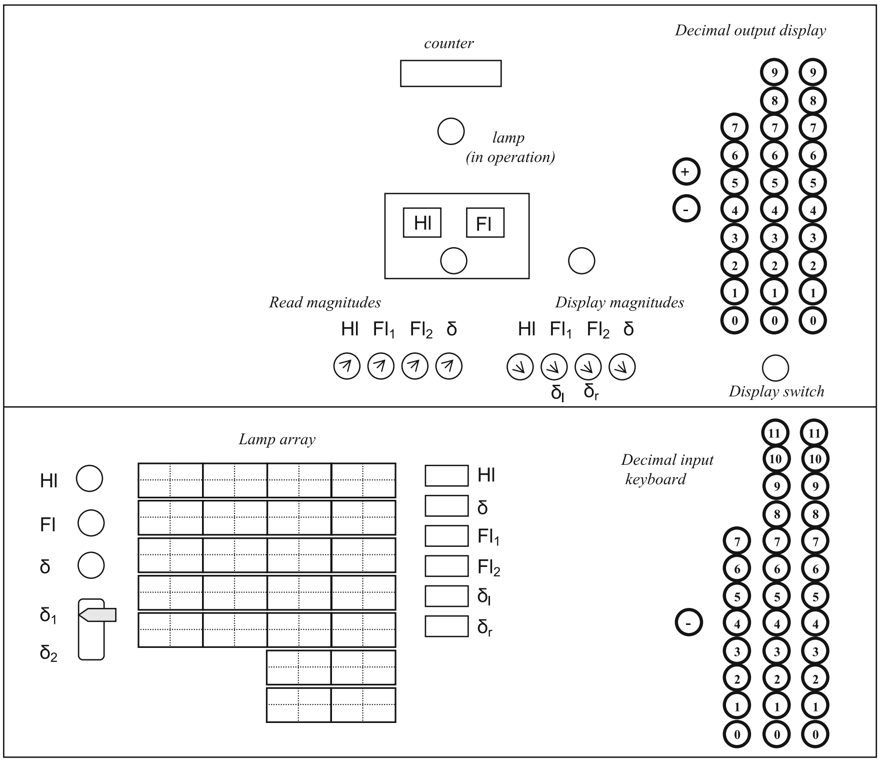


**Fig. 7.7** Diagram of the keyboard of the S1. It is not clear why the decimal keyboard included buttons for 10 and 11 in two of the keyboard columns maybe just for convenience. Rectangles correspond to lamps and circles to buttons, except for the decimal output display lamps (upper right) (Zuse 1942a)

measurement using the decimal keyboard on the lower right side of the console. Then the computation was restarted by pressing the Tgl button. The S1 went on computing until the next number was needed. It would then stop again and prompt the operator to enter the data (switching on the next lamp in the lamp array).

As mentioned above, the data entries for the left and right wings were handled separately from the data for the rear wing (the $Hl$ sum was computed using only the measurements of the rear wing). Apparently, since it is not documented, the operator pressed the $Hl$ key on the lower left side of the console as long as such data were being entered. When the measurements for the wings had been entered, the operator pressed the $Fl$ key (the keys were mutually exclusive). Once all 96 numbers had been entered, the operator would press the $\delta$ key, and $\delta_1$ and $\delta_2$ were computed, according to the position of the switch on the console. The results were stored in memory. Since the machine had only seven memory addresses, $\delta_1$ and $\delta_2$ were stored write-erasing $Fl_1$ and $Fl_2$.

The operator could then read the results by pressing the keys with an arrow pointing down. It was possible to read $Hl$, $Fl_1$, $Fl_2$, and $\delta$ (before the final computations were made), and $\delta_1$, $\delta_2$, after the $\delta$ computations had been performed (Fig. 7.8). The result appeared on the decimal display. Only three decimal digits were shown.

The keys with the upward arrow, allow the operator to load values of $Hl$, $Fl_1$, $Fl_2$, and $\delta$, directly from a previous computation, probably for testing the machine, or for confirming a previous result. Those keys read a decimal number, transform them into binary, and store it in memory.

Some lamps on the console (rectangles labeled $Hl$, $Fl_1$, etc) allowed the operator to follow the state of the computations and see which quantities were being computed at any given moment.

Summarizing: (1) the operator would start the machine, (2) enter 96 numbers one after the other, signaling in between the change from rear wing to frontal wings measurements, (3) start the delta computations at the end, and (4) request the final results by pressing the read-out buttons. Decimal to binary conversion was handled as in the Z3. The highest decimal digit from the keyboard was read into register A (encoded as four-bit numbers). Register A was added with zero, and the result was stored back to register A and to register B, but shifted two places

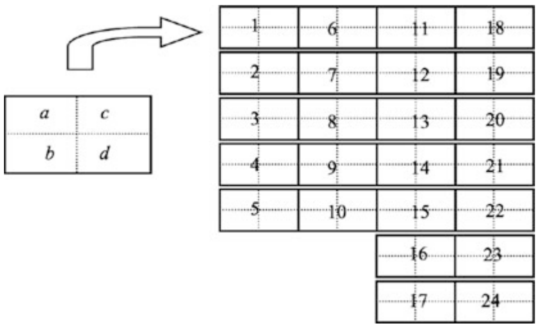


**Fig. 7.8** Numbers assigned to the 24 fields in the lamp array, and the four subfields corresponding to each field

to the left (multiplication by four). A new addition of both registers, storing back the result to register B shifting it one place to the left (multiplication by two), led to register B holding the original decimal digit multiplied by ten. The procedure was repeated with the next lower decimal digit and so on, until all three decimal digits had been processed. Binary to decimal conversion was handled similarly, but in reverse, and also in the same way as done in the Z3. The S1 performed 96 iterations. In each iteration, three additions/subtractions were needed, as well as one multiplication. Multiplications were performed in an ad-hoc manner, by shifting and adding the multiplicand as needed. The multiplication constants were hardwired in the machine. Rotary switches selected the constant used for multiplication at each iteration.

An example is illustrative of the way all operations were microprogrammed in the machine and of the coding technique used by Konrad Zuse. Before the final computation of the delta corrections, the state of the memory, after all data have been entered, is the following:

| Address | Contents |
|---|---|
| 3 | $Fl_1$ |
| 4 | $Fl_2$ |
| 5 | $Hl$ |
| 6 | $\delta$ |

The computations and register transfers in 12 cycles are shown in Table 7.1.

The first four columns describe the operations executed in 12 cycles. Ra, Rb denote register A and register B. Re denotes the result of one addition or subtraction.

**Table 7.1** Sequence of operations for computing the corrections to the wings. $Fl_1$, $Fl_2$, $H$, and $\delta$ are stored in memory cells 3, 5, 4, and 6

| Description of the operations | | | | Control signals | | | | |
|---|---|---|---|---|---|---|---|---|
| | Mnemonics | Load/Store | Result | +/− | | | Address | r/w |
| 1 | $Fl_1 - H$ | C3→Rb | | | Fp6 | | C3 | read |
| 2 | | C5→Rb | Re→Ra | sub | | Fd | C5 | read |
| 3 | | Re→C5 | | | Fp3 | | C5 | |
| 4 | $Fl_2 - \delta$ | C4→Rb | | | Fp6 | | C4 | read |
| 5 | | C6→Rb | Re→Ra | sub | | Fd | C6 | read |
| 6 | | Re→C4 | | | Fp3 | | C4 | |
| 7 | $(Fl_1 - H) + (Fl_2 - \delta)$ | C5→Rb | | | Fp6 | | C5 | read |
| 8 | | C4→Rb | Re→Ra | | | Fd | C4 | read |
| 9 | | Re→C3 | | | Fp3 | | C3 | |
| 10 | $(Fl_1 - H) - (Fl_2 - \delta)$ | C5→Rb | | | Fp6 | | C5 | read |
| 11 | | C4→Rb | Re→Ra | sub | | Fd | C4 | read |
| 12 | | Re→C4 | | | Fp3 | | C4 | |

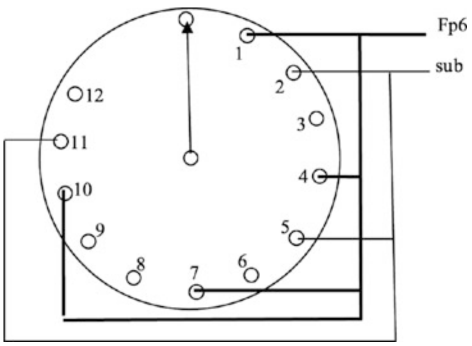


**Fig. 7.9** Example of the wiring of a rotary switch with 12 contacts

The adder is always working and executing addition/subtraction on each cycle. The last four columns describe the control signals needed: the signal "sub" sets subtraction in the adder, the $Cx$ signal selects the memory cell with address $x$, the read signal prepares the memory for a read operation, the default being a write operation. The Fp and Fd signals open or close the data buses so that the necessary register transfers are executed. Fd opens register B so that it can be rewritten. Register A is always rewritten with the last result. Fp6 opens the memory bus for reading a memory cell into register B. Fp3 opens the memory bus for a memory write. From this table, it is easy to wire a rotary switch with 12 steps. Figure 7.9 shows the connections only for the "sub" and "Fp6" control relays. The switch advances one position on each cycle, starting from a neutral initial position. The switch activates all relays connected at the point of contact when it arrives there.

This example clarifies how Konrad Zuse hardwired the S1. Instead of providing primitive instructions (such as addition, multiplication, load, store, etc.) and combining them into programs, he directly transformed all necessary operations into microinstructions, which were then hardwired using steppers. The number of relays was minimized, but the price was the reduced flexibility of the machine. In the 1950s, Konrad Zuse produced a special-purpose machine for the optical industry in which all necessary calculations were again hardwired into the machine.

## 7.5 The S2 and Automatic Process Control

After the S1 had been completed, Konrad Zuse started a more ambitious project—automating data entry. The bottleneck for the whole computation was the manual work needed to measure the wings' shape, fill out the forms, and enter the data into the calculator. Furthermore, the data were entered twice to check the computations. Zuse conceived a mechanism for automatically detecting the position of the measuring instrument in such a way that the result would be immediately available to the computing engine. It was a clever idea: the machine would provide a sequence of pulses to a lever attached to the measuring instrument. Starting from

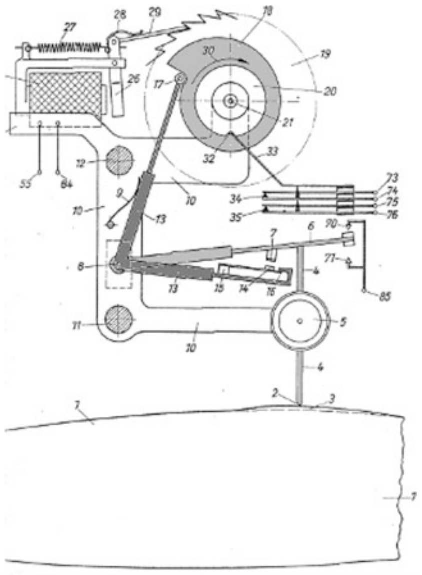


**Fig. 7.10** Diagram by Konrad Zuse of the analog to digital transformation of the wing's measurement (Zuse 1953b)

a known position, the tip of the lever advanced a little with each pulse. When the tip of the lever had moved a distance proportional to the measurement, it would make contact with a metal piece, then a current would flow and stop the sequence of pulses. The analog measurement was then equal to the total number of pulses given to the instrument multiplied by a small constant (the displacement for each pulse). Analog measuring was thus transformed into digital counting. Figure 7.10 shows the sketch sent by Zuse to the patent office. All moving parts have been colored gray. The spiral wheel labeled 30 rotates a little with each pulse and displaces the lever in contact at point 17. The metallic contact 14 moves up until it eventually meets the metallic contact 7, lifted by the measuring tip resting on the wing. Note that lever 6 is passive. The moving lever is the one shaded in darker gray. The pulses for rotating the spiral wheel are delivered by the relay plate 26.

A measuring platform was built, with more measuring points than the S1 had used. Twenty were used for the rear wing, and 24 for the left and right wings, each of them defining four $a$, $b$, $c$, and $d$ measurements as described above, for a total of 176 measuring points. The measurements were transmitted by cables directly to the S2, which could perform the computations for corrections immediately (Fig. 7.11).

The S2 had the same basic architecture as the S1. The main difference (other than the automatic data entry) was the fact that the measuring instruments were read out twice: once when moving toward the wing and once when moving back to the parking position. All computations were performed twice, with the direct read-out and the retracting read-out. The results were compared automatically at the end, and an alarm was given when they were not the same. Therefore, the S2 checked its results through this redundant computation. The S2 was never used as intended. In 1944, it was installed at a Henschel factory in what is now the Czech Republic.

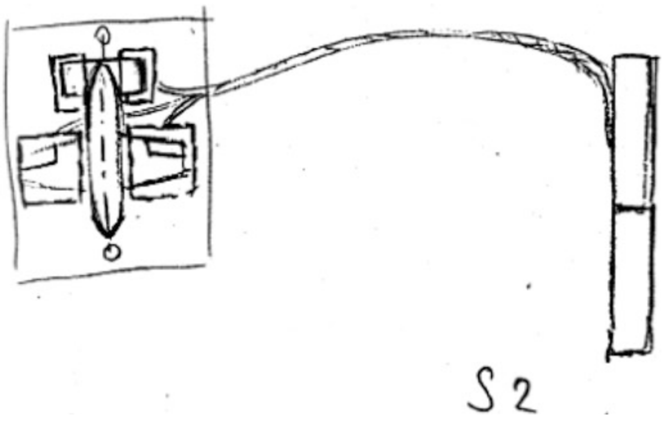


**Fig. 7.11** Sketch by Konrad Zuse of the connections from the measurement instruments to the S2 (Zuse 1942a)

While Zuse was still assembling the S2, brought directly from Berlin, the order to dismantle the complete factory arrived. Zuse feverishly continued assembling the machine although all other equipment around him was being dismantled. The machine worked as it should and was disassembled for transport shortly after, never to be seen again (Zuse 1970). It was probably destroyed during the final months of the war.

## 7.6 Discussion

The S1 and S2 were probably the first digital computing machines used for factory process control. The measurement instrument used in the S2 was also probably the first analog-to-digital converter, although it was never really used in production. Both machines were, from the computational point of view, subsets of the Z3 built in 1941. They were binary, fixed-point, special-purpose, non-programmable machines. Their existence remained unknown to the public at large for many years after the war.

Zuse seems to have been someone fully obsessed with his work, ready to adapt to any social circumstances, as long as he could continue doing his research. The scene he later recalled, of the dismantlement of the S2 in 1944, is especially telling: While the world around him was falling apart, Konrad Zuse was still assembling the S2 so that he could see it functioning at least once. It did not matter that its original task, helping to produce better flying bombs, had become obsolete and absurd. This was a special-purpose machine with no purpose left, but it was his brainchild and he had to see it work.

Some contemporaries of Konrad Zuse fared really well after the war. Prof. Herbert Wagner, his protector and boss at the Henschel Flugzeug-Werke, chief engineer of the German cruise missiles effort, continued his career in the USA, working for the Navy right after 1945, and later as a private military contractor for companies such as Raytheon. In 1951, he started his own company in California. Wernher von Braun, another obsessive engineer and the mind behind the V2 rockets, finished his career working for NASA and directing the flight to the moon. Von Braun had been a member of the SS who knew of the use of slave workers for building the V2 rockets in the Harz caverns in Germany. Curiously, in 1945, both von Braun's and Konrad Zuse's teams were fleeing from the advancing Soviet army and arrived in the very same town in the south of Germany. It has been reported that Zuse avoided contact because he expected von Braun's group to be tried by the Allies (Zuse 2000). How little he understood about the politics of world domination! In 1995, the Technical University of Berlin published a declaration on the occasion of the 50th anniversary of the war's end (Anonymous 1995). The declaration deplores the fact that the university could become a "vehicle of fascist ideology" and that it helped develop "science and technology that harmed humanity." Both Herbert Wagner and Wernher von Braun are mentioned as examples of brilliant yet "ambivalent" scientists and alumni.

Konrad Zuse was a great engineer, but he was compelled to impart a military dimension to his research. He did it willingly, a destiny he shared with several of his fellow scientists toiling in the shadows of Germany's darkest hour.

## References

Anonymous. 1995. Erklärung des Fachbereichs 10, TU Intern, Newsletter.

Rojas R, F. Darius, C. Göktekin, and G. Heyne. 2005. The Reconstruction of Konrad Zuse's Z3. *IEEE Annals of the History of Computing* 27(3): 23–32. https://doi.org/10.1109/MAHC.2005.48.

Zuse, K. 1942a Modell S1/S2. Skizze der Gesamtanlage und Rechenschema. Available online at the Zuse Internet Archive.

Zuse, K. 1942b. Programmgesteuerte Rechenmaschine für Flügelvermessung. Available online at the Zuse Internet Archive.

Zuse, K. 1944a Gerät S2a Rechengerät für Flügelvermessung mit automatischer Ablesbarkeit der Messuhren. 2. Ausführung des Spezialmodells S2. Available online at the Zuse Internet Archive.

Zuse, K. 1953a Patentschrift nr. 872645: Verfahren zur Abtastung von Oberflächen und Einrichtung zur Durchführung des Verfahrens (für Zuse). Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0998/.

Zuse, K. 1953b Verfahren zur Abtastung von Oberflächen und Einrichtung zur Durchführung des Verfahrens. German Patent Office, Patent 872645.

Zuse, K. 1970 Der Computer—Mein Lebenswerk. Landsberg: Verlag Moderne Industrie.

Zuse, H. 2000. Konrad Zuse—Seine Rechenmaschinen. In *Konrad Zuse—Der Vater des Computers*, ed. Alex Jürgen, Flessner Hermann, Mons Wilhelm, Pauli Kurt, and Zuse Horst. Fulda: Verlag Parzeller.

# Chapter 8
# The Architecture of the Z4

*This chapter describes the programming architecture of Konrad Zuse's Z4 computer. The machine's logic was implemented with telephone relays while the memory was a mechanical module. The Z4 was the successor to the Z3 machine completed in 1941—it was designed and built in less than 4 years, until German capitulation. In its first embodiment, the machine featured 12 words of mechanical memory, two CPU registers, one punched tape reader, and one tape puncher. The keyboard accepted decimal input, but the internal numerical representation was fully binary, based on a particular floating-point format. The computer had a relatively large instruction set for arithmetic operations. In 1949, the Swiss Federal Institute of Technology (ETH) in Zurich decided to lease-buy the Z4 from Zuse's fledgling computer company. An additional punched tape unit for reading auxiliary programs or tables of numbers was added, together with the instructions necessary for calling subprograms. The instruction set was extended with a conditional jump.*[1]

## 8.1 Introduction

The Z4 was a computer designed and built by Konrad Zuse between 1942 and 1945 (the original name was *Versuchsmodell 4*, or V4). It represents an important milestone in the history of computing in Germany. This machine was the culmination of the chain of innovations launched by Zuse with the Z1, a mechanical computer completed in 1938 in Berlin. A further embodiment of the same general architecture was the Z3, a machine built completely from telephone relays and demonstrated in 1941 (Petzold 1992; Rojas 1998a; Rojas et al. 2014) (see Chap. 5). However, only the completed Z4 can be understood as the commercial computer Zuse had been struggling to build for so many years. The Z1, Z3, and Z4 shared a few fundamental

[1] Chapter based on Rojas (2021a).

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_8

principles that we recognize today as constitutive of modern computers: all of them had a processor distinct from memory, their design was fully binary with decimal input, computations were performed using floating-point hardware, registers were available in the CPU, and all operations were microprogrammed. However, the programs were held in an external medium, i.e., a punched tape. This control tape could be made to loop just by attaching its two ends. The Z4 was the materialization of Konrad Zuse's evolving concept. He had started working on computing machines long before the war, around 1935–1936, but the bulk of the development effort for the Z4 was done while the war was raging in Europe (Zuse 1970).

Here, I will describe the architecture of the Z4 mainly from the point of view of the programmer. This is what is sometimes called the "functional" or "programming" architecture of a computer. Figure 8.1 shows the Z4 as it used to

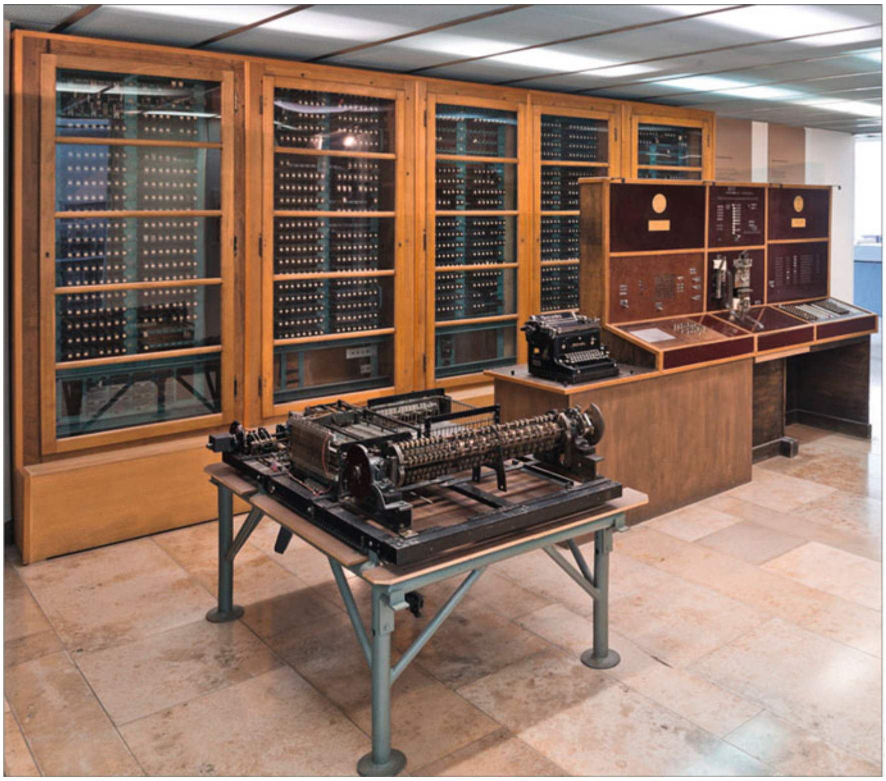


**Fig. 8.1** The Z4 at Deutsches Museum. The mechanical memory and the control console are visible in the foreground. In the background, we can see the racks of telephone relays used for the logic components. The two tape readers are visible in the center of the console. The right side of the console is used for entering instructions and addresses. The left side is used for entering decimal numbers, which are displayed in a lamp array similar to the keyboard of a vintage cash register. Results could be printed with the electric typewriter on the left of the console (Image: Deutsches Museum)

stand in the history of computing hall of Deutsches Museum in Munich. The console is in the front, and the circuits built from relays are in the back.

Konrad Zuse's relatives and a Berlin instrument maker financed the construction of the Z1. The Z3 was built while Zuse was working part-time for the Henschel Flugzeug-Werke making calculations for the wings of airplanes and flying bombs during World War II. Zuse dedicated the bulk of his working time to his own company, which was classified as necessary for the war effort. After the successful demonstration of the Z3, Zuse obtained a loan from an aerodynamics institute (*Deutsche Versuchsanstalt für Luftfahrt*) for the development of a more ambitious machine, that is, the Z4, which was assembled in Zuse's workshop (Petzold 1992). The contract was later taken over by the Aviation Ministry, and the machine was to be delivered to Henschel. A few weeks before the Red Army occupied Berlin, the Z4 was transported to Bavaria, where it remained until 1949.

## 8.2 Block Architecture

It is easier to describe the architecture of the Z4 from the point of view of a programmer by referring to a block diagram containing the essential components (Fig. 8.2, based on Zuse (1952a)).

Programs for the Z4 were encoded in punched tapes using a binary code. The main tape reader, At0, could read one instruction at a time and advance the tape. The control unit transformed each instruction into a sequence of microinstructions for the central processing unit. The processor contained two registers (OR-I and OR-II, also called register $x$ and register $y$, respectively). Data read from memory were loaded to these registers, and then operations requiring two arguments, such as addition or multiplication, were executed with their contents. The result was always rewritten into the first register (OR-I). There was an instruction for storing the contents of OR-I to a specific memory address. The machine had 64 memory words, with addresses 0–63.

The processor of the Z4 computed all arithmetic operations using floating point. The format in memory was similar to alternatives used today. The Z4 used seven bits of each memory word for encoding the exponent (in two's complement representation), 23 bits for encoding the mantissa, and one bit for encoding the sign of the number (Zuse 1952a). An additional bit was used to flag "special values," such as infinity and "indefinite," or "not a number," as we would say today (NaN). Therefore, each word of memory consisted of 32 bits.

The Z4 could be used as a kind of manually triggered calculator: the operator could enter decimal numbers through the decimal keyboard; these were transformed into the floating-point representation of the Z4 and loaded to the CPU registers, first to OR-I, then to OR-II. Then it was possible to start an operation using the "operations keyboard" (an addition, for example). The result was held in OR-I, and the user could continue loading numbers and computing. The result in OR-I could

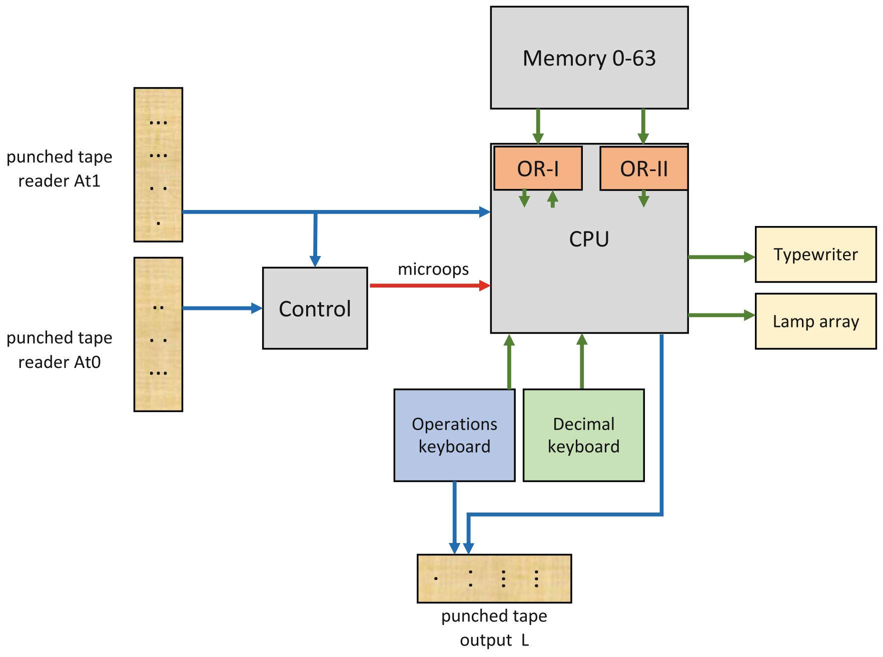


**Fig. 8.2** The block architecture of the Z4 with its main components, as explained in the text. The diagram is based on Zuse (1952a)

be made visible in decimal notation by transferring it to a decimal lamp array (at the push of a button). It could also be printed using an electric typewriter.

The operator could also use the instructions keyboard to punch a program directly to a tape. Electronics in the console translated keypunches into the appropriate binary code for each instruction. This procedure semi-automated the creation of new programs, which for the Z3 still had to be manually encoded by the programmer in binary notation. The CPU could also control the tape puncher directly. It was then possible to store the binary representation of tables of numbers in punched tapes. The table could be reused later, using the secondary tape reader. The Z4 could also compute the binary code for program instructions and punch them on a tape (the program doing this was then a "super program"; today it would be a kind of compiler).

The secondary tape reader, At1, could be used to execute a subprogram. While At0 was reading instructions from the main program, control could be transferred to At1 by the control unit. The operator would have previously loaded a tape from a library of subprograms in At1. When control returned to At0, a new tape could be loaded in At1, in case a new subprogram would be needed afterward. At1 could also be loaded with a table of numbers produced by the tape puncher, and when one of those numbers was needed during execution, the appropriate instruction could load the next number available at At1 to one of the registers.

The Z4 could only use absolute addresses (there was no relative addressing). Therefore, even if it was feasible to simulate a program loop through the simple expedient of joining the ends of a tape, it was not possible to make an index variable point sequentially to a range of addresses, to read them one by one. But in that case, the tape reader At1 could be used as a kind of substitute, since the tape advanced one position each time a number was read from it.

If I had to summarize the Z4 (of 1950) for an audience of modern computer programmers in just a few words, I would say that the Z4 was a programmable machine featuring 64 words of memory. It had a floating-point CPU with two registers and used two punched tape readers to read programming code, one of which could also be used as an external numerical memory or for library tapes. The bulk of the instruction set was dedicated to arithmetic operations, but subroutines could be called (one level deep) and (since 1950) there was a conditional jump instruction. The output of the machine could be visualized in the console using lamps or could be printed.

## 8.3 Architectural Details

There are a few idiosyncrasies of the Z4 that it is convenient to explain at this point. The first surprise is that, while the control part of the Z4 was designed using telephone relays, the memory was a mechanical apparatus (Zuse 1944c). Zuse was a master of mechanical design, and his first computer was built using entirely mechanical components. For the Z4, Zuse returned to a mechanical memory because telephone relays were expensive and bulky at that time (Zuse 1946a). He reckoned that he could build the mechanical components necessary for storing memory bits more economically and needing much less volume using his so-called "mechanical relays." His original intention was to build a mechanical memory of up to one thousand words in a small volume (Fig. 8.3). He wrote in 1943: "A storage unit with 1000 storage cells occupies a surface of 1 to 2 square meters. For the same purpose, we would need 45,000 electromagnetic relays (...) For the processor, we could have a similar reduction of the space needed. If we built mechanical devices applying the same principles as for pocket watches, we could make them fit in the format of a typewriter. They could be used in cars, ships and even airplanes" (Zuse 1943a). Both the prescience of these sentences and their naiveté regarding the future of the technology needed for computers are surprising.

The first iteration of the Z4 contained only one bank of memory for 64 words (the prototype had only 12). But it required much less space than the equivalent memory built with relays (the memory of the Z3 used relays). We don't know how much calculating speed was lost by reading from a mechanical instead of a relay memory. However, such loss was lessened by making the memory work in parallel with the processor. It must be said, though, that the mechanical memory proved to be reliable enough during the many years the Swiss Federal Institute of Technology (ETH) operated the machine in Zurich, although it had been the main

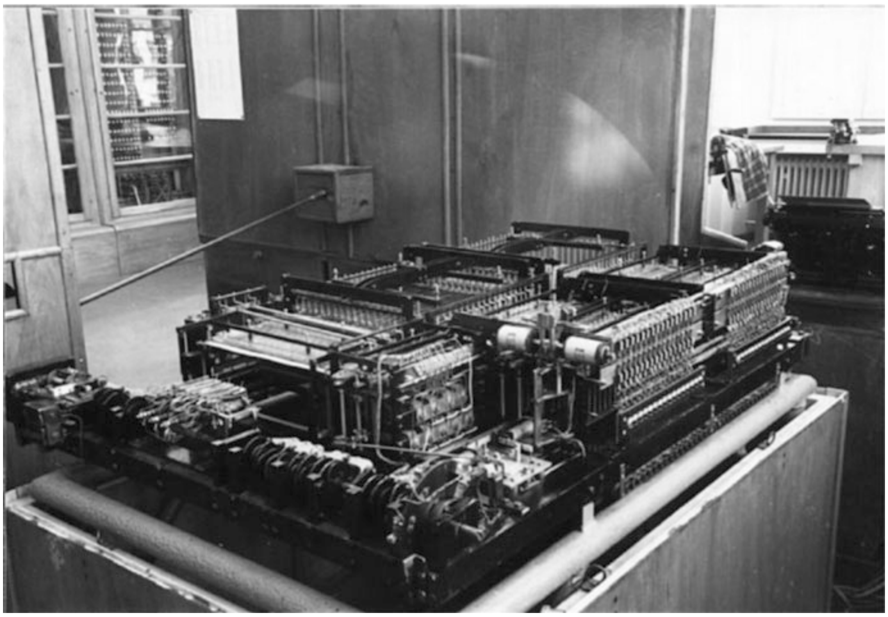


**Fig. 8.3** Picture of the mechanical memory of the Z4 at the ETH in Zurich (Image: ETH Library)

concern when the machine was leased (Bauer 2008). However, assiduous users of the machine complained about occasional problems when parts of the mechanical memory jammed (Bruderer 2012).

Execution times for the important instructions were as follows:

- Addition/Subtraction: 0.5 seconds
- Multiplication: 3 seconds
- Division and square root: 6 seconds
- Memory access: 0.5 seconds (could be overlapped with the execution time of arithmetic operations)

In typical programs, the Z4 could compute a mix of around 1000 arithmetic operations per hour (Bauer 2008). Since memory access overlapped with processing, the slow mechanical memory was not a handicap for the machine.

As explained before, the floating-point format used by Zuse reserved 7 bits for the exponent, in two's complement representation, so that the range of possible binary exponents ran from $-64$ to $+63$. The mantissa was stored using normalized floating point, where the leading bit before the binary point is always 1. In total, 23 bits were needed to store the mantissa in memory, but the representation was expanded to 24 bits in the processor (supplying the leading one). One bit was used for the sign of the number, and another to signal the presence of a special value so that a total of 32 bits were necessary to store a number in memory. There was a special coding for zero (which cannot be represented as a normalized floating-point number), and also for infinity and not a number (NaN), which was called an “indefinite value” and was represented symbolically by Zuse as “?”. Dividing zero by zero, for example, could produce an indefinite “?”. The special coding for zero, infinity, and NaN is not documented in the programming manual. We only know that additional bits were used to differentiate between normal numbers and special values (Zuse 1952a).

To enter a decimal number through the decimal keyboard, required selecting the specific decimal digits of the number, followed by the exponent. The result was visualized with an array of nine columns of lamps. Every column had a lamp for the digits 0–9. The specific decimal digit at each decimal position, for every one of the nine columns, was lit, and the operator could write down the result produced by the machine. That lamp array can be seen on the upper left of the console in Fig. 8.1. There were nine full columns for decimal digits. The position of the decimal point in the mantissa was indicated with an additional row of lamps under every column (only one “decimal point” lamp would be switched on). The exponent of the result was shown with additional lamps. There were also lamps for the sign, and to indicate underflow, overflow (infinity), or not a number, indicated by a lamp with a question mark.

## 8.4 The Arithmetic Instruction Set

The instructions for the Z4 were implemented using microoperations, as in the Z3. For every operation, there was a control sequencer, which was just a circular mechanical stepper advancing from one position to the next, like a clock. At every microstep, different circuits of the processor were energized, and this produced the information flow in the Z4. Such a microcoded architecture was first developed for the Z1, while rotary microsteppers were used for the Z3 (Rojas 1997) and Z4. Microcoding made Zuse’s machines very flexible. New operations could be created by simply wiring a new stepper.

Every program (punched tape) for the Z4 started with the instruction “St.” The end of the code was signaled with the instruction “Fin.” The Z4 advanced a new tape until the first “St” appeared.

The Z4 performed arithmetic operations with one or two arguments. In what follows, the first register (OR-I) is abbreviated as “$x$” and the second as “$y$.” The operations with a single argument included nine multiplications by constants (the divisions were transformed into multiplications). The binary representation of the constants was hardwired in the machine:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $-x$ | $2x$ | $10x$ | $\pi x$ | $x/2$ | $x/3$ | $x/5$ | $x/7$ | $x/\pi$ |

Additional one-argument operations were:

| | | | | |
|---|---|---|---|---|
| $x^2$ | $\sqrt{x}$ | $\|x\|$ | $\mathrm{sgn}(x)$ | $\max(0, x)$ |

Two-argument operations were:

| | | | | | |
|---|---|---|---|---|---|
| $x - y$ | $y - x$ | $x \times y$ | $x/y$ | $\max(x, y)$ | $\min(x, y)$ |

The instructions for reading and writing to memory were "A n" and "S n," where n represents the memory address. The first A-instruction would load register OR-I, while the next A-instruction would load register OR-II. It was also possible to read a number from the tape reader using the command $\uparrow m$, where $m$ is a number encoded in the punched tape immediately after the arrow-up command.

When the Z4 was running, it could request decimal input from the operator. The instruction to request manual input was $\uparrow$. The result contained in register $x$ could be shown activating the console decimal lamps using the instruction $\downarrow$, or printed using the instruction D, immediately after $\downarrow$.

There were a few additional special commands used for formatting the output or for making possible certain combinations of otherwise prohibited instruction sequences. For example, because the memory was mechanical, the processor would be too fast. It was not possible to store a result to an address and read the address immediately. The programmer had to be careful and wait a certain number of instructions for the memory to be addressable again. This confirms that memory access was partly asynchronous, relative to the processor, in order not to slow the latter down. Prohibited sequences of operations, in terms of timing, made the machine stop during execution (Bauer 2008). An expert programmer reviewed always the users' code to make sure that forbidden combinations of instructions were not present.

And that's it. This was most of the instruction set used by the Z4 until 1945 (Zuse ca 1945). It was effectively a superset of the instruction set of the Z3 and the main missing ingredient is, of course, conditional branching and being able to call subroutines. In fact, the tape reader At1 in Fig. 8.1 was included after 1945, and the instruction set was extended to deal with conditionals.

Figure 8.4 shows a diagram produced by Zuse's company before the Z4 was leased to the ETH in Zurich. The mechanical memory has the label 10. There is one tape reader (8) and a tape puncher (7). The processor logic was housed in the relay casings 1–4. There is no electrical typewriter. A planar array of lamps (5b), organized in rows and columns, could be used to light up a specific lamp under a sheet of paper, signaling to the operator the name of the variable that had to be entered if the machine stopped for manual input. The programmer had to take care to switch on the correct lamp, which was reached by going down in the rows and to the right, using special instructions embedded in the code stream. This was the state

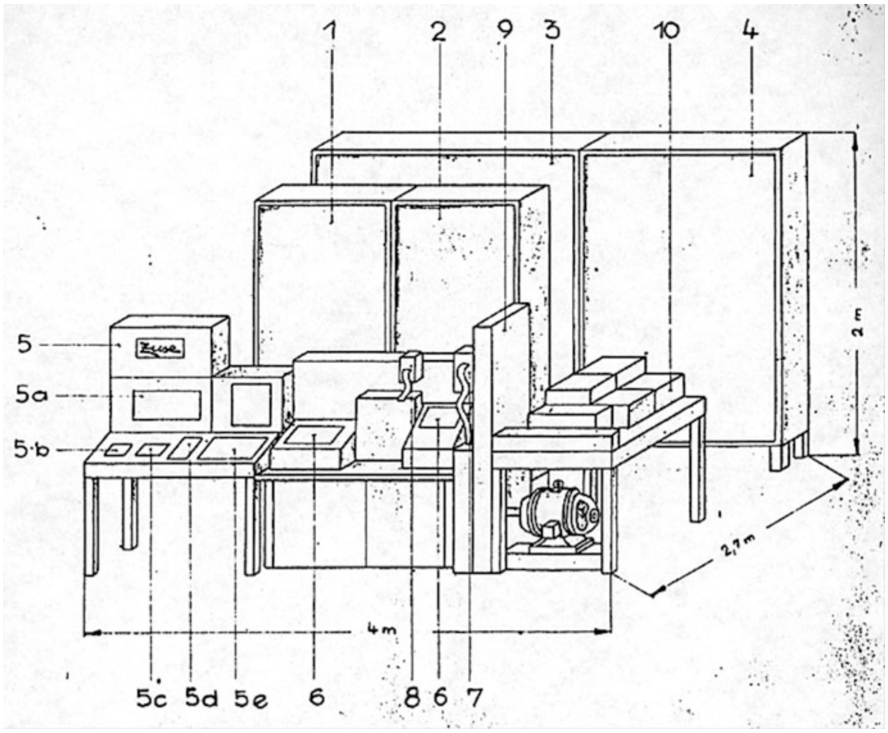


**Fig. 8.4** The Z4 after 1945 and until 1949 (Zuse 1946c)

of development of the Z4 until 1945 and before the ETH became interested in the machine.

## 8.5 Conditionals and Control Transfer

Konrad Zuse explains in his memoirs that the Z4 was transported out of Berlin a few days before the city fell. The machine spent several years in a barn in Bavaria. The Swiss mathematician Eduard Stiefel heard about the computer, visited Zuse in 1949, and was able to see the Z4 working properly. His university, the ETH, decided to lease the machine, with an option to buy it at the end of 5 years, but important modifications had to be made beforehand, the main one being the inclusion of conditional branching. Zuse complied, and in 1950, the Z4 was delivered to the ETH (Speiser 2000; Bruderer 2012). It was the first commercial computer rented or sold by any company in continental Europe. During the first months after delivery, the relays had to be adjusted since the number of switching operations was much higher than in telephone networks (Petzold 2004).

The conditional jump promised to Stiefel was implemented by Zuse using the new instruction "Sp" (for "Sprung", in German). If the contents of register $x$ is $+1$, the punched tape is rolled down until a new start instruction is found in the tape (that is, "St"). Execution continues normally from that point on.

Before a conditional instruction could be executed, a logic result had to be computed in register $x$. For this purpose, there were five arithmetic test operations, designed to check if a condition is fulfilled. The test "$x\ =?$", for example, verifies whether the result of the previous operation was a NaN or not. A successful test fills the register $x$ with $+1$, otherwise with $-1$. The test operations were:

| | | | | |
|---|---|---|---|---|
| $x = 0$ | $x \geq 0$ | $x = \infty$ | $x =?$ | $\lvert x\rvert \geq 1$ |

The instruction "Up" was used for transferring control to subroutines. Execution continued at the current position of the punched tape in At1, and control was returned to At0 when the instruction "Fin" was reached in the subprogram (Rojas 2014a). There was a conditional variation of Up and Fin that is less relevant to the discussion here (see the next chapter for a full discussion of subroutine transfer and conditional jumps in the Z4). Figure 8.5 shows the lamps in the console for the complete instruction set.

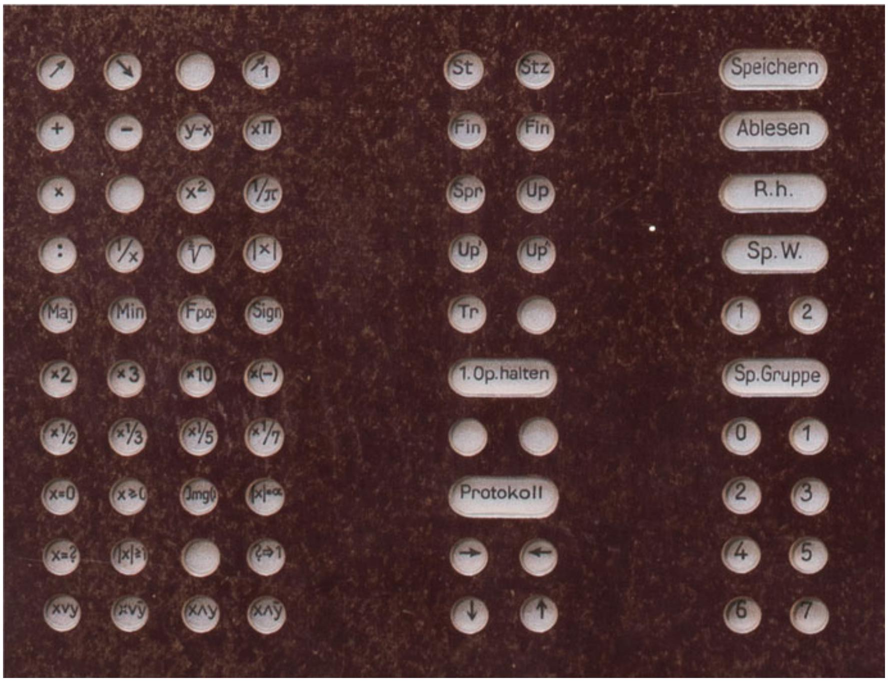


**Fig. 8.5** The complete instruction set in the console lamps of the Z4. The last row of logic operations to the left is not covered in the programming handbook (Image: Deutsches Museum)

After the new conditional and control transfer instructions were introduced, it was possible to call subroutines. The additional tape reader also allowed the programmer to use the secondary tape for reading numbers punched by another program.

However, there is a problem with this implementation of the conditional jump. The program can only jump down in the code. But programming loops require reusing previous code so that the jump would have to be taken upward in the instruction sequence. The solution in the Z4 was to glue the punched tape, making the code execute in a cycle. The programming manual of the Z4 explains how to glue the tape and also that a minimum length of 50 cm is required so that the tape does not jam. If the loop had only a few instructions, several copies of its code had to be punched, one after the other, until the minimum tape length was obtained. This is called "loop unrolling," and it was used in the Z4 so that the punched tape could meet the minimum specified length.

Zuse's way of implementing the conditional jump is unsatisfactory, because either the programmer resigns to have a single loop in the code or special tricks have to be applied (Rojas 1998b). One trick would be to have the body of the loop as external code in At1, which can be called successively a fixed number of times in the main program. The other would be to assign a sequential number to all loops in a program and arrange them into a single loop of tape. Then, during execution, we would only enter the loop guarded by a conditional comparison with the value of a variable placed before the loop body. This is cumbersome, and it would be interesting to find actual code of the Z4 to learn how users solved this problem. Note that Charles Babbage, who used punched cards strung together, designed the conditional jump to go up into the stream of cards so that loops could be easily implemented (Bromley 2000).

Since the Z4 did not have indirect addressing capabilities, loops that need to address memory sequentially are difficult to implement. Adding 20 numbers in memory, for example, would require specifying the 20 addressing instructions with the consecutive absolute addresses. The alternative would be to use the tape reader At1 so that the numbers can be read one by one from that reader without having to specify absolute addresses. The sequencing of the data is then done automatically when the reader advances. Charles Babbage had the same difficulty with the Analytical Engine, but he envisioned making the stream of "number cards" bidirectionally steerable so that interesting combinations of data input could be achieved when running loops (Rojas 2021b).

## 8.6 Conclusions

From a contemporary point of view, the architecture of the Z4 can be readily explained using modern terminology. And that is the biggest surprise when thinking about this machine and comparing it with the computers designed up to 1945.

The first comparison that comes to mind is with Babbage's Analytical Engine. Zuse's Z1 was actually something like Babbage's dream materialized, in the sense that all important arithmetic operations were implemented using only mechanical means. The Analytical Engine went further, though, since it included conditional instructions, so important for universal computation. Curiously, Zuse did not include the conditional jump as a programming instruction, neither in the Z1, the Z3, nor in the Z4 (until 1945). Zuse referred to the programs that could be written in this way as "rigid" because he was fully aware that conditional instructions could make programming "flexible." Curiously, the microcode in all his machines was based on conditional execution.

The Z4 was a floating-point machine. Neither the Harvard Mark I nor the ENIAC used floating point. Both used a fixed-point representation. Neither the Mark I nor the ENIAC were fully binary. The internal representation of numbers was decimal, using gears in the case of the Mark I, and arrays of vacuum tubes in the case of the ENIAC. The Z1 was already completely binary when it was finished in 1938.

The separation of memory and processor is also complete and pervasive in Zuse's machines. In the Mark I and the ENIAC, memory and processor are still intertwined, since memory words are used as accumulators for arithmetic operations. Even the Analytical Engine is superior in that respect, since the storage was completely separated from the mill, and they even ran independently, each one using its own set of punched cards. Curiously, the ENIAC did not execute code. The code was embedded in the way the machine was hardwired, and the connections had to be rearranged for each new problem.

As we can see, all of these machines brought something new in terms of the computing architectures that would become possible in later years. At some point, all of them have been called the "first computer." However, I think that a comparison of their architectures confirms that we can only talk about the "first computers," in the plural, since the dawn of the third industrial revolution was an endeavor that went beyond national boundaries. The start of the computer age was a collective enterprise whose first creative spark flashed during the heyday of the first industrial revolution with the inception of the Analytical Engine.

## References

Bauer, F.L. 2008. Über "Episoden aus den Anfängen der Informatik an der ETH" von A. Speiser. *Informatik-Spektrum* 31(6): 600–612. https://doi.org/10.1007/s00287-008-0291-8.

Bromley, A.G. 2000. Babbage's analytical engine plans 28 and 28a – The programmer's interface. *Annals of the History of Computing* 22(4): 5–19. https://doi.org/10.1109/85.887986.

Bruderer, H. 2012. *Konrad Zuse und die Schweiz: Wer hat den Computer erfunden?* Munich: Oldenbourg Wissenschaftsverlag. https://doi.org/10.1524/9783486716658.

Petzold, H. 1992. *Moderne Rechenkünstler: die Industrialisierung der Rechentechnik in Deutschland*. Munich: C.H. Beck.

Petzold, H. 2004. Hardwaretechnologische Alternativen bei Konrad Zuse. In *Geschichten der Informatik: Visionen, Paradigmen, Leitmotive*, ed. HD Hellige. Berlin, Heidelberg: Springer. https://doi.org/0.1007/978-3-642-18631-8-5.

Rojas, R. 1997. Konrad Zuse's legacy: The architecture of the Z1 and Z3. *IEEE Annals of the History of Computing* 19(2): 5–16. https://doi.org/10.1109/85.586067.

Rojas, R. 1998b. How to make Konrad Zuse's Z3 a universal computer. *IEEE Annals of the History of Computing* 20(3): 51–54. https://doi.org/10.1109/85.707574.

Rojas, R. 2014a. Konrad Zuse und der bedingte Sprung. *Informatik-Spektrum* 37: 50—53. https://doi.org/10.1007/s00287-013-0717-9.

Rojas, R. 2021a. The architecture of Konrad Zuse's Z4 computer. In *IEEE 7th IEEE History of Electrotechnology Conference*, Moscow, 43–47.

Rojas, R. 2021b. The computer programs of Charles Babbage. *IEEE Annals of the History of Computing* 43(1): 6–18. https://doi.org/10.1109/MAHC.2020.3045717.

Rojas, R., Röder, J., and H. Nguyen. 2014. Die Prozessorarchitektur der Rechenmaschine Z1. *Informatik-Spektrum* 37(4): 341–347. https://doi.org/10.1007/s00287-013-0749-1.

Rojas, R. 1998a. *Die Rechenmaschinen von Konrad Zuse*. Berlin: Springer-Verlag. https://doi.org/10.1007/978-3-642-71944-8.

Speiser, A. 2000. Konrad Zuse's Z4: Architecture, programming, and modifications at the ETH Zurich. In *The first computers - history and architectures*, eds. R. Rojas, U. Hashagen Cambridge, MA: MIT Press.

Zuse, K. 1943a. Rechenplangesteuerte Rechengeräte für technische und wissenschaftliche Rechnungen. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0217/

Zuse, K. 1944c. Notizen zu den Schaltungen. Available online at the Zuse Internet Archive.

Zuse, K. 1946a. Kurze Beschreibung des mechanischen Speicherwerks der Firma Zuse. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0249/

Zuse, K. 1946c. Zuse Calculators. Available online at the Zuse Internet Archive.

Zuse, K. 1952a. *Bedienungsanweisung für Zuse Z4*. Zurich: ETH Zurich. https://doi.org/10.7891/e-manuscripta-98601.

Zuse, K. 1970. *Der computer – Mein Lebenswerk*. Landsberg: Verlag Moderne Industrie.

Zuse, K. ca 1945. Rechenpläne für das Rechengerät V4. Available online at the Zuse Internet Archive.

# Chapter 9
# The Conditional Jump: Making the Z4 Universal

*This chapter describes the conditional instructions that Konrad Zuse retrofitted to the instruction set of the Z4 around 1949, including the conditional jump. The instruction set upgrade for the Z4 was a request from the mathematicians at the ETH Zurich, who needed the increased functionality for iterative numerical computations. If the truth value stored in a register was "true," the conditional jump would bypass all instructions further down in the tape containing the program until a "start" marker in the code was reached. If the tested truth value was false, the jump instruction was simply ignored. Some simple programming examples help to understand the operational semantics of the conditional instructions of the Z4.*[1]

## 9.1 Coding for the Z4

It is well known that none of the first calculating machines built by Konrad Zuse featured the conditional jump in their instruction sets. The Z1 (1936–1938), the Z3 (1940–1941), and the Z4 (1942–1945) were originally designed to execute fixed sequences of arithmetic operations (Rojas, 1998a), i.e., the machines could add, subtract, multiply, divide, and process some other arithmetic instructions, one after another. Working in combination with a memory unit, the processor could thus evaluate complex arithmetic expressions. Commands for input and output were also available—only the conditional jump was missing, so that both the Z3 and Z4 could be classified as full-fledged computers (Rojas, 1997).

The programs for these Zuse machines were punched on a tape using binary coding. The instructions were read and executed one at a time. Of course, the program could not be changed while the machine was running—the most that could be done was to tie the punched tape into a loop that could be run repeatedly.

[1] Chapter based on Rojas (2014a).

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_9

I have shown elsewhere that a universal computer can be implemented without a conditional jump by simulating the IF command with such an arithmetic loop (Rojas, 1998b) (see Chap. 6). The whole approach is theoretically conceivable, but impractical, due to the enormous growth of the program code (and thus the length of the punched tape).

The Z4 had to be enhanced with the conditional jump as a precondition for its leasing by ETH Zurich (Zuse, 1970; Bruderer, 2012). The instruction set was adapted to the needs of numerical analysis at the request of the ETH mathematicians: thus, the jump and other conditional instructions were implemented. It was then possible to change the order of calculations adaptively, as is common in any programming language today.

This chapter describes the implementation of the conditional jump for the Z4, the first commercial computer in mainland Europe. The chapter is based on documentation of an exhibition at the ETH from 1981 (Anonymous, 1981). We know for sure that conditional commands were not introduced in the Z4 until after 1945 (Zuse, 1970).

For the following description, it is sufficient to know that the processor of the Z4 has two floating-point registers (called OR-I and OR-II) and 64 memory cells. All arithmetic operations use these two registers and return the result to OR-I.

## 9.2 The Punched Tape of the Z4

Two paper tape readers were included in the Z4 delivered to ETH (Zuse had thought of even more readers, but did not implement them for this project). In the first reader (called At0, for "Abtaster Null"), the main program could run until an Up command handed over control to the second punched tape reader (At1), as we will discuss below. It was then possible to call subprograms from a main program.

The program code started with the pseudo command St (a pseudo command is just a symbol in the sequence of punched cards). A program ended with the Fin command. Upon reaching Fin in the main program, a red light was turned on, to alert the operator about the end of the whole computation.

The Up command passed control from the main program being read by the At0 unit to the subprogram at At1. When control was handed over to At1, its punched tape was advanced to the nearest St. The new instruction sequence for the processor now came from At1. Upon reaching Fin, control was given back to At0. All subprogram parameters and results could be transferred via the contents of memory cells (there was only one global memory with addresses 0 to 63). However, address 63 was special; it was used to hand back an error code. Since the Z4 worked with floating-point numbers, some arithmetic exceptions (infinite result or not a number) could be stored in special bits of the exponent, as in today's IEEE floating-point format. After a subprogram gave control back, address 63 could be tested for exceptions. For example, the Z4 command "$x$ =?" checked whether the content of address 63 (previously loaded in OR-I) was a valid floating-point number.

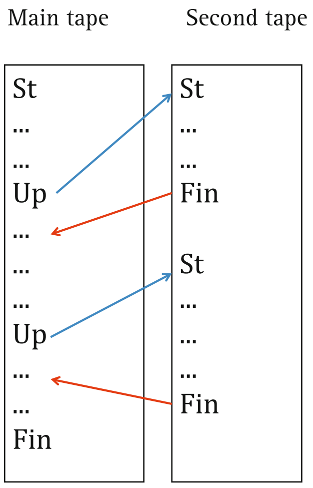


**Fig. 9.1** Subroutine invocations. Scanner At0 (left) passes control twice to scanner At1 (right). The subroutines are called in the order in which they are present in the punched tape reader

Therefore, arithmetic exceptions in subprograms could be reported back to the main program. Figure 9.1 shows a diagram of the execution flow for two subprogram calls. The order in which they were executed resulted from the current position of the punched tape in At1.

## 9.3 Conditional Commands

Zuse encoded truth values using two integers, namely $-1$ for false and $+1$ for true. The logical operations of the Z4 stored the truth value of a result in OR-I. For example, the operation "$x = 0$" stores $+1$ in OR-I if OR-I is equal to zero, otherwise a $-1$ is stored. Other logical operations were, for example, the comparison of OR-I with 1, or testing if OR-I was greater than or equal to zero. Conditional commands derived their actions from the resulting truth values.

The Up' command (with an apostrophe) was equivalent to Up but executed conditionally. If the truth value in OR-I was false, Up' was skipped; if OR-I was true, control was passed to the subprogram in At1.

The Fin' command in the main program was equivalent to Fin, but was skipped if the current logical value in OR-I was false. However, if OR-I was true, the machine stopped completely at Fin'. In this sense, an executed Fin' in a subprogram was like a Fin in the main program. This asymmetry of the Fin commands was not satisfactory in the Z4, but Zuse must have had reasons for this approach. Obviously, an Up' command should not be used in subprograms in At1, since there was no third punched tape reader in the Z4 (Fig. 9.2).

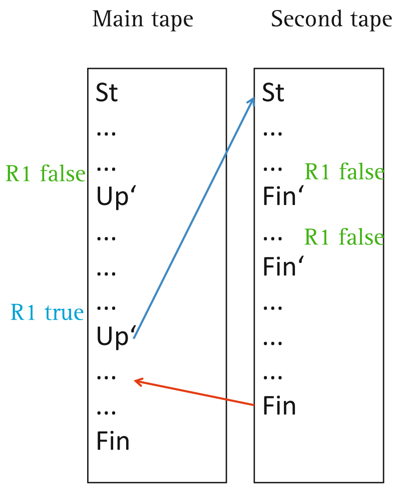


**Fig. 9.2** Subprogram call. Reader At0 gives control to reader At1. The first Up' command is not executed (OR-I contained −1), only the second Up' is. The first two Fin' pseudocommands are not executed (since OR-I contained −1 at that moment). Only the Fin command at the end transfers control back to the main program

## 9.4 The Conditional Jump

And finally: the conditional jump. It was an odyssey of almost 14 years (1936–1950) until the first conditional jump in a Zuse machine saw the light of day (Fig. 9.3). The Z4 command Spr in a program or subprogram caused all subsequent commands to be skipped until the next start mark (St) (Ambros Speiser called this command Spr', a slight difference with the Z4 programming handbook (Bauer, 2008)). This means that the command pair Spr-St acts like a bracket: everything in between is only

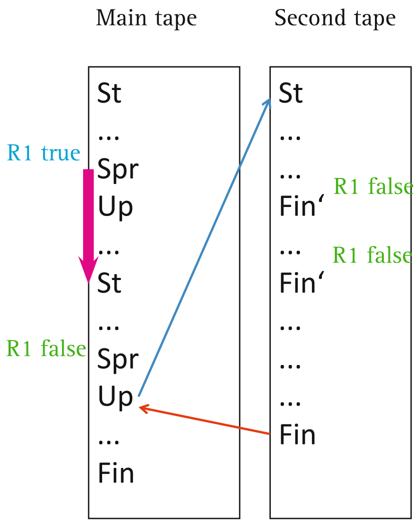


**Fig. 9.3** The first Up command is skipped; the second is not. In the subprogram, the two Fin' instructions are not executed. The appropriate logical values in OR-I for this are: true, false in the main program, and false, false in the subprogram

executed if the register OR-I contains a $-1$. Otherwise all the bracketed commands are skipped.

Other subtleties of the instruction set are not relevant for us (for example, there were forbidden sequences of instructions). We can only underline how simple Zuse's solution for the introduction of conditional commands was. This conditional jump does not refer to addresses (it is not like a GOTO n), i.e., the program cannot jump "backward" in the punched tape, only forward to the next St pseudoinstruction. The commands Up' and Fin' are similarly easy to implement: you can ignore them depending on the truth value in OR-I. Anything can be programmed with such additional commands. We noted above that the subprograms are called in the order of their presence in the punched tape read by At1. However, if we want to assign identifiers to the subprograms, so that, for example, we can call any subprogram between 1 and 6, this is easily done. To that end, we make a loop with the subprogram tape and store the subprogram number in address 0. When the subprogram code starts, we test at the beginning of the successive subprograms if the number in address 0 corresponds to their assigned number. If it does, that subprogram is executed until Fin. If not, that subprogram is skipped up to the next St mark, which corresponds to the next subprogram.

## 9.5 The Competitors

Today, almost eight decades after 1945, it seems odd that Konrad Zuse implemented so late the conditional jump in his machines. However, one should not ignore the state of computing at the time and the work of the competition. Even the American ENIAC, often praised as the world's first electronic calculator, did not initially have a conditional jump (Goldstine and Goldstine, 1996). This was only implemented later, using a trick: data lines that transported pulses were repurposed as control lines (for starting a computation in an accumulator), and so a sign bit could start a chain of arithmetic operations. This had not been foreseen at the beginning of the design. Also, John Atanasoff's machine was built only for a fixed sequence of operations (the Gauss algorithm for solving linear equations). In the Turing machine of 1936, control jumps are implicit in all state transitions. However, if a Universal Turing Machine simulates a specific Turing machine, encoded in its tape, conditional operations have to laboriously skip program and data cells in the tape, one by one.

It is interesting to add that Charles Babbage's Analytical Engine handled the conditional jump very much as Zuse did, testing a memory address and skipping instructions stored in punched cards, until a mark was reached. However, Babbage's conditional jump was provided for jumping up in the sequence of instructions, that is, for implementing iterative loops. In the case of the Z4, iterative loops had to be implemented as successive subroutine calls (using loop-unrolling) or by joining both ends of the punched tape.

It is sometimes easy to criticize the short-sightedness of inventors of the past. But you have to remember that those researchers were busy creating something completely new. The computer was not born like Athena from the head of Zeus, complete and armored. The development of the computer was not a single flash of inspiration—it was an extended revolutionary epoch that continues to this day.

## References

Anonymous. 1981. Documentation: Konrad Zuse and the Early Days of Scientific Computing at the ETH. ETH Zurich, 17.6.-15.7.1981, the documentation contains the list of commands for the Z4.

Bauer, F.L. 2008. Über "Episoden aus den Anfängen der Informatik an der ETH" von A. Speiser. *Informatik-Spektrum* 31(6): 600–612. https://doi.org/10.1007/s00287-008-0291-8.

Bruderer, H. 2012. *Konrad Zuse und die Schweiz: Wer hat den Computer erfunden?* Munich: Oldenbourg Wissenschaftsverlag. https://doi.org/10.1524/9783486716658.

Goldstine, H.H., and A. Goldstine. 1996. The electronic numerical integrator and computer (ENIAC). *Annals of the History of Computing* 18(1): 10–16. https://doi.org/10.1109/85.476557.

Rojas, R. 1997. Konrad Zuse's legacy: The architecture of the Z1 and Z3. *IEEE Annals of the History of Computing* 19(2): 5–16. https://doi.org/10.1109/85.586067.

Rojas, R. 1998a. *Die Rechenmaschinen von Konrad Zuse*. Berlin: Springer-Verlag. https://doi.org/10.1007/978-3-642-71944-8.

Rojas, R. 1998b. How to make Konrad Zuse's Z3 a universal computer. *IEEE Annals of the History of Computing* 20(3): 51–54. https://doi.org/10.1109/85.707574.

Rojas, R. 2014a. Konrad Zuse und der bedingte Sprung. *Informatik-Spektrum* 37: 50–53. https://doi.org/10.1007/s00287-013-0717-9.

Zuse, K. 1970. *Der computer – Mein Lebenswerk*. Landsberg: Verlag Moderne Industrie.

# Chapter 10
# Plankalkül

*This chapter describes the first implementation of Plankalkül, the programming symbolism invented by Konrad Zuse in 1945. Plankalkül is both a high-level imperative programming language and a logic specification notation. In Plankalkül, programs can define functions that can be called non-recursively in other programs. There are no preliminary variable declarations: the type of a variable is specified when it is used. The main imperative constructs are: variable assignment, arithmetic and logic operations, guarded commands, and While loops. Plankalkül is also declarative: some special list, set-theoretic, and logic functions are part of the language definition. Plankalkül uses a two-dimensional layout that defies traditional parsers. This and some inconsistencies in the original definition were the main obstacles to its implementation.*

*At the beginning of this century, our team fixed some inconsistencies in Plankalkül and identified a powerful subset of the language for which we wrote a syntax-driven editor, a parser, and a runtime system. The code was written in Java and could run on any computer connected to the Internet (until Java applets became obsolete). The editor can generate only syntactically valid programs following Zuse's original two-dimensional layout, even when the programmer is first learning the language. More than 55 years after its conception, the first Plankalkül programs ran in February 2000.*[1]

## 10.1 Introduction

Konrad Zuse completed the design of the high-level programming language Plankalkül (Calculus of Programs) in 1945, after leaving Berlin at the end of World War II. The original manuscript was published in revised form in Zuse

[1] This chapter is based on Rojas et al. (2000).

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_10

(1972). Plankalkül was his special project for several years, as it was intended to be part of a doctoral thesis. Anyone who has had the opportunity to study the original definition of Plankalkül is struck by its modern flavor and powerful constructs—as if it had been created much later than 1945. Most surprising, however, is the fact that at the time that Konrad Zuse finished his Plankalkül document, the only two working computers in the world were the ENIAC and the Harvard Mark I (and some other special-purpose machines, such as the computers built at Bell Labs and the British cryptographic machines). None of them used a compiler or a formula translator—the ENIAC had to be rewired for every different problem.

Between 1936 and 1945, Zuse built three programmable computers that embodied the same general computating principles. The Z1 (1938), the Z3 (1941), and the Z4 (1945) were all binary "algebraic" floating-point machines, with a memory separate from the processor, and a program stored in punched tape. They were programmed in machine language, as were the first American or British computers. However, by his own account, Zuse very soon realized that his "combinatorics of conditionals" (as he called it) was identical to propositional calculus, and later on he conceived a simpler but powerful machine, the "logic machine," which would be able to solve both numerical and symbolic processing problems.

Although Zuse applied for a patent for the logic machine, he never really finished designing it. The logic computer was minimalistic and similar to a Turing machine: it consisted of a memory with one-bit words and a processor capable of executing only the logic operations AND, OR, and NOT with one-bit operands. The machine would represent the lowest level in a computational hierarchy with Plankalkül at the top.

In 1942/43, Zuse began writing a PhD dissertation. The draft describes the predicate logic "to make it accessible for engineers" and goes into great detail about its implementation with mechanical and electrical relays. Konrad Zuse's planned thesis, never submitted, is in fact one of the first treatises on the systematic construction of computer circuits (Shannon did something similar in 1936). He describes how to map logic formulas to relay circuits and vice versa. He considers the problem of minimizing circuits and how to overlap them in order to use fewer components. He explains clocked circuits and everything else that is needed to build a computer.

The continuation of this unfinished work is the Plankalkül document, written between 1943 and 1945. Prevented from working on the Z4, which he moved from Berlin to Hinterstein, a small town in the Bavarian Alps, Zuse sat down to summarize how he thought logic machines of the future should be programmed. His original intention was that Plankalkül would be the basis for a complete "calculus of programs," that is, a method for deriving programs from other programs. The initial notation for the Plankalkül was based on the predicate calculus and from there Zuse derived the imperative constructs needed to execute the "explicit form" of the computation. In modern terminology, the Plankalkül has the following main features:

- declarative predicate logic and set-theoretic instructions are part of the language
- it is a high-level imperative programming language (an algorithmic language)
- it is also a specification language for computations using the predicate calculus
- programs are reusable functions
- functions are not recursive
- only call by value is used in function invocation
- variables are local to functions (programs)
- it is a typed language
- the fundamental data types are multidimensional arrays and tuples of arrays
- the type of the variables does not need to be declared in a special header
- conditionals are processed using guarded commands
- there is a WHILE construct for open iteration
- there is no GOTO construct

The main non-modern feature of Plankalkül is its mixed one-dimensional and two-dimensional layout, which has puzzled many readers of the original document. Variables are written using four lines instead of using brackets to enclose indices. It may well be that this peculiar layout was one of the main deterrents to the development of a compiler or interpreter for the language in the first years after its definition.

## 10.2 Origins of the Plankalkül

During the time that Zuse was building the Z4 machine, he began to write a long document that he intended to be his doctoral thesis, possibly under the supervision of Prof. Alwin Walther at Darmstadt. The draft had a long title: "Contributions to a theory of general computation, with special consideration of the propositional calculus and its application to relay circuits."[2] The document explains the propositional calculus and how logic formulas can be mapped to logic circuits and vice versa. He examines the logic operations, the reduction of logic formulas to a normal form, and the duality principle present in DeMorgan's laws that allow the engineer to transform logic circuits into equivalent dual circuits.

In his draft, Zuse makes a distinction between "implicit" and "explicit" algebraic problems. If we only write the quadratic expression $ax^2 + bx + c = 0$, and ask for the roots of the polynomial, we have an implicit representation of the problem. If we derive the formula for the roots of the quadratic polynomial using the rules of algebra, we then have an explicit computational procedure for finding them.

[2] Ansätze einer Theorie des allgemeinen Rechnens unter besonderer Berücksichtigung des Aussagenkalküls und dessen Anwendung auf Relaisschaltungen

Zuse then considers the equivalent formulation for logic problems. Given a logic formula, how do we find the combination of logic truth values for the variables that makes the whole formula true? This is nowadays called the satisfiability problem. Zuse gives an example that shows that even for very simple formulas, trying to find a binary assignment of logic variables leads us to consider, in the end, all possible combinations of binary assignments for those variables. Today we know that the satisfiability problem for logic expressions is NP-complete, that is, there is no known algorithm that can solve the problem in polynomial time and the problem is equivalent to many other hard problems. Therefore, Zuse does not delve deeper into this issue and continues developing an abstract notation for relay circuits, which is used to examine the quasi-automatic "synthesis" of circuits given a logic expression for binary variables. The abstract relay notation used by Zuse is the same that he developed for the circuits of the Z1 and his other computers, also for his patent applications.

All this is relevant for understanding Plankalkül better. In the draft of the doctoral thesis, Zuse considers the logic quantifiers "for all" and "there exists" and thus the transition to the predicate calculus. He then talks about "rigid" and "non-rigid" programs. The first kind of program works with a fixed number of computations and without conditional branches. For example, given a logic formula of some variables, and the binary value of those variables, the truth value of the formula can be found simply by computing the sequence of logic operations and their relationships. But in a "non-rigid" (or "free") program, the computational path can arrive at junctions that determine alternative computational paths. At this point, Zuse's examples are more related to the predicate calculus rather than propositional logic.

In this sense, the Plankalkül draft of 1945 is only a continuation of the research begun in the dissertation draft. While propositional logic is the main topic in the first book of the dissertation draft, the predicate calculus with quantifiers becomes now the field of research (Zuse, 1945).

In the Plankalkül draft, Zuse writes that his intention was to develop a complete calculus for programs (hence the name of the document), but he also explains that this could not be fully achieved. He mentions that obtaining explicit logic expressions from implicit logic formulas is much harder than in the case of algebraic problems and that he did not pursue the subject further. In modern terminology, we would say that he uses the formalism of predicate logic to write specifications for computations, but does not attempt to prove expressions automatically. He thus distinguishes between the "implicit" and the "explicit form" of Plankalkül. The latter is the imperative form, which is similar to a high-level imperative language, as we describe in the following sections. The former is the logic expression that represents a program. If we ask for all elements $x$ in the set $V$ that satisfy a function $f(x)$, we would write the formula $(x)(x \in V, f(x))$ (using Hilbert's notation).

Zuse would have liked to have a general procedure that would automatically transform the implicit into the explicit form, but he did not have such a method. Therefore, he settles for a manual alternative that consists of writing the implicit form on top of programs written in Plankalkül (which is very descriptive of the computation at hand) followed by the explicit form obtained manually by the programmer. In this sense, Zuse writes: “In the following programs an implicit and an explicit form are given. But the implicit form represents a comment and the explicit form is the actual program” (Zuse, 1972). That is, we can interpret these comments as a logic specification of the computation in the imperative code that follows. However, we know how difficult it is to transform a specification into a program, and this is also Zuse’s difficulty, which is solved by doing the transformation by hand.

That being said, there are parts of the Plankalkül draft that seem to point toward the full calculus of programs that Zuse would have liked to have. It would be the equivalent of the logic synthesis approach in the draft of the thesis: given a logic formula, transform it into an equivalent relay circuit. Now it would be: given a predicate logic formula, transform it into an equivalent imperative program. Zuse mentions that he had an operator for assertions (for introducing axioms), which he did not use for code in the end. He writes: “The assertion symbol ⊢ ... does not belong to the syntax of PK proper. It only has meaning in comments. However, it shows that I originally planned to extend the PK as a real “calculus”, beyond its use as algorithmic language” (Zuse, 1972). Had Zuse developed the calculus he had in mind, he would have created the first automatic theorem proving system. Nevertheless, some authors think that Zuse stood out among early computer pioneers because of his keen interest in this issue (Bibel, 2020).

Zuse has a section in the draft of Plankalkül where he shows how to pre-transform formulas of the propositional calculus into a so-called “machine form” that makes the automatic evaluation easier. The M-Form of logic formulas is just a transformation of logic expression with parentheses to a kind of Polish notation that does not require parenthesis. It is a small part of the Plankalkül, but one that clearly shows the great interest Zuse had in the automatic synthesis of programs derived from the formulas of the predicate logic.

## 10.3 Symbolic Computation

Among the many examples of different kinds of computations in the Plankalkül draft, but there is one that is especially important because it represents the first fully symbolic computation ever written in a programming language. In the 19th century, Charles Babbage had already worked on computational problems with a symbolic background. For example, he was interested in the multiplication of polynomials, a problem at the root of computational algebra. Zuse, on the other hand, was interested in the automatic synthesis of programs, and since this was not possible in general, he settles for a smaller problem: given a formula of the predicate calculus with quantifiers, functions, and parentheses, check whether the formula is well-formed. A special case would be checking that arithmetic expressions are well-formed.

There is an immediate connection here to the Z4, which Zuse was building at the time. For the Z4, Zuse wanted to have a *Planfertigungsgerät*, which we can literally translate as “producer of programs,” that is a program synthesizer (Zuse, 1944e). His idea was that the users of the Z4 would have a keyboard on which they could write arithmetic expressions, with parentheses, functions, and operators, which would then be automatically converted into executable code for a punched tape. Zuse could not do this on time for delivery of the Z4 so the final result was a kind of first step. In the Z4, all operations and functions (such as square root or maximum of two numbers) had a button in the console. The operator pressed the sequence of operations he wanted to have for the punched tape, and the Z4 produced the opcodes needed. This was much easier than having to remember and punch the binary opcodes, as was the case with the Z3. Moreover, the Z4 could compute program code on its own and print the corresponding punched tape. This capability was certainly never used, but it shows how aware Zuse was of the possibility of the Z4 producing its own programs. If the user simply typed the arithmetic formulas on the keyboard, the Z4 could have acted as a compiler, translating the algebraic expressions into machine code. Again, this was never done, but the idea was clear, and the necessary equipment was available in the Z4.

Such a compiler would have been written in something similar to Plankalkül. In fact, Chap. 4 of the PK draft deals with the symbolic processing and checking of formulas of the propositional calculus. Zuse describes something akin to a grammar for valid expressions and then shows how to process a stream of characters coming from the keyboard for checking. He defines well-formed expressions and writes, for example, that a variable is an expression, a negated variable is also an expression, two expressions can be connected by an operator to produce an expression, and so on. It is just the kind of simple grammars that computer science students learn to use today to parse arithmetic expressions that are transformed into executable code.

So there we have it. Babbage's dream has finally come true: the first description of how computers could be used to solve not only numerical problems, but also algebraic equations and a wide range of symbolic computations necessary for the development of high-level programming languages and their compilers.

Looking back on Plankalkül today, we should not forget that it was originally much more than its second imperative half. The first half was a pioneering glimpse of what would evolve many decades later into logic programming and automatic theorem proving.

## 10.4 The Syntax of the Plankalkül

In the next sections, we will concentrate on the imperative form of Plankalkül, and we will describe the subset that we selected for the first implementation.

The original document describing Plankalkül (Zuse, 1972) is not free of contradictions, and several ambiguities need to be resolved before attempting to write a compiler for the language. Therefore, in what follows we have identified a powerful subset of the language that is computationally complete and free of ambiguities. In defining this subset, we were guided by the following principles:

- Historical accuracy. We wrote a syntax-driven editor that preserves the original two-dimensional structure of the language.
- Simplicity. Zuse left alternative syntactic options open at several points in the definition of the language. We retained only one option in each case, to make the syntax unambiguous, especially with respect to data types.
- Induction from examples. When Zuse did not clearly outline the syntax or operational semantics of language constructs, we inferred them from the numerous examples contained in the founding Plankalkül document.
- Regularity. When syntactic options were ambiguous, we chose one that made the language more regular and "orthogonal."
- Simple implementation. For the first subset of Plankalkül that we defined, we chose only those constructs that are easy to implement in a conventional computer. We left set and predicate logic constructs out of the selected language subset. These can be implemented later using macrodefinitions and a standard Plankalkül library.

We call the subset of Plankalkül obtained from these principles "Plankalkül 2000."

### 10.4.1 Variables and Data Types

Variables are essential for any imperative programming language. In Plankalkül there are three main classes of variable:

- V variables, numbered V0, V1, etc., which are read-only.
- Z variables, numbered Z0, Z1, etc., which can be read and written.
- R variables, numbered R0, R1, etc., which are write-only.

The V variables are used to pass parameters to programs, the Z variables hold intermediate results, and the R variables are used to return the final result of a subroutine. Additionally, there are "loop variables," which are used in While loops. They are denoted as $i_0$, $i_1$, $i_2$, etc., according to the depth of loop nesting, and are of generic numeric type. We will have more to say about these variables later.

All variables have a "structure" or type. The following data types are possible:

- One bit, denoted as "0"
- $n$ bits, where $n$ is an integer, denoted "$n$.0"
- Tuples of other types. For example, (3.0,4.0) denotes a pair of variables, one of 3 bits, the second of 4 bits. Tuples can have two or more elements.
- $m$ times any other type, for example, 4.5.0, which denotes an array of four elements, each one of five bits.

Some examples of possible data types are:

| | |
|---|---|
| • 8.0 | a byte |
| • 16.8.0 | a vector of 16 bytes |
| • (0, 8.0, 16.0) | a triple consisting of one bit, 8 bits, and 16 bits |
| • 32.(0, 8.0, 16.0) | an array of 32 triples with the structure above |

It is easy to see that data structures in Plankalkül can be implemented as trees. The last example given above represents a tree with 32 child nodes at the root level. Each child has three children, and so on. It is important to note that tuples are just another syntactical way of referring to arrays. We need tuples when the data types of the elements in the array are different. We use vectors when the data type of each element is the same. In Plankalkül 2000 all variables are vectors or tuples, or combinations of both.

There are no variable declarations at the beginning of a program. Instead, each variable carries its own type. Variables are generally written using four lines:

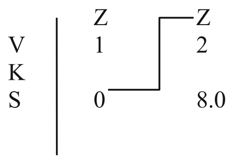
$$\begin{array}{r|l} & \mathrm{Z} \\ \mathrm{V} & 1 \\ \mathrm{K} & 0 \\ \mathrm{S} & 5.0 \end{array}$$

This example refers to the variable Z1 of type 5.0 (five bits). The subindex of the variable is written in the "V" line, the component of the variable in the "K" line, and the type in the "S" line. The annotations to the left of the vertical line are just a mnemonic device and are not part of the syntax. The variable Z1 is a vector of five bits. If we want to refer to the first bit in the vector we write:

$$\begin{array}{r|c} & \mathrm{Z} \\ \mathrm{V} & 1 \\ \mathrm{K} & 0 \\ \mathrm{S} & 0 \end{array}$$

Note that the components of arrays are numbered starting from zero. Also note that the type of the component selected in the example is a single bit. The VKS annotation can be omitted, as we do in the examples that follow.

Component indices can be variable. We can refer to the component of the variable Z1 whose number is stored in variable Z2 as follows:



The connecting line means that the content of variable Z2 (a byte) is used as an index for variable Z1. The indexed component is of type “0” (a bit).

### 10.4.2 Arithmetic and Logic Statements

The symbol $\Rightarrow$ is used to denote value assignment. Variable assignments are read from left to right, like in the following example of a Plankalkül statement:

$$\begin{array}{ccccc}
V & + & V & \Rightarrow & Z \\
0 & & 0 & & 2 \\
0 & & 2 & & \\
8.0 & & 8.0 & & 8.0
\end{array}$$

Here, component 0 of V0 and component 2 of the same array are added and the result is stored in variable Z2, which is an array of eight bits. The component line of Z2 is left empty because we want to refer to the entire array of eight bits. Only V and Z variables (and loop variables) can appear in expressions on the left of the assignment symbol.

The four basic arithmetic operations are defined in Plankalkül. We use the symbols +, $-$, $\times$, and / to denote addition, subtraction, multiplication, and division. Since each variable “carries” its type, the programmer must be careful to write only valid arithmetic operations, otherwise a runtime error will result. We adopt the convention that arithmetic and logic operations are only valid for arguments of the same type (generically, $n$.0). The result also has the same type as the arguments. We use two’s complement arithmetic to perform the operations.

There are logic operators for conjunction, disjunction, and negation, denoted with the symbols $\wedge$, $\vee$, and $\neg$ The conjunction of two bits, for example, can be written as:

$$\begin{array}{ccccc}
Z & \wedge & Z & \Rightarrow & Z \\
0 & & 1 & & 2 \\
 & & & & 1 \\
0 & & 0 & & 0
\end{array}$$

Here, we compute the conjunction of two variables Z0 and Z1 (single bits), and store the result in variable Z2, component 1. Variable Z2 is therefore an array of bits, and we are selecting only one of its components. Negation is expressed in Plankalkül by writing a dash above the name of a variable or an expression. For ease of implementation, we will use the unary operator $\neg$ to denote negation instead. Two other logic operators are defined in Plankalkül: the identity operator $\sim$ and the

XOR operator / $\sim$. Constants are written in Plankalkül in the first line of the tabular notation, like in:

$$\begin{array}{ccccc}
Z & + & 2 & \Rightarrow & Z \\
0 & & & & 2 \\
 & & & & \\
8.0 & & & & 8.0
\end{array}$$

We will always assume that the type of a constant is the type of the other variable argument or of the result (when two constants are combined). There are no arithmetic operations on tuples, but tuples can be assigned to tuples with the same number of elements.

## 10.4.3 Guarded Commands

There is a construct in the Plankalkül that could be interpreted in other high-level programming languages as an IF-THEN statement. It corresponds to deterministic guarded commands in some modern languages.

The symbol $\rightarrowtail$ is used to denote conditional execution; it separates a logic expression and a statement (Zuse used an arrow with a dot underneath). The statement to the right of the arrow is executed only if the logic value to the left of the arrow is true (that is, a 1). For example, the statement:

$$\begin{array}{ccccccccc}
Z & \wedge & Z & \rightarrowtail & V & + & V & \Rightarrow & Z \\
0 & & 1 & & 0 & & 0 & & 3 \\
 & & & & 0 & & 2 & & \\
0 & & 0 & & 8.0 & & 8.0 & & 8.0
\end{array}$$

means that if the conjunction of the two bits stored in Z0 and Z1 is true, the addition is performed, and the result is stored in Z3. Note that the arrow symbol binds more strongly than the assignment symbol, and the logic and arithmetic operation symbols, more strongly than the arrow. Brackets can be used to disambiguate expressions, like in the example below:

$$\begin{array}{ccccccccc}
(Z & \wedge & Z) & \rightarrowtail & (V & + & V) & \Rightarrow & Z \\
0 & & 1 & & 0 & & 0 & & 3 \\
 & & & & 0 & & 2 & & \\
0 & & 0 & & 8.0 & & 8.0 & & 8.0
\end{array}$$

Note that brackets open and close in the upper line. Statements are written using a line. A block of statements is marked as a unit by enclosing it in square brackets, which are as large as needed, for example, a block of two instructions, an addition, and a multiplication:

$$\left[\begin{array}{ccccc} Z & + & 2 & \Rightarrow & Z \\ 0 & & & & 2 \\ & & & & \\ 8.0 & & & & 8.0 \\ & & & & \\ Z & \times & Z & \Rightarrow & Z \\ 2 & & 1 & & 3 \\ & & & & \\ 8.0 & & 8.0 & & 8.0 \end{array}\right]$$

Conditions can be tested with the equality and inequality operators (=, >, <), which are used to check if the first argument is equal, larger, or smaller than the second. Any two structures can be tested for equality, but only structures that can be interpreted as numbers ($n$ bits) can be tested with the other two operators. We can store the greater of two numbers Z1 and Z2 in Z3 using the following instructions:

$$\begin{array}{ccccccccc} Z & \Rightarrow & Z & & & & & & \\ 1 & & 3 & & & & & & \\ & & & & & & & & \\ 8.0 & & 8.0 & & & & & & \\ & & & & & & & & \\ Z & < & Z & \rightarrowtail & Z & \Rightarrow & Z & & \\ 1 & & 2 & & 2 & & 3 & & \\ & & & & & & & & \\ 8.0 & & 8.0 & & 8.0 & & 8.0 & & \end{array}$$

## 10.4.4 Iterations

There is a kind of WHILE statement that is useful for performing iterations. The syntax of the construct is

W [Block]

where [Block] denotes a block of statements. In general, an iterative construct has the form:

$$\mathrm{W}\left[\begin{array}{ccc} \mathrm{C1} & \rightarrowtail & \mathrm{S1} \\ \mathrm{C2} & \rightarrowtail & \mathrm{S2} \\ \ldots & & \\ \mathrm{Cn} & \rightarrowtail & \mathrm{Sn} \end{array}\right]$$

The block is executed repeatedly until all conditions C1, C2, etc., tested inside the block, fail in a single run. The statements S1, S2, etc., are executed according to the truth value of their respective conditionals. The construct W0(num) preceding a block of instructions can be interpreted as an FOR operation: the block of instructions is executed "num" times. If we want to have access to a loop variable containing the current iteration number, we use the construct W1(num). A loop variable *i* runs from 0 to num-1. The loop variable is a special variable with an unspecified default numeric type and can only be accessed within the block following the W1 declaration. If nested loops are used, they are numbered using the index row and their loop variables also use these numbers.

$$\begin{array}{c} \mathrm{W1} \\ 0 \\ \phantom{0} \end{array}\left[\begin{array}{ccccc} \ldots & \mathrm{i} & \ldots & \mathrm{W1} & \\ \ldots & 0 & \ldots & 1 & \\ & & & & \end{array}\left[\begin{array}{ccc} \ldots & \mathrm{i} & \ldots \\ \ldots & 1 & \ldots \\ & & \end{array}\right]\right]$$

In the example above, the first loop has index 0 and the second loop has index 1. The loop variables are i0 and i1. They can only be used within the scope of their respective While loops. Zuse defined a built-in function that is very helpful when processing arrays. The function N applied to a variable returns the number of components of the variable as result. See below for an example of its application.

## *10.4.5 Examples of the Implicit Form of Plankalkül*

We said earlier that programs in Plankalkül can be specified using the implicit form, while the explicit form represents the concrete execution model. In this subsection, we look at the quantifiers and operators used in Plankalkül, and some examples of the implicit expressions that Zuse used.

The quantifiers and operators used by Zuse are the following:

- The all-quantifier. Instead of writing $\forall x$, Zuse used Hilbert's notation: $(x)$.
- The existence quantifier $(Ex)$.
- The operator "all those," written $\tilde{x}$, returns all elements that satisfy a given condition bundled in a set.

- The operator "all those, even when repeated," written $\tilde{\tilde{x}}$, returns all elements in a set that satisfy a condition, even if they are repeated.
- The operator "the only element such that," written $\acute{x}$, is like $\tilde{x}$ but there can be only one element fulfilling the condition.
- The operator "next element," written $\mu x$, returns the next element in a set that fulfills a condition. Since Zuse used lists to represent sets, there is an implicit order in the elements.

The following examples show how the quantifiers and operators were used. If we want to express that for all $x$ in a set $V$ the logic predicate $R(x)$ is true, we write:

$$(x)(x \in V \rightarrow R(x)).$$

If we want to say that there exists an $x$ in the set $V$ such that $R(x)$ is true, we write:

$$(Ex)(x \in V \wedge R(x)).$$

We can collect in a subset Q all elements in a set for which $R(x)$ is true by writing

$$\tilde{x}R(x) \Rightarrow Q.$$

We collect elements, including repetitions, with $\tilde{\tilde{x}}$. If we want to store in Q the single element that satisfies $R(x)$, we write

$$\acute{x}R(x) \Rightarrow Q.$$

If we want to go through the elements in a set and pick the next element fulfilling $R(x)$ we write

$$\mu x \ R(x) \Rightarrow Q.$$

An example from the section in Plankalkül that deals with chess could be the following: "given two different squares A and B on the chess board, check if there is another square C, so that A and C are in knight relationship, and also B and C":

$$(Ex)(R(A, x) \wedge R(B, x) \wedge (A \neq B)$$

Here the predicate $R$ checks the knight relationship. It is a subprogram that we have written before. The set from which the squares are being taken is the complete chessboard. This set is implicit in the above expression. The examples have been simplified using functions and variables with a single-letter name and without specifying their type.

### *10.4.6 Linearized Form of the Plankalkül*

To simplify the rest of the chapter, we adopt a linearized form of the Plankalkül, in which variables are written as in the following examples:

| | |
|---|---|
| `V0[1:5.0]` | Variable V0, component 1, of type 5.0 |
| `Z1[5.3:9.0]` | Variable Z1, component 3 of component 5, of type 9.0 |

Some special symbols are written using ASCII characters or combinations of them. Conjunction, disjunction, and negation are represented by the characters “&”, “|” and “!”. Assignment is expressed using “=>” and the conditional arrow is written “->”. The conditional expression in the section about guarded commands can be simplified by writing:

```
Z0:0 & Z1:0 -> V0[0:8.0] + V1[2:8.0] => Z3[:8.0]
```

This is much more convenient than the four-line syntax of the Plankalkül draft. We use square brackets to enclose blocks of statements, and the semicolon as a separator between statements to write more than one statement on each line. An instructive example is to compute the sum of the bytes stored in an array V0 of type 16.8.0. The following statements would accomplish this task:

```
0 => Z1[:8.0]
W1(16) [ Z1[:8.0] + V0[i:8.0] => Z1[:8.0] ]
```

Note that we don’t write the type of the constant 16 and the type of the loop variable. They are generic numeric variables.

### *10.4.7 Functions and Function Calls*

Programs in Plankalkül are functions that can be called from other programs. Each program is assigned a unique number. The declaration of input and output variables is done in the “Randauszug” (“boundary summary,” i.e., the program interface), which, for example, has the following form:

| | | | | | | |
|---|---|---|---|---|---|---|
| P3 | R | (V, | V) | ⇒ | (R, | R) |
| | | 0 | 1 | | 0 | 1 |
| | | | | | | |
| | | 8.0 | 8.0 | | 4.0 | 4.0 |

In this example, program P3 is defined with two input variables V0 and V1, each of 8 bits, and two output variables R0 and R1, each of four bits. The number of input and output variables in the Randauszug is variable. Zuse always numbered the input variables from 0 to $n - 1$ and the output variables from 0 to $m - 1$, where $n$ and $m$ are the number of input and output variables, respectively. The identifier for the program in the example above is just an “R”. When this program is called as a function, we write “R3” with the two variable arguments enclosed in parentheses. Note that in the Randauszug, the input and output variables are written without the component row, that is, they can only be used as *complete* variables.

Functions can also have a symbolic identifier. For example, if we want to define a function to select the maximum of two bytes, we could write the following Randauszug

$$\begin{array}{cccccc}
\mathrm{P5} & \mathrm{Max} & (\mathrm{V}, & \mathrm{V}) & \Rightarrow & \mathrm{R} \\
 & & 0 & 1 & & 0 \\
 & & & & & \\
 & & 8.0 & 8.0 & & 8.0
\end{array}$$

followed by the appropriate block of instructions. We will write the Randauszug in the following linearized form:

```
P5  Max (V0:8.0,V1:8.0) => R0[:8.0]
```

When a function is called with a subindex we select the component of the output we are interested in:

$$\begin{array}{ccc}
\mathrm{R3} & (\mathrm{Z}, & \mathrm{Z}) \\
0 & 1 & 3 \\
 & & \\
4.0 & 8.0 & 8.0
\end{array}$$

This is a call to program P3 above, which computes two result variables, R0 and R1. In the call, we select variable R0 of type 4.0 from the result tuple. In the linearized form we write for this call: R3(Z1:8.0, Z3:8.0)[0]:4.0 ⇒ Z4:4.0. Finally, although Zuse did not signal the end of a program with any special keyword, we will write END at the end of every program.

### 10.4.8 Input and Output

Zuse did not define any primitive instructions for input and output. He seems to have considered this type of instruction to be machine-specific and not part of the main language constructs. In our implementation of the language, we have not defined any input/output instructions. The user can inspect and modify the state of the variables stored in memory by opening a memory window, which is specific to each program since Plankalkül variables are local. The example below shows a program that computes the maximum of three variables by calling the function max.

```
P1 max3 (V0[:8.0],V1[:8.0],V2[:8.0]) => R0[:8.0]
    max(V0[:8.0],V1[:8.0]) => Z1[:8.0]
    max(Z1[:8.0],V2[:8.0]) => R0[:8.0]
END

P2 max (V0[:8.0],V1[:8.0]) => R0[:8.0]
    V0[:8.0] => Z1[:8.0]
    (Z1[:8.0] < V1[:8.0]) -> V1[:8.0] => Z1[:8.0]
    Z1[:8.0] => R0[:8.0]
END
```

## 10.5 Implementation Issues

There are several syntactical aspects of Plankalkül that must be dealt with in any implementation. The programmer writes the type of the variables each time they are used, and this can lead to inconsistencies. We decided that each variable in a program has a unique type, which cannot be "casted" into another type. This leaves open the possibility of writing the type of a variable only once and using type inference at any other point in the program where it is used. However, type inference is made a bit more difficult because we can refer to a component of a variable before we refer to the variable itself. For example, a reference to V0[1.1]:8.0 may appear in a statement before a reference to V0:16.8.8.0. Therefore, the type inference routine must look at the entire program before deciding on the type of a variable. We did not implement type inference in the first implementation of Plankalkül 2000 so the programmer has to write the type of each variable or component that is used. We decided to refer to components of variable calls by using square brackets in the linearized form of the Plankalkül. We also decided that the type of the result must always be written. Zuse usually did not write the type of the result of a variable call, since it can be inferred from the definition of the language. In the first implementation of Plankalkül 2000, we always write the type of variable or variable call.

### *10.5.1 The Editor*

The two-dimensional syntax of Plankalkül is very difficult to handle with a conventional editor. Therefore, we developed a syntax-driven editor that allows the user to write a program by selecting options from menus. Figure 10.1 shows the start window of the editor. By clicking on the "Plan" keyword, it is possible to get a layout for a new program, which includes the program number, the Randauszug, and the instructions. Figure 10.2 shows the state of the window after several selection steps. The user has selected program number 1 from a menu and 4 and 3 variables for the input and output in the Randauszug. The user can now select a statement

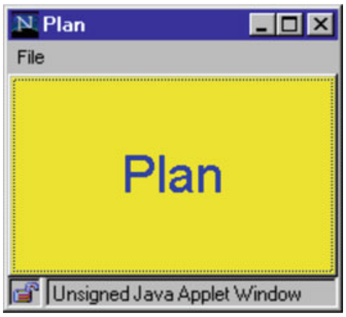


**Fig. 10.1** Start window of the syntax directed editor

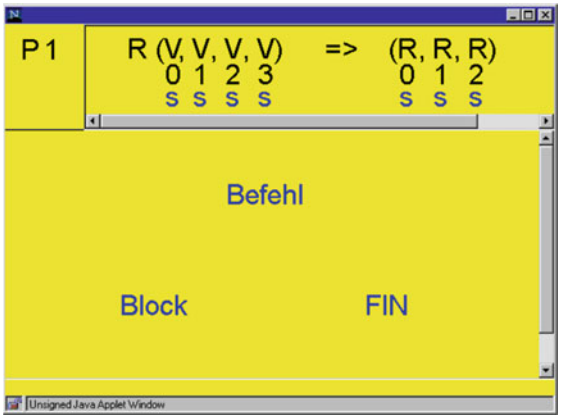


**Fig. 10.2** State of the editor after some selections

from several options ("Befehl"), a new block of statements ("Block"), or end the sequence of statements ("FIN").

Basically, the options offered by the menus are the only ones valid under the defined grammar. One can think of this syntax-driven editor as one that only allows the user to develop the Backus–Naur form for the programming language we are considering. In this respect, this editor is similar to those written by Teitelbaum and Reps (1981), by Arefi et al. (1990), and others. The user is restricted to stay within the boundaries of the Plankalkül grammar and cannot write invalid programs. Any program written with this editor is grammatically correct, although it may of course be semantically incorrect. The syntax-driven editor produces a linearized version of the Plankalkül program. Thus, the programmer can write his program directly in linear form using any kind of editor or use this two-dimensional editor to produce the linear code. Although both possibilities are open to the programmer, the syntax-driven editor should give to the user the "look and feel" of writing Plankalkül programs in the original syntax.

The editor was written in Java and was operational for several years on our website for the project.

### *10.5.2 The Parser*

The syntax of Plankalkül 2000 is summarized in Appendix A. We wrote a parser for this syntax using the public domain version of the Cocktail compiler generator system (Grosch and Emmelmann, 1991). The figure below shows the structure of the whole Plankalkül system: the syntax-driven editor transforms the two-dimensional code into the linearized version of Plankalkül described in this document. The parser then transforms this code into a simpler textual representation of the program that we call the "intermediate code." This intermediate code is then interpreted by the runtime system. This allows the user to set the values of variables through an interactive user interface (Fig. 10.3).

The parser produces not only intermediate code for the runtime system but also TeX code that can be interpreted and sent to a PostScript printer.

### *10.5.3 The Runtime System*

The runtime system was written in Java. When the system starts, a window displays the contents of the memory variables. This can be changed interactively by the user. Figure 10.4 is an example of the state of the memory after running a sorting program. The first row in the window shows an array of five numbers, each of 8 bits. The ones are shown as full circles (the least-significant bits are written to the left). The decimal equivalent is written below each element of the array. The last row shows the result of the sorting routine. The rows in the middle are intermediate (Z) variables.

Before the program starts, the user can modify the values of the V variables by clicking on the individual bits. After the program runs, the user can inspect the result

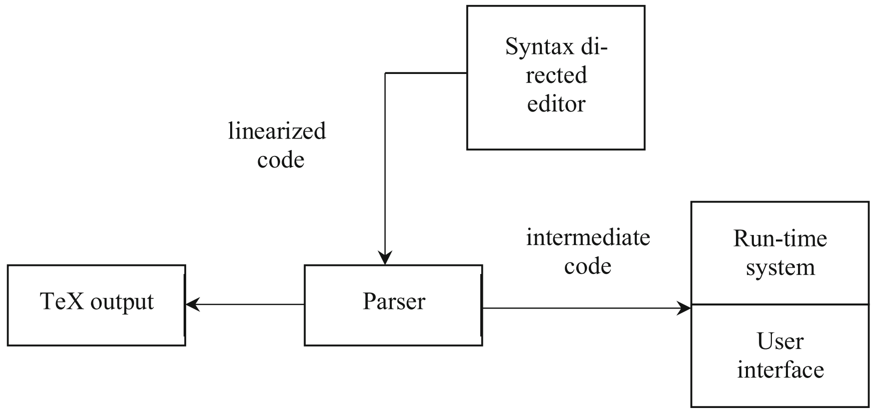


**Fig. 10.3** Structure of our Plankalkül system

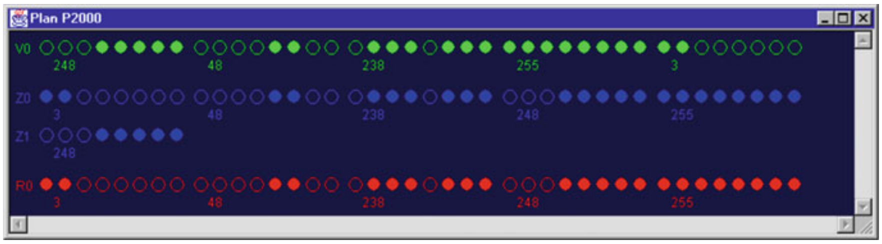


**Fig. 10.4** The result (last row) of sorting the V variables (first row)

variables. The original definition of Plankalkül does not include any input–output instructions. Zuse left this part of the language undefined. Curiously, this is also the case in modern programming languages, where input and output routines are defined in a special system library.

## 10.6 Sample Programs

In this section, we provide some examples of Plankalkül 2000 programs written as linear code. All these programs have been parsed and executed by our system. Program P1 assigns the conjunction of two input variables to the result variable.

```
P1     R(V0[:0],V1[:0]) => R0[:0]
       V0[:0] & V1[:0] => R0[:0]
END
```

Program P2 computes the expression a+b*c.

```
P2     R(V0[:16.0],V1[:16.0]) => R0[:16.0]
       V0[:16.0] + V1[:16.0] x V1[:16.0] => R0[:16.0]
END
```

A variation of the program above (to test syntactic alternatives).

```
P3     R(V0[:16.0],V1[:16.0]) => R0[:16.0]
       (V0[:16.0] + V1[:16.0]) * V1[:16.0] => R0[:16.0]
END
```

Another variation.

```
P4     R(V0[:16.0],V1[:16.0]) => R0[:16.0]
       (V0[:16.0] * 6)+(V1[:16.0]*V1[:16.0]) => R0[:16.0]
END
```

Program P5 computes the factorial of 5 (the generic type is 32.0)

```
P5     R(V0[:32.0]) => R0[:32.0]
       1 => Z0[:32.0]
       W1 (5) [
            i * Z0[:32.0] => Z0[:32.0]
            ]
      Z0[:32.0] => R0[:32.0]
END
```

Program P6 sorts 16 numbers using insertion sort.

```
P6 sort (V0[:6.8.0]) => R0[:6.8.0]

W1[0](4)
    [
    V0[i0:8.0] => Z0[i0:8.0]
        1 => Z4[:32.0]
    W1[1](i0)
    [
    (V0[i0:8.0] < Z0[i1:8.0]) & (Z4[:32.0]=1) ->
    [
      i0-i1 => Z1[:32.0]
      W1[2](Z1[:32.0])
        [
        i0 - i2 - 1 => Z3[:32.0]
        i0 - i2 => Z2[:32.0]
        Z0[Z3[:32.0]:8.0] => Z0[Z2[:32.0]:8.0]
        ]
      V0[i0:8.0] => Z0[i1:8.0]
      0 => Z4[:32.0]
    ]
    ]
    ]
END
```

## 10.7 Conclusions

It is unfortunate that programs written in Java at some point became insecure for the Internet so that many browsers will not run legacy code anymore. Our implementation of Plankalkül was operational for several years. A modern implementation would have to take into account the evolving standards of the Internet. This chapter offers enough detail for anyone interested in developing his or her runtime system for Plankalkül 2000.

In retrospect, the definition of Plankalkül reads like a miracle, given that the year of the draft is 1945. Plankalkül anticipates many constructs and data structures now common in computer science. The use of Plankalkül both as a logic specification and as an imperative language is fascinating. The grounding of the language in predicate calculus anticipated later developments, such as logic programming and theorem proving. This does not mean that Zuse knew how to develop proof systems, but it shows the depth of the theoretical ground on which Plankalkül was built.

Although Zuse tried to disseminate his work within the emerging community of computer experts by publishing the main features of Plankalkül in scientific journals

(Zuse, 1948), his work received little attention. It would take almost 10 years until computer companies started developing compilers for programming languages, such as FLOW-MATIC (1955), from which COBOL (1959) was derived. FORTRAN appeared in 1957 and ALGOL in 1958. Although some participants in the ALGOL meetings had some superficial knowledge of Plankalkül, the language was never considered a serious candidate for further development. It was virtually unknown until 1972 when the German Society for Mathematics and Computing published the draft finished in 1945.

For our story in this book, Plankalkül represents the highest point in the computational hierarchy defined by Konrad Zuse, as we show in the last chapter of this volume. It is an astonishing achievement considering that Zuse was building several computers at the same time (Z4, S1 and S2), trying to keep his company afloat, and all of this happening under wartime conditions. The world around was crumbling—amidst the rubble, the age of computing was emerging.

## Appendix: Syntax of the Implementation of Plankalkül 2000

In the following, we adopt these conventions: a vertical bar (|) separates optional syntactical elements, {expr}* means that expr can be concatenated zero or more times, all identifiers with a defined rule can be expanded, any other characters are included literally in the expanded expressions.

### *Symbols*

```
digit ::= 0 | 1 | 2 | ... | 9
digits ::=digit {digit}*
letter ::= a | b |...| A | B |...| Z
type-letter ::= a | b | ...h | j |... | A | B | ...| Z
identifier ::= letter {letter | digit}*
pos-constant ::= digits
neg-constant ::= - digits
constant ::= pos-constant | neg-constant
dot ::= "."
comma ::= ","
```

I.e. `type-letter` does not contain “`i`”.

### *Data Types*

```
simple-type ::= 0
tuple-type ::= (type, type {comma type}*)
```

```
type ::= simple-type | tuple-type | digits dot type
var-type ::= type-letter dot type
all-type ::= type | var-type
```

## Variables

```
v-variable ::= V digits [component: type] | V digits[: all-type]
z-variable ::= Z digits [component: type] | Z digits[: all-type]
r-variable ::= R digits [component: type] | R digits[: all-type]
loop-var ::= "i" | "i" digits
loop-expr ::= loop-var | loop-var + pos-constant |
              loop-var - pos-constant
type-var ::= type-letter
type-expr ::= type-var | type-var + pos-constant |
              type-var - pos-constant | type-expr + type-expr |
              type-expr - type-expr
component ::= digits | v-variable | z-variable | loop-expr |
              type-expr | component dot component
```

## Function Call

```
zv-call-arg ::= v-variable | z-variable | call | constant |
                loop-var | type-var
call-all ::= R digits [:type] ( zv-call-arg {,zv-call-arg}*)|
identifier [:type] ( zv-call-arg {,zv-call-arg}*)
call-one ::= R digits[component : type](zv-call-arg{,zv-call-arg}*)|
           identifier[component : type](zv-call-arg{,zv-call-arg}*)
call ::= call-all | call-one
```

## Arithmetic Operations

```
arith-argument-left ::= v-variable | z-variable | constant |
                        loop-var | type-var | call | arith-operation |
                        (arith-operation)
arith-argument-right ::= v-variable | z-variable | pos-constant |
                         (neg-constant) | loop-var | type-var | call |
                         arith-operation | (arith-operation)
arith-argument ::= arith-argument-left | arith-argument-right
arith-operation ::= arith-argument-left {+|-|x|/} arith-argument-right
```

## *Logic Operations*

```
log-constant ::= + | -
condition ::= arith-argument = arith-argument |
              arith-argument > arith-argument |
              arith-argument < arith-argument |
              zv-tuple = zv-tuple
pos-literal ::= v-variable | z-variable | log-constant | call |
                condition | (condition)
neg-literal ::= !v-variable | !z-variable | !call | !(condition)
logic-argument ::= pos-literal | neg-literal | logic-operation |
                   (logic-operation)
logic-binary ::= logic-argument { "|" | & | ~ | /~} logic-argument
logic-operation ::= pos-literal | neg-literal | logic-binary |
                    !(logic-binary)
```

## *Assignment*

```
assignment0 ::= arith-argument => {z-variable | r-variable}
assignment1 ::= logic-argument => {z-variable | r-variable}
assignment2 ::= zv-tuple => zr-tuple
assignment3 ::= zv-tuple => {z-variable | r-variable}
zv-tuple ::= ( zv-arg, zv-arg {comma zv-arg}*)
zv-arg ::= v-variable | z-variable | constant | call | loop-var |
           type-var | zv-tuple
zr-tuple ::= ( zr-arg, zr-arg {comma zr-arg}*)
zr-arg ::= r-variable | z-variable | zr-tuple
assignment ::= assignment0 | assignment1 | assignment2 |
               assignment 3
```

## *IF-THEN*

```
if-then ::= logic-argument -> statement
```

## *WHILE*

```
block ::= [ statement{; statement}*]
while ::= w block | w [digits] block | w1 (arith-arg) block |
          w1[digits] (arith-arg) block
```

### *Statements*

```
built-ins ::= FIN | FIN digits
statement ::= assignment | if-then | while | block | built-ins
```

### *Programs*

```
program ::= P digits randauszug {statement }* END
```

### *Randauszug*

```
randauszug ::= identifier v-tuple => r-tuple
v-tuple ::= v-variable | (v-variable {, v-variable}*)
r-tuple ::= r-variable | (r-variable {, r-variable}*)
```

// The variables are numbered sequentially, starting with 0
// constant, indices, N(), have generic type

## References

Arefi, F., Hughes, C.E., and D.A. Workman. 1990. *Automatically generating visual syntax-directed editors*. *Communications of the ACM* 33(3): 349–360. https://doi.org/10.1145/77481.77487.

Bibel, W. 2020. On the development of AI in Germany. *Künstliche Intelligenz* 34(2): 251–258. https://doi.org/10.1007/s13218-020-00654-x. Open Access https://rdcu.be/b3oxS.

Grosch, J., and H. Emmelmann. 1991. A tool box for compiler construction. In *Compiler Compilers. CC 1990*, ed. D Hammer. Lecture Notes in Computer Science, vol. 477, 106–116. Berlin, Heidelberg: Springer. https://doi.org/10.1007/3-540-53669-8-77.

Rojas, R., Göktekin, C., Friedland, G., Krüger, M., Langmack, O., and D. Kunißs. 2000. Plankalkül: The First High-Level Programming Language and its Implementation. Technical Report B-3/2000, Freie Universität Berlin.

Teitelbaum, T., and T. Reps. 1981. The cornell program synthesizer: A syntax directed programming environment. *Communications of the ACM* 24(9): 563–573. https://doi.org/10.1145/358746.358755.

Zuse, K. 1944e. Planfertigungsgeräte. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0220/.

Zuse, K. 1945. Theorie der Angewandten Logistik, 2. Buch. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0233/.

Zuse, K. 1948. Über den Plankalkül als Mittel zur Formulierung schematisch kombinativer Aufgaben. *Archiv der Mathematik* 1(6): 441–449.

Zuse, K. 1972. *Der Plankalkül*, vol. 63. Sankt Augustin: Berichte der Gesellschaft für Mathematik und Datenverarbeitung.

# Chapter 11
# Zuse's Computer for Binary Logic

*This chapter deals with the architecture of Konrad Zuse's "logistische Maschine" (logic machine). The computer was conceived by Zuse around 1944 as a minimal architecture that could nonetheless implement all operations available in his previous "algebraic machines," that is, conventional numerical computers. The logic machine featured a memory consisting of one-bit words and a processor with two one-bit registers. The CPU operations were limited to the conjunction and disjunction of the two one-bit registers (each register could be negated beforehand, if desired). The program was stored in a read-only punched tape. As we show, the machine was as powerful as Zuse's algebraic machines, but only in theory. In practice, the simplicity of the hardware would have resulted in extremely long programs.*

## 11.1 Introduction

It was while designing his machines Z1, Z2, and Z3 (1936–1941) that the German inventor Konrad Zuse gradually became aware that the instruction set of a digital computer could be reduced to sequences of logic operations acting on single bits and pairs of bits. The addition of two 8-bit numbers, for example, can be reduced to the manipulation of each bit in the numbers' binary representations, executing the necessary sequence of negations, conjunctions, and disjunctions, in the appropriate order. In other words: one can add two 8-bit numbers going bit-column by bit-column, from right to left, using both numbers, adding each binary column, and propagating the carry to the left. In Zuse's "algebraic machines," such as the Z3, the four elementary arithmetical operations were implemented in hardware as bit-parallel operations. The elementary bit-parallel operations provided by the CPU were addition/subtraction of two binary numbers and shifting of one number. Complex operations were implemented from sequences of these elementary

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_11

"microinstructions." A rotating dial was employed to selected one microinstruction after another, once per cycle. For example, division in the Z3 required 18 cycles to complete (Rojas, 1997) (see Chap. 5).

Therefore, since an instruction can be reduced to simpler microinstructions, Zuse realized that a minimal computer, that is, one able to work only on at most two bits at a time, could execute each of the four arithmetic operations, provided that one can write the corresponding program. Zuse called it the "logistische Maschine" (he used Logistik as a synonym for Logik). This idea is an integral part of Zuse's passion for what would now be recognized as a high-level programming language, the "Plankalkül." In this language, designed by Zuse between 1941 and 1945, all data structures are arrays or tuples of single bits, each of which can be accessed individually by a program (all bits are indexed elementary components of a data structure). Therefore, it was natural to consider a machine capable of sequentially handling single bits and emulating the algebraic machines in software, at a minimal cost. The idea of designing a "logic machine" was thus closely intertwined with the conception of the Plankalkül (Zuse, 1972).

During the development of his different machines, Zuse made an early distinction between "algebraic machines," suited for floating-point computations, and "logic machines," i.e., those dedicated to solving logic problems and capable of symbolic processing. One recurring example mentioned by Zuse is the parsing and interpretation of expressions from predicate calculus (Zuse, 1972). According to Zuse, a problem can be solved in an "explicit form," for example, when we use a formula to compute the roots of a quadratic polynomial, or it can be stated in an "implicit form" if we only specify the quadratic equation and ask for its roots. Within this framework, the implicit terms for logic computers would be expressions of the predicate calculus, which could be reduced automatically to their explicit form, i.e., an imperative program. Zuse maintained this distinction between algebraic and logic computers all the time, but the meaning of the latter term gained a more special connotation toward 1944. Plankalkül was a language for solving both numerical and symbolic problems, as exemplified by Zuse's code. He provides examples of the bit-by-bit computation of floating-point operations and the extraction of the square root of an integer. In his examples for logic operations, a minimal computer could represent the underlying hardware.

In 1944, Konrad Zuse submitted a patent application for the logic machine, and he made a similar application again in 1947 (Zuse, 1944d). Zuse had a habit of filing for patents at an early stage to protect his ideas. However, the war and the closure of the German Patent Office for several years made it impossible for him to benefit from such early applications. When the patent office finally reopened, some of his patent applications were denied, and the ones that were approved had little to no commercial impact. The patent for the "logistische Maschine" was granted in Austria in 1952, but had no commercial implications for Zuse's company (Zuse, 1952a).

In this chapter, we delve into the details of the "logic machine." We review its general structure and discuss, toward the end, whether this machine was "arithmetically complete," that is, if it had the capability to perform all the fundamental arithmetic operations that were executed by machines like the Z3.

## 11.2 General Structure of the Logic Machine

The logic machine was characterized by its remarkable simplicity: it consisted of an addressable memory for storing single bits, a tape reader, and a small CPU (Fig. 11.1). In modern terms, we would say that the word length was one bit. The processor could read and store single bits in memory. The program was stored on the punched tape encoded in binary format. The tape could be of variable length. There was also the theoretical possibility of attaching both ends of the punched tape to create a single "loop" of instructions.

The processor of the logic machine featured just two registers, A and B: each of them could be loaded with a single bit from a memory address. The first bit loaded into the processor was stored in register A, and a flag (referred to as Pr by Zuse) was set. The second bit loaded into the processor would automatically be directed to register B based on the previously set flag.

In Fig. 11.1 we see the memory on the left, the program in a punched tape on the right, and the processor in the middle. The instructions were read one by one. Load operations were used to load the registers, while the store operation sent the contents of register A to a memory location.

The processor could only execute two register operations: OR (disjunction) and AND (conjunction) of the two one-bit registers. Before executing the conjunction or disjunction, the contents of each register could be negated if desired. The opcode

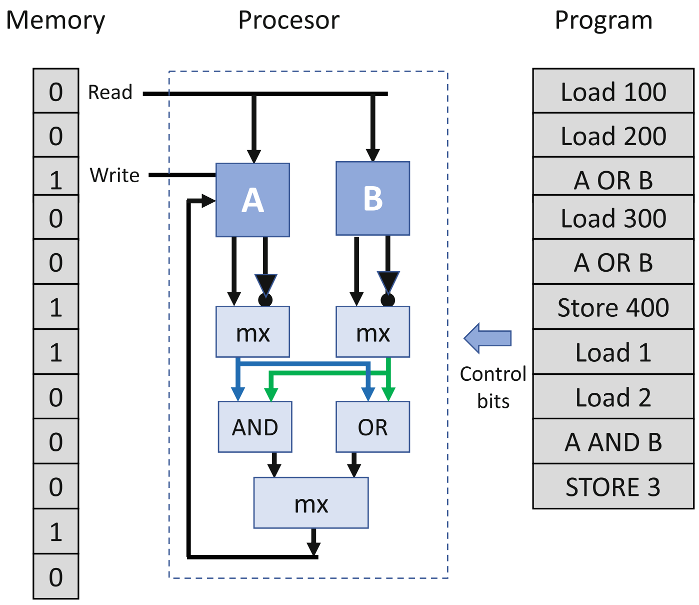


**Fig. 11.1** Block diagram of the logic machine

of the instruction being executed completely specified all such combinations. In Fig. 11.1 we can see that register A is read through one connection and negated through another. A multiplexer selects either A or the negation of A. The same applies to register B. The control bits for both multiplexers come from the opcode of the instruction that has been read from the punched tape. Once both arguments have been selected, the AND and OR operations are executed, and a third multiplexer selects only one of the two results. The control bit for the multiplexer is provided by an additional bit from the opcode.

The machine operated by first retrieving a command from the punched tape. There were three types of command: logic operations, load operations from memory, and store operations to memory. The result of a load operation went to register A or B, as explained above. The result of a logic operation was stored back to register A (which was flagged as occupied). A store operation would send the bit in register A to the specified address and declare the register empty (so that it could be loaded again).

A processor cycle consisted of five subcycles numbered I, II, III, IV, and V. A conducting line labeled III, for example, received a voltage only during subcycle III. The operation of the relays was synchronized by activating them only at the specified times. Bits could be stored over long periods using "self-halting" relays, that is, relays with two solenoids. When the first solenoid was energized, the relay closed, which in turn energized a second solenoid that kept the relay closed as long as there was power available. A "clear" signal just disconnected both solenoids from the power supply. As usual, in Zuse's diagrams, the relays are always drawn in the "zero" position. A switch to the position "one" moved the relay to the other contact (see the diagrams below).

The complete instruction set of the logic machine is summarized in Table 11.1 (where the symbol ¬ means negation):

**Table 11.1** Instruction set of the logic machine: LOAD and STORE operate with the memory. The possible logic combinations of the two register bits A and B are shown

| Memory operations | | | |
|---|---|---|---|
| LOAD n | | | Load a bit from memory address n to register A, if Pr flag is zero, and set Pr to 1. Otherwise, load the bit to register B |
| STORE n | | | Store register A to memory address n and set Pr to 0. Clear registers A and B |
| Logic operations | | | |
| A | AND | B | Conjunction of A and B |
| ¬A | AND | B | Conjunction of ¬A and B |
| A | AND | ¬B | Conjunction of A and ¬B |
| ¬A | AND | ¬B | Conjunction of ¬A and ¬B |
| A | OR | B | Disjunction of A and B |
| ¬A | OR | B | Disjunction of ¬A and B |
| A | OR | ¬B | Disjunction of A and ¬B |
| ¬A | OR | ¬B | Disjunction of ¬A and ¬B |

This implicit addressing of register A or B and their reset after a logic operation allows us to also have single-bit operations with the CPU. The negation of memory address n, for example, can be obtained from:

```
LOAD n
¬A OR B
STORE n
```

Here we have assumed that both registers were cleared before the code starts.

## 11.3 Encoding of the Instruction Set

In Zuse's patent application of 1944, each control word is eight bits long. The bits of the opcode are named $t_1$ to $t_8$. The upper two bits of the opcode select the kind of operation. The following two-bit combinations are possible:

| $t_1$ | $t_2$ | |
|---|---|---|
| 0 | 0 | NOP |
| 0 | 1 | logic operation |
| 1 | 0 | load operation |
| 1 | 1 | store operation |

In the case of load or store operations, the rest of the word specified the memory address. In the case of a logic operation, the next bit specified a conjunction or disjunction:

| $t_3$ | |
|---|---|
| 0 | conjunction |
| 1 | disjunction |

The bit $t_4$ specifies if register A, or its negation, should be operated on ($t_4$ has the value 1 or 0, respectively). The bit $t_5$ specifies the same for register B.

For example, the instruction "not(A) OR B" (which is equivalent to A implies B) would be represented with the following eight bits

01101XXX

where each bit X can be a 1 or a 0 (the extra bits were used for addresses, not needed in the case of a logic operation). With this eight-bit coding only 64 addresses can be specified. More addresses can be obtained by simply extending the opcode.

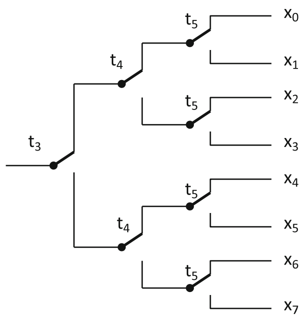


**Fig. 11.2** The decoding tree for the address $x_i$ of a memory cell. The $t$'s are bits from the opcode. In this example, only three bits from the opcode are interpreted, so that in total only eight memory cells can be addressed. For more addresses, the opcode has to contain enough bits, and the decoding tree has to grow accordingly

## 11.4 The Control and Decoding Unit

The control unit has to interpret the opcode of the instruction word—it is distributed in several parts of the machine. Bits $t_1$ and $t_2$ of the opcode activate or deactivate the memory unit. When the memory unit is inactive, the processor is active, and vice versa. If a memory operation is needed, the memory address is decoded using a decoder tree (which Zuse called a "Tannenbaum" (fir tree) circuit). Figure 11.2 shows the decoding circuit for an example with three-bit addresses. The values of the bits $t_3$, $t_4$ and $t_5$ select one of the eight possible memory addresses. The result of the address decoding is stored in self-halting relays named $x_0$ to $x_7$.

## 11.5 The Memory Unit

The memory of the machine consists of $m$ addressable single bits. In our example from the patent application, the memory unit would have 64 single-bit words addressable with six bits. In the example in Fig. 11.3 three address bits have been decoded and activate one of the relays $x_0$ to $x_7$. The relays $c_0$ to $c_7$ represent the content of the addresses 0–7. For example, a load operation ($t_1 = 1$, $t_2 = 0$) from address 3 would activate $x_3$ and load the contents of relay $c_3$ to register A, or register B, according to the state of the relay Pr. When Pr = 0, the first register A is selected as storage for a bit read from memory. When Pr = 1, register B is selected. Pr is always zero at the beginning and flips to 1 immediately after a load. It is cleared after a store operation.

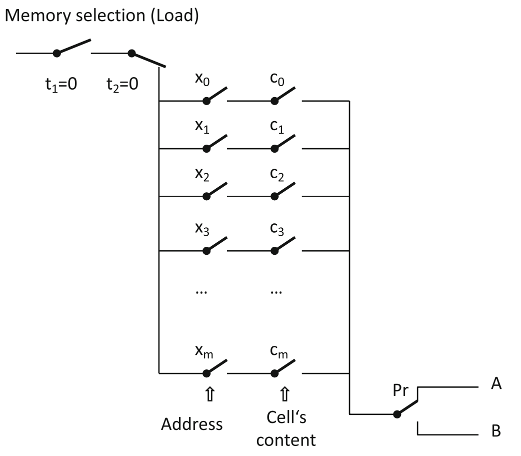


**Fig. 11.3** A LOAD operation reads the bit $c_i$ from memory address $x_i$ and stores it in Register A or Register B, according to the condition flag Pr

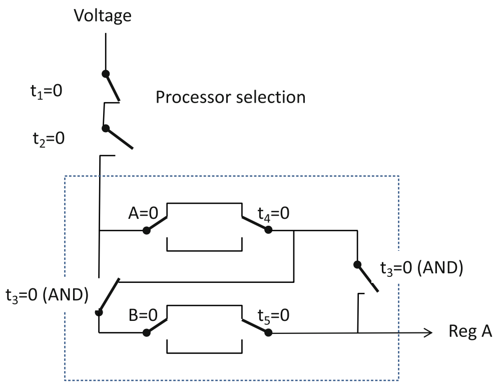


**Fig. 11.4** The processor of the logic machine (enclosed in a box) contains just the contacts of five different relays

## 11.6 The Processor

Figure 11.4 shows the processor of the logic machine. We can see the state of the two registers (represented by the state of relays A and B). When $t_1 = 0$, $t_2 = 1$, and $t_3 = 0$, we obtain the result of the operation (A AND B). When $t_1 = 0$, $t_2 = 1$, and $t_3 = 1$ we obtain the result of (A OR B). The relay $t_4$ inverts register A, when $t_4 = 0$, while $t_5 = 0$ inverts register B. The result is stored back in Register A.

## 11.7 Was the Machine Arithmetically Complete?

One important question to ask is whether this machine could really compute all four arithmetical operations, working bit by bit with the numbers stored in the memory unit. The answer is: yes.

To begin with, since subtraction can be reduced to the addition of two's-complement numbers, we first need to consider the addition operation. The addition of two bits A and B, including also a carry C, can be computed as ((A XOR B) XOR C). The carry can be computed with the logic expression ((A AND B) OR (A AND C) OR (B AND C)). Since all these operations can be executed by the logic machine, and since we can proceed from the lower to the higher bits in two 32-bit numbers, for example, the machine is "addition-complete." This is the operation mode used by all "bit-sequential" computers such as Atanasoff's ABC, the EDVAC, or Turing's ACE (von Neumann, 1945).

Now, the multiplication of two $k$-bit numbers can be reduced to $k$ additions of $2k$-bit numbers, where each number in the additions is just a shifted copy of the second multiplication argument or is a zero. For example, we can multiply 111 by 101 as follows:

$$\begin{array}{cccccc} & 1 & 1 & 1 & & \\ & & 0 & 0 & 0 & \\ & & & 1 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 & 1 \end{array}$$

In this example, the second row in the addition is zero because the second bit of the number 101 is zero. We can reduce the second row to zero by computing the conjunction of the second bit in 101 (that is, a zero) with each one of the bits in the multiplier 111. The conjunction of the first and third bit of the multiplicand (a one) with the multiplier reproduces the multiplier 111. It is evident that the whole multiplication can be computed by a deterministic formula and a fixed number of logic instructions.

Now, what about division? In the case of a binary division, we have to show that we can reduce the division of two non-signed 8-bit numbers to addition, subtraction, and multiplication. Assume that we want to divide $n$ by $m$. Assume that the result can be encoded in the eight bits $b_7$ to $b_0$. We then have:

$$n = (2^7 b_7 + 2^6 b_6 + \ldots + 2^0 b_0)m + r$$

where $r$ is the remainder of the integer division $m/n$. This remainder is positive or zero. We can find the bit $b_7$ by testing if $n - 2^7$ is negative or positive. If it is negative, $b_7$ must be 0. If it is positive, then $b_7$ is 1.

The main problem here is that testing a result and continuing with the division algorithm seems to require some type of "IF" instruction. However, this is not so. Such simple decisions can be implemented using logic flags. In the case above, we can compute $n - 2^7$ (the latter number is a constant) and inspect the "sign" bit $S$ of the result (the last carry-over bit). Then we can set $b_7$ to $b_7 = not(S)$. We can then continue recursively.

From these considerations, a very simple algorithm results:

```
for $k = 7..0$
    $r = n - 2^k * m$
    $c = \text{sign\_bit}(r)$
    $b(k) = (1 - c)$
    $n := (1 - c) * r + c * n$
```

If the quotient is zero, the remainder has to be set to $m$.

## 11.8 Discussion

Very early during the development of his different floating-point computers, Konrad Zuse became aware that solving combinatorial and logic problems required more than floating-point registers. Modern computers have an integer ALU and one FP unit. In Zuse's terminology, machines designed for numerical operations were "algebraic" while machines designed for logic and combinatorial problems were "logistic" (another name for logic (Zuse, 1943b)). After the war, Zuse even requested a patent for a machine with one "logic" and one FP processor, as well as a common memory (Zuse, 1950). Well before 1945, he started thinking about reducing the size and requirements for the logic machines, until he developed the ideas explained in the previous sections.

As this chapter has shown, the logic machine can implement the four arithmetical operations. With the processor of the logic machine and a sequence of instructions, it is possible to implement any logic computation requiring a fixed number of steps. If the number of iterations of an arithmetical algorithm is known, loop unrolling can produce a single program executable by the logic machine. If the number of iterations is unknown, a loop can still be implemented. However, the machine will not stop.

It is easy to stop the processor in the Z3 and Z4 computers: dividing zero by zero produces an exception and stops the machines. In the logic machine, there is no way to end the chain of computations in a loop since the processor does not handle exceptions.

We see then that the logic machine was an intellectual exercise for Zuse, just a proof of concept that a minimal machine could implement all of the arithmetic operations and displace the algebraic machines. It was also important from the point of view of Plankalkül, which strived to reduce all software computations to one-bit operations on composite data structures. The only possible application could have been as the "core" of a microinstruction unit for the algebraic machines. Such a core would have had a program tape for each high-level instruction (addition, subtraction, multiplication, division of floating-point numbers, for example) and would have made the algebraic machines cheaper and easier to reprogram.

The logic machine could be compared with Turing machines. Both act on single bits, transforming a memory bank. However, the logic machine addresses memory randomly, and thus, the size of the memory is limited by the word length of the punched tape commands. In the Turing machine, all addressing is done relative to a pointer, the read-write head. There is no need to handle absolute addresses.

The CPU of the logic machine is simpler than the CPU of a Turing machine. The latter includes a table of all state transitions depending on the current state and input. Some authors have examined the minimal number of states and symbols needed for universal computation. It turns out that a Universal Turing Machine can be defined using just two states and a few symbols, or two symbols and a few states (Minsky, 1967). The CPU of Zuse's machine has only two registers and three hardwired operations: negation, conjunction, and disjunction. It seems much simpler than the Turing machine. However, the programs that have to be written are, in the end, as overblown and incomprehensible as the programs written for a Turing machine.

The main issue with Zuse's logic machine is the absence of a test operation (an IF operation). This instruction can be simulated using arithmetic, but what cannot be simulated is an IF followed by a stop in one of the IF alternative threads. Therefore, the logic machine cannot implement open-ended loops (WHILE loops) that stop.

There are some alternatives. Zuse could have used a program consisting of a single loop to simulate conditional branching (as was explained for the Z3) and do the rest with arithmetic. Of course, this is only feasible in theory, due to the exponential blowup of the necessary program.

We can also imagine a processor with its clocked circuits, executing cyclically. The logic machine could compute every single bit computed by such a processor in one cycle (a finite amount). The program code to be executed would be stored in the main memory and the external program in the punched tape would be fixed. It would be the "universal logistic machine" that proceeds by executing one cycle of the simulated microprocessor in every iteration.

The main problem, though, is that the load and store operations use fixed addresses. There is no indexed addressing so that accessing an address given by a pointer is not possible. This would be necessary for simulating the conditional branch instruction in programs. There are theoretical ways for going around this problem, but the necessary code blows up, so I will not explain this further. Turing got it right when he decided to use relative addressing of the symbols in memory for his machine, thus avoiding the problem of the memory size, which in the Turing machine can be an unbounded tape.

There are several ways of achieving universality with different types of hardware (see Rojas (1998b)), but the logistic machine is so simple that, without indexed addressing, it is unfeasible to do it. The logistic computer is thus limited to forward computations without conditional branch.

Obviously, the logistic machine will not be used any time soon but it is a fascinating piece of computational theory, which requires a minimal hardware implementation. It is a magnificent example that only shows the depth of Zuse's thoughts about the theory of computation as he concentrated on finishing his most important computer in those years, the Z4.

## References

Minsky. M.L. 1967. *Computation: finite and infinite machines*. Upper Saddle River, NJ: Prentice Hall. https://doi.org/10.5555/1095587.

Rojas, R. 1997. Konrad Zuse's legacy: The architecture of the Z1 and Z3. *IEEE Annals of the History of Computing* 19(2): 5–16. https://doi.org/10.1109/85.586067.

Rojas, R. 1998b. How to make Konrad Zuse's Z3 a universal computer. *IEEE Annals of the History of Computing* 20(3): 51–54. https://doi.org/10.1109/85.707574.

von Neumann, J. 1945. First Draft of a Report on the EDVAC. Unpublished draft.

Zuse, K. 1943b. Rechenplangesteuerte Rechengeräte für technische und wissenschaftliche Rechnungen. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0217/

Zuse, K. 1944d. Patentanmeldung Z-394: Rechenvorrichtung. Available online at the Zuse Internet Archive.

Zuse, K. 1950. Patentschrift Nr.926449, Kombinierte numerische und nichtnumerische Rechenmaschine. Deutsches Patentamt, 11 Seiten.

Zuse, K. 1952a. *Bedienungsanweisung für Zuse Z4*. Zurich: ETH Zurich. https://doi.org/10.7891/e-manuscripta-98601.

Zuse, K. 1972. *Der Plankalkül*, vol. 63. Sankt Augustin: Berichte der Gesellschaft für Mathematik und Datenverarbeitung.

# Chapter 12
# The First Code for Computer Chess

*This chapter describes the first computer chess code ever written. A library of computer chess functions was compiled around 1945 by Konrad Zuse, after he had fled from Berlin to the Bavarian Alps during World War II. Zuse wrote the code to illustrate the expressiveness of his "Plankalkül" (calculus of programs), the high-level programming language he had been developing intermittently between 1942 and 1945. Zuse's chess code was originally published in German in 1972. However, due to the lack of a compiler for Plankalkül at the time, the code remained unimplemented and untested until 1999. In that year, we developed a compiler for Plankalkül and also rewrote Zuse's chess functions in Java. As part of this effort, we introduced a move generator and a user-friendly visual interface, allowing users to easily manipulate chess pieces on a virtual chessboard and observe the computer's moves. Here we describe Zuse's code, the state of computer chess by 1950, and compare Zuse's approach to Alan Turing's chessboard evaluation strategy.*

## 12.1 Computer Chess

Many printed and online publications commonly refer to Claude Shannon, the renowned American mathematician, as the "father" of computer chess. However, as we argue in what follows, this is not so—other scientists had dreamed of, or designed specific mechanizations of different aspects of the game before Shannon's contributions became well known. Certainly, Shannon was the most influential person in computer chess at the turn of the 1950s having published two widely read articles describing how to program a computer to play chess (Shannon, 1950). Nathan Ensmenger, who has written a very comprehensive social history of computer chess, puts emphasis on the fact that "it was the mathematician Claude Shannon who wrote the very first article ever published on the art of programming a computer to play chess" (Ensmenger, 2012).

However, there were at least two predecessors who either very carefully described how to program a computer to play chess or even wrote substantial code for a chess-playing program. The first was Alan Turing, the British mathematician, who designed and executed a chess-playing program—by hand—and reported on his work in 1953. The second was the German inventor Konrad Zuse, who deserves credit for having written the world's first chess code, although the library of functions remained incomplete. Zuse's chess code was finished in 1945, at the time of his forced "exile" in the Alps. However, his notebooks contain sketches of functions and subroutines for computer chess written years earlier, between 1941 and 1945. Unfortunately, the code, consisting of more than 100 functions and comprising 50 printed pages, was not published until 1972, when Zuse's book about the "Plankalkül", the high-level programming language he developed during World War II, was printed in Germany (Zuse, 1972). The publication in German went largely unnoticed outside the country, and to this day many books and online archives about the history of computer chess have failed to register this achievement.

## 12.2 Prehistory of Computer Chess

Charles Babbage was apparently the first scientist to write about the possible mechanization of chess. It is true that Wolfgang von Kempelen's chess-playing android had already been exhibited in Europe in the 18th century, but we now know that an operator was hidden inside the so-called Mechanical Turk. None other than Edgar Allan Poe published a short commentary trying to "reverse engineer" the deception (Poe, 1836). Napoleon was fooled by the trick and was rather annoyed at losing to a machine (as was Gary Kasparov many years later after losing to IBM's Deep Blue).

Babbage, who knew of Kempelen's automaton, clearly saw that the regularity and rule-driven nature of chess made it a good candidate for mechanization. He certainly underestimated the challenge, but he wrote in "Passages of the Life of a Philosopher" (1864) that while designing his Analytical Engine (which, if completed, would have been the world's first computer) he realized that "games of purely intellectual skill" could be mechanized. He then proposed what would be the first algorithmic description of the steps to be followed by a chess-playing automaton: (1) evaluate the consistency of the board, (2) if the chess pieces have consistent positions, has the machine lost? (3) if not, did it win? (4) if not, can it win in one move? Make that move. (5) Can the opponent win in the next move? (6) if so, prevent that move, (7) "If his adversary cannot win the game at his next move, Automaton must examine whether he can make such a move that, if he were allowed to have two moves in succession, he could at the second move have two different ways of winning the game" (Babbage, 1864). Babbage then goes on to underestimate the number of possible games when he thinks that his machine would be able to consider all of them.

The first real chess automaton was designed by the Spanish scientist Leonardo Torres y Quevedo in Spain around 1910. He built a machine capable of playing an endgame with a white king and rook against a black king (a KRK game). The game started with both sides sufficiently separated, so it was just a matter of maneuvering until the black king could be captured by the white pieces. Only five different actions and eight conditional tests were required. The pieces were moved by mechanical arms (later electromagnets), and the human opponent's move was sensed by electrical contacts on each square of the board (Randell, 1982a). Torres y Quevedo's chess automaton was demonstrated in 1912 and later at many other scientific events in Europe, until 1950.

The first mathematician to consider chess and games as ordered sequences of moves (in a decision tree) was Ernst Zermelo, who gave an address to the International Congress of Mathematicians in Cambridge in 1912, stating that in chess, white or black can force a win for either side, or both can force a draw, in less than $N$ moves, where $N$ is the total number of positions available on the chessboard. It has been said that this "Zermelo's Theorem" is the first real mathematical analysis of strategies in games (Zermelo, 2010). After the Second World War, computer chess was obviously "in the air," as can be read in a seminal book of that time, Norbert Wiener's *Cybernetics*. In this book, Wiener mentions chess as a type of problem solvable by the general techniques investigated by John von Neumann and by performing search in a decision tree.

## 12.3 Enter Turing

Alan Turing worked during the Second World War in the cryptanalytic unit bunkered at Bletchley Park near London. One of the topics of discussion among the mathematicians working there was the possible mechanization of chess (Copeland, 2004). Probably some of them had read Babbage or heard about Torres y Quevedo's machine. Turing was also aware of John von Neumann's writings, having worked with him in Princeton, and would have known of the publication of von Neumann and Morgenstern's book on game theory (von Neumann and Morgenstern, 1944). In this book, the authors discuss games with "perfect information" (such as chess) and possible strategies for rational players. However, they do not provide an actual "algorithm" for playing the game—that would be the task of Turing and others.

Several of the Bletchley Park staff were accomplished chess players, including James Macrae Aitken, the Scottish champion, and the 1938 English champion, Hugh Alexander, who was also Turing's successor as head of "Hut 8" at Bletchley Park. Turing worked there until 1942 and then alternated between the USA and England. He did consulting work for Bell Laboratories and met Claude Shannon in early 1943, when they discussed their common interest in cryptography and computing machines. Hodges mentions that Turing's colleagues remembered that he had been experimenting with the minimax and best-first heuristics for playing chess at Bletchley Park since 1941 (Hodges, 2006).

Thus, the stage was set for the development of a chess-playing "algorithm," i.e., a set of rules and evaluation heuristics, which Turing perfected in collaboration with a former student named D.G. Champernowne. The set of rules, apparently completed and written down around 1948, while Turing was working at the National Physical Laboratory in Manchester, was christened "Turochamp".[1] In 1952, Turing "ran" the program on paper and lost a chess game to a colleague. Each move is said to have taken 30 minutes to compute on paper. Turing describes his chess playing heuristics in a paper written around 1953 (Turing, 1953).

Turing's code first looks at all possible moves by white (all plies, a ply being a move by one of the two opponents) and all possible responses by black. Then it determines whether after two plies there are "considerable" moves or not. These are white moves in which a piece can be captured, especially by a piece of lesser value, or a check can be made. If black can recapture, that is also a possible move. A series of captures–recaptures would then follow until the last capture. In principle, if no captures are possible, only two moves would be considered and evaluated. Turing's evaluation function has a static part, considering only the values of the pieces and a dynamic part, considering their positions on the board and their mobility. Turing proposed to value each piece as follows: pawn 1, knight 3, bishop 3.5, rook 5, queen 10, and checkmate is valued with $\pm 1000$. More points are added according to the square of the number of moves available to each piece, additional points are given for protected pieces, and so on. Even pawns increase in value for each row they have already moved. Turing's evaluation function is not easy to compute by hand, since it contains so many positional additional elements: 30 minutes of hand calculation for each board evaluation seems to be a good approximation.

## 12.4 Zuse's Chess-Playing Program

Konrad Zuse wrote the framework for a chess program between 1941 and 1945. The final manuscript was completed after the war. Unable to resume work on his computing machines, he decided to finish a theoretical treatise he had been working on intermittently, detailing his ideas about general computation. This manuscript became the *Plankalkül* (calculus of programs), which was published in the 1970s with some additional notes and an introduction (Zuse, 1942c).

In the introduction to the Plankalkül, Zuse wrote that he had only learned to play chess in 1937, because he thought that this game could be automated with the help of calculating machines. Zuse's notebooks (of which several hundred pages survived the war) contain handwritten notes which confirm that he was thinking

[1] In a letter from Turing to J.. Good in September 1948, Turing writes that he has not yet written down the Turochamp rules, but intends to do so in order to play the Shaun-Michie "chess machine" (Turing, 1948).

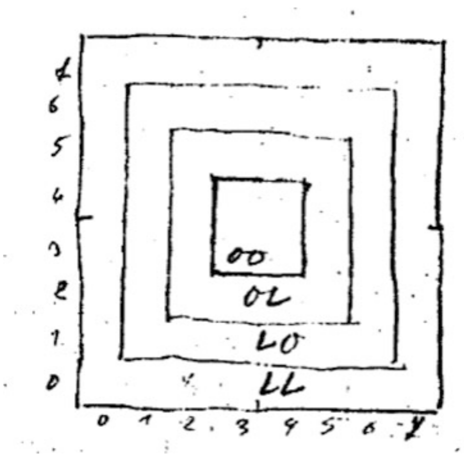


**Fig. 12.1** Zones in the 8 by 8 squares chess board (from Zuse's notebooks, Zuse, 1941-1942)

about his Plankalkül and computerized chess during the war, at the same time as he was building the Z4 and the special machines S1 and S2 (see Fig. 12.1).

Plankalkül was the first draft of a high-level programming language ever written (because of its connection to chess, W. Bibel has called Plankalkül "the first AI language" (Bibel, 2014)). Consider the early computers: Babbage's machines were programmed in "assembly language," since the elementary processor instructions were simply listed one by one. The ENIAC had to be hardwired to perform any useful computation. The Harvard Mark I was also programmed in assembly language. In fact, the first widely used programming languages did not appear until the 1950s, especially when FORTRAN and COBOL took off.

In Plankalkül, variables are multidimensional arrays of bits. For example, a 32-bit number would be represented as an array of 32 bits. A vector of ten 32-bit numbers would be a double-indexed array of 10 times 32 bits. Indices allow the programmer to access any component or subarray in an array. The elementary imperative constructs of the language are the assignment of values and the evaluation of complex arithmetic expressions. The language also includes an IF and a WHILE statement, as well as function definition and function application. The only curious aspect of the language syntax, from a modern perspective, is that Zuse wrote variables as four-line structures. For example, the four lines

$$\begin{array}{c} V \\ 0 \\ 1 \\ 32.0 \end{array}$$

represent the variable V0, component 1, of type 32 bits. The addition of two such variables V0 and V1 could be assigned to a result-variable V2 as follows:

$$\begin{array}{ccccc} \mathrm{V} & = & \mathrm{V} & + & \mathrm{V} \\ 2 & & 0 & & 1 \\ 1 & & 1 & & 1 \\ 32.0 & & 32.0 & & 32.0 \end{array}$$

Apart from this strange four-line notation, Plankalkül resembles a modern generic imperative language with adequate programming constructs and the necessary flexibility to write any desired program. Experienced programmers can learn to code in Plankalkül in less than an hour. Zuse's computer chess framework constitutes the entire Chap. 5 in the Plankalkül manuscript. The main data structures that Zuse designed for chess are:

(a) A representation of the current state of the board. This was done using an array of 64 four-bit numbers. Four bits were needed to encode the binary ID of each of the 12 possible pieces (six white and six black types) as well as the "cell-free" value:

$$Board = (x1, x2, \ldots, x64),$$

where $xi$ is the binary code of the chess piece or a free cell. The four-bit binary codes were chosen so that the lowest bit represents the color (white or black) of the piece.

(b) A representation of a position on the board. This was done using two integers $x$ and $y$, which take values from 0 to 7:

$$Position = (x, y)$$

(c) A move representation, which is just an array of two positions on the board: the start and end position, and a bit to describe the color of the piece being moved.

$$Move = (V0, V1, s)$$

where V0 and V1 represent board positions, and s is 0 or 1.

(d) A table of values for each chess piece: the value for a pawn is 1, for the bishop and knight it is 2, for the rook it is 3, 4 for the queen, and 5 for the the king. The king's value is superfluous since it is not used in any other chess function.

Zuse then wrote functions numbered from 1 to 203 (there are some gaps in the numbering, so only about 120 functions were written). For example, one function takes a board and a move as arguments and checks the validity of the move according to the rules of chess. Another function takes two positions as arguments and checks if they lie on a diagonal or a line on the board. The complete set of functions is extensive because it is used to ensure that the rules of the game are followed.

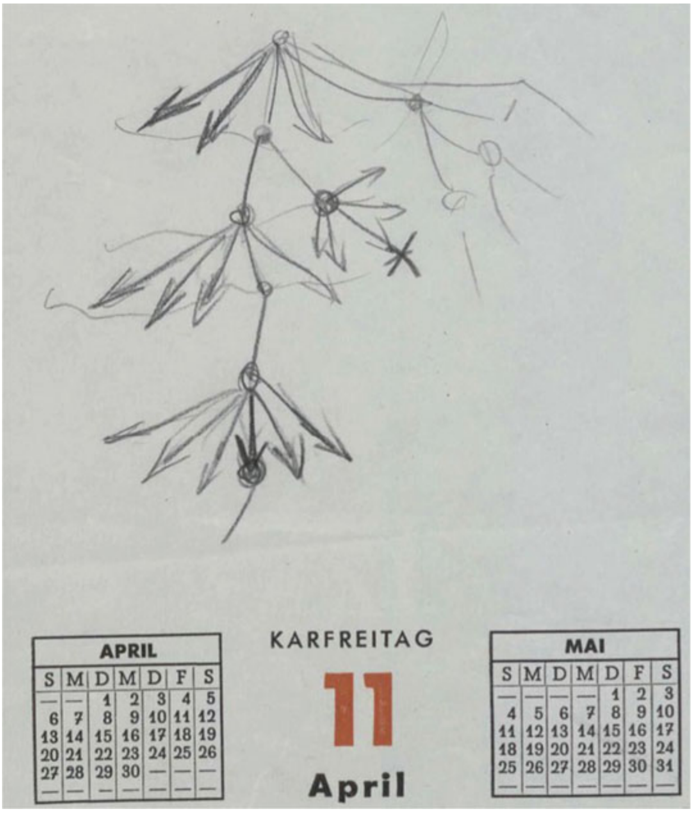
However, Zuse overprogrammed. Some of the functions are superfluous for a real game. One of them, for example, checks whether a given chess board is possible, i.e., whether the total number of pieces of different types could have been produced during a chess game. Since computer chess starts with a legal board, and all moves by the human player or the computer are checked immediately, i.e., when they are entered or produced, no illegal board can be generated during a game, so checking the legality of the number of pieces with a special function is a nice programming exercise, but not necessary.

Despite this overprogramming, it can be said that the auxiliary functions contained in Chap. 5 of the Plankalkül are sufficient to implement a complete game of chess, with one notable exception mentioned below. For example, there are functions that test the king's check, en passant captures, and the possibility of castling. There is a function that, given a chessboard and a legal move, generates the next board. How did Zuse compare alternative moves? An evaluation function is needed, and here Zuse did not go much beyond counting pieces by their value. Zuse's evaluation function for a move is therefore trivial: each piece, except the king, on the white side is multiplied by its value and the results are added. The same is done for the black pieces. Zuse's evaluation function is similar to Turing's initial evaluation of the board, but only the first step, before positional advantages are considered and added to the final result. Zuse's evaluation function is certainly not very sophisticated. He was never a good chess player, and his evaluation function reflects this fact.

## 12.5 Move Generation in Zuse's Notebooks

The most important omission in Chap. 5 of Plankalkül is the move generation function itself! In the introduction to the Plankalkül manuscript, Zuse wrote that he had thought of some kind of minimax evaluation for the chess program, but he never got around to writing down the code. Although the chess code in the Plankalkül is extensive and contains dozens of auxiliary functions, the one really important function (the move generator) is missing from the list of subroutines. Written in parentheses (and this is actually the very last line of the 1945 text), we find the words: "complete move generator." Had Zuse considered programming a decision tree or not? It is hard to imagine that he had not. Even Babbage, in his reflections on automated chess, wrote that the sequence of moves and countermoves during a game must be considered by the machine. Fortunately, we now have access to parts of Zuse's notebooks and on some pages dated 1941/1942 we do indeed find evaluation trees for chess and sketches of their possible development. Figure 12.2 shows one of these sketches (from 1942) and Fig. 12.3 a decision tree.

As these sketches show, Zuse was well aware of the need to apply the evaluation function at each level of the board, taking into account the opponent's countermoves.



APRIL

| S | M | D | M | D | F | S |
|---|---|---|---|---|---|---|
| — | — | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| 27 | 28 | 29 | 30 | — | — | — |
| — | — | — | — | — | — | — |

MAI

| S | M | D | M | D | F | S |
|---|---|---|---|---|---|---|
| — | — | — | — | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| — | — | — | — | — | — | — |

**Fig. 12.2** A generic decision tree for chess. Note the calendar for 1941 (Zuse, 1941-1942)

Writing down the code was certainly difficult: it would have been the most difficult coding example in the Plankalkül book. The necessary work was left for later, a time that never came.

Claude Shannon did not have a chess program in 1949/50 when he published his first papers on computer chess, but he had a good description of the necessary steps. Shannon distinguished between a type A strategy, in which all combinations of moves (one's own and the opponent's) are explored to a maximum depth, and a type B strategy, in which only promising branches of the tree of possibilities are expanded. Early computer chess programs used a Type A strategy, and Zuse's notebooks seem to indicate that this is what he had in mind for his chess program. Turing followed a mixture of a type A and a type B strategy, because the fixed depth level (two plies) could be extended in case a piece could be captured or revenged.

All early computer chess programs used brute force, i.e., an extensive search in the move tree guided by a heuristic evaluation function. The method described by Shannon in 1950 is now known as the "minimax algorithm." It consists of selecting the best possible move under the assumption that the opponent will also respond

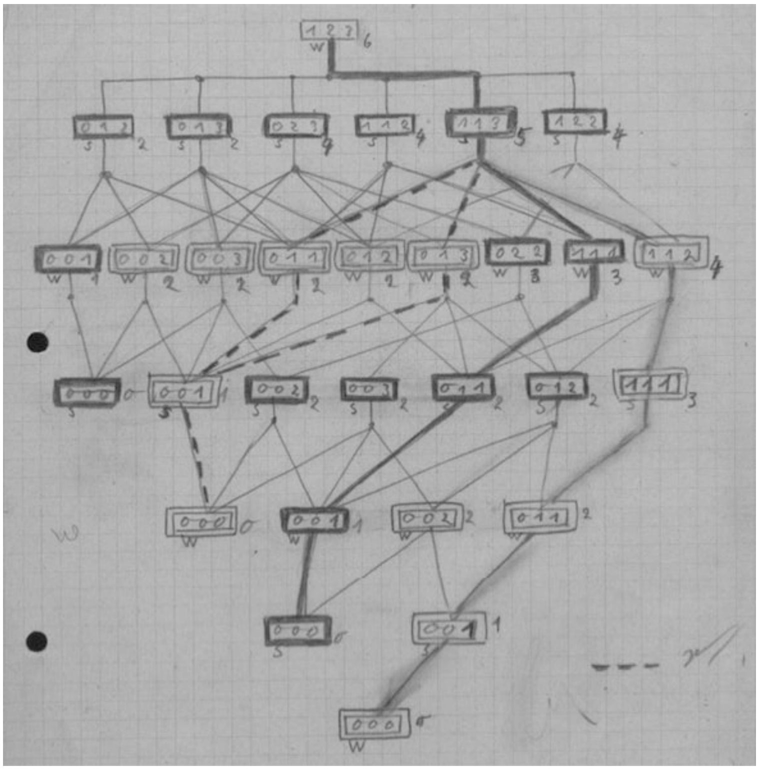


**Fig. 12.3** A decision tree for moves of white (W) and black (S) pieces. The meaning of the numbers is unclear. Here the white pieces move first and reach a final state while the game proceeds along the decision tree (Zuse, 1941-1942)

with his best move. The player moving a piece tries to maximize his gain according to the evaluation function and the opponent tries to minimize it. The interplay of these two intentions gives rise to the "minimax" strategy described by John von Neumann in his study of game theory.

## 12.6 Computer Chess After Zuse, Turing, and Shannon

The first full chess program to run on a computer was written by Alex Bernstein and others for an IBM 704 in 1957. It examined all future four-ply games (a move by black or white is called a ply) in about 8 minutes and could win against amateurs. A smaller program playing on a 6 by 6 board had been implemented a year earlier on the MANIAC I, a computer built at Los Alamos National Laboratories. But the first chess program that played in a real tournament was the legendary MacHack VI, written by Richard Greenblatt for a PDP-6 at MIT. Later it was available on all PDP-

10 computers, a machine very popular at universities and research centers. It was the first chess program I ever played against.

Although Herbert Simon and Allen Newell predicted in 1956 that a computer program would become world champion within 10 years, several decades passed before a computer could win a match against the world champion. The reason seems to be that good human players can recognize patterns on the chess board that leads them to consider only a handful of possible moves. While computer chess programs can play decently because they examine thousands or millions of moves per second, a human player applies a higher-level strategy. Good moves "pop up" in the mind of a grandmaster, just as we recognize a human face in a crowd. The process of recognition itself is not open to introspection. We do it, but we don't know why.

In 1989, Deep Thought, a parallel chess machine built by a team led by Feng-Hsiung Hsu, became the world champion in computer chess (Hsu et al., 1990). Deep Thought achieved this result by using a large library of opening games and raw computing power. Although the machine could examine 2 million moves per second, Gary Kasparov easily defeated the program in the same year. That would change the moment IBM hired the developers of Deep Thought with the medium-term plan of overpowering Kasparov. The IBM machine, Deep Blue, a parallel computer with 32 processors and 256 special chips, defeated Kasparov for the first time under tournament conditions in 1996, but lost the series 4-2. Just 1 year later, Kasparov lost the rematch 3.5 to 2.5 in a highly publicized event broadcast over the Internet. With the world champion defeated, computer chess became something of a solved problem, and the artificial intelligence community moved on to more challenging areas. But Shannon, Turing, and Zuse would surely have been proud of such an achievement.

## References

Babbage, C. 1864. *Passages from the life of a philosopher*. London: Longman, Green, Longman, Roberts & Green. https://doi.org/10.1017/CBO9781139103671.

Bibel, W. 2014. Artificial intelligence in a historical perspective. *AI Communications* 27(1): 87–102. https://doi.org/10.3233/AIC-130576.

Copeland, B.J. 2004. *The essential turing: The ideas that gave birth to the computer age*. Oxford: Oxford University Press. https://doi.org/10.1093/oso/9780198250791.001.0001.

Ensmenger, N. 2012. Is chess the drosophila of artificial intelligence? A social history of an algorithm. *Social Studies of Science* 42(1): 5–30. https://doi.org/10.1177/0306312711424596

Hodges, A. 2006. *Alan turing: The enigma*. New York: Walker & Company

Hsu, F.H., Anantharaman, T., Campbell, M., and A. Nowatzyk. 1990. A Grandmaster chess machine. *Scientific American* 263(4): 44–51.

von Neumann, J., and O. Morgenstern. 1944. *Theory of games and economic behavior*. Princeton: Princeton University Press.

Poe, E.A. 1836. Maelzels chess player. *Southern Literary Messenger* 2: 318–326

Randell, B. 1982a. From analytical engine to electronic digital computer: The contributions of ludgate, torres and bush. *Annals of the History of Computing* 4(4). https://doi.org/10.1109/MAHC.1982.10042.

Shannon, C.E. 1950. Programming a computer for playing chess. *Philosophical Magazine* 41: 256–275. https://doi.org/10.1007/978-1-4757-1968-0-1.

Turing, A. 1948. Document amt/k/1/77. Alan Turing Digital Archive.

Turing, A. 1953. Digital computers applied to games. In *Faster Than Thought*, ed. Bowden BV. London: Sir Isaac Pitman & Sons Ltd.

Zermelo, E. 2010. *Ernst Zermelo – collected works*, 1st edn. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-540-79384-7.

Zuse, K. 1941-1942. Stenografische Notizen zum Schachspiel. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0746/.

Zuse, K. 1942c. Vorarbeiten zum Plankalkül. Schachprogramme. Zuse Papers 012/012.

Zuse, K. 1972. *Der Plankalkül*, vol. 63. Sankt Augustin: Berichte der Gesellschaft für Mathematik und Datenverarbeitung.

# Chapter 13
# The Reconstruction of Konrad Zuse's Z3

*This chapter describes our reconstruction of Konrad Zuse's Z3, the first programmable computing machine in the world. The original Z3 was built in Berlin between 1938 and 1941, it was destroyed in a bombing raid during World War II. Our reconstruction project started in 1994 with a detailed study of the original circuits and software simulations. Between 1999 and 2001, a functional replica was built and unveiled at a conference commemorating the 60th anniversary of the public presentation of the original machine. The hardware reconstruction of the Z3 is now part of the collection of the Konrad Zuse Museum in Hünfeld, Germany. As part of our project, we also wrote Java simulations of components, of the whole machine, and a 3D functional simulation of the Z3 and its user console. All of them were available in the Konrad Zuse Internet Archive until browsers discontinued Java applets.*[1]

## 13.1 Introduction

Between 1999 and 2001, we built a replica of the computing machine Z3, designed and constructed by the German inventor Konrad Zuse six decades before. Like the original Z3, the replica was built in Berlin. It was the central piece at a conference held in May 2001 in the German capital to commemorate "Sixty Years of Computation." It was a prolonged undertaking—the preparatory work for making the reconstruction possible started almost 5 years earlier. During that time, we studied the available documentation, conducted interviews with Konrad Zuse himself (who died in 1995), wrote some simulations of parts of the machine, and built a hardware replica of the addition unit of the Z3. Therefore, the final product, the reconstructed Z3, is just like the tip of the iceberg: there is much more to building

[1] Chapter based on Rojas et al. (2005).

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_13

a reproduction of a historical computer than just soldering old components. This chapter is our account of our journey into history, about how the reconstruction was made, and why it is important for historians of computing and for computer scientists to pursue such projects.

The story of the original Z3 has been told many times. Here we provide only a minimum of background. For a more detailed historical account, see Konrad Zuse's autobiography (Zuse, 1970), or Petzold's history of early computing in Germany (Petzold, 1992).

As early as 1935–1936, the student Zuse started thinking about programmable mechanical calculators specially designed for engineers. His vision at the time, and in the ensuing years, was the desktop calculating machine, not the large and bulky supercomputer. As this vision slowly matured in his mind's eye, Zuse took a series of important and unconventional decisions.

His first peculiar decision was to build a fully binary machine. As a student, Zuse had experimented with the binary system (on paper) and had rediscovered Boolean logic. He realized that all computations could be reduced to binary logical operations and saw in this fact an economy of means that could match his own limited financial resources. Remember that most calculators built at the time used an internal decimal representation.

Zuse's second unconventional decision was to build a binary machine using mechanical components. As a teenager, Zuse was a masterly machine builder and won some prizes for his "building blocks" creations (a German version of Meccano (Rojas, 2001)). He was not very familiar with electrical circuits and thought that it would be easier to build a mechanical machine. He was also worried about the reliability, cost, and size of a machine built with telephone relays. On the one hand, a back-of-the-envelope calculation clearly showed that a relay machine would be too heavy and expensive for its intended users. On the other hand, he had infinite confidence in his mechanical prowess and in the possibility of miniaturizing mechanical components in the future. As late as 1950, Zuse still believed that mechanical memory components could be used in computers.

Zuse's third unconventional decision was to resign from his job at a German company in 1936, in order to build his machine and start a company of his own. The *Zuse Ingenieurbüro und Apparatebau* start-up was, on paper, the first computer company in the world. However, Zuse's first real product would be the Z4, finished in 1945 just weeks before the end of World War II, and delivered to the Eidgenossische Hochschule (ETH) in Zurich as late as 1950! Untypical for Germany, Konrad Zuse was perhaps the first computer entrepreneur in the world. Karl Marx once wrote that in Germany everybody dreams of becoming a "civil servant"—not Konrad Zuse, who was one of the few who foresaw an important market for programmable personal calculating machines.

In what follows, we assume that the reader is familiar with the basics of the architecture of the Z3, showcased by Zuse in 1941.

## 13.2 Architecture of the Z3

Figure 5.2 (on page 91) shows a block diagram of the architecture of Zuse's Z3. The main components are the 64-word memory, the separate floating-point processor with two registers, the punched tape reading unit, the control unit, which decodes an instruction read from the punched tape and orchestrates all data transfers inside the machine, and the I/O devices. In the Z3, numbers were entered through a decimal keyboard similar to those used in cash registers. The decimal exponent of a number could be selected by pressing a button. The output was shown by switching on lamps located behind an array of decimal digits. The exponent of the result was also shown by switching on a light behind a line of exponents numbered from −8 to +8.

As can be seen from the block diagram, the Z3 possessed a kind of "von Neumann" architecture. The only exception would be the program being held externally, but there was no way of affording the memory necessary to store internally the user code, although Zuse thought about this possibility. Externally held programs (or hardwired programs) were used in all early computers (Rojas and Hashagen, 2001).

Figure 5.4 (page 96) shows the internal structure of the processor. Two addition units are used, one for the floating-point exponents and one for the mantissas. In the Z3, an argument for an arithmetical operation was stored in the register (Af,Bf), where Af refers to the exponent, and Bf to the significand (mantissa) of the number. The second argument was stored in the register (Ab,Bb). The register (Aa,Ba) was a temporary register invisible to the programmer. The results of operations were stored back in the register (Af,Bf). The addition units in the processor were integer units that could operate on two's complement binary numbers. That is, they could only add numbers or their two's complement. An instruction cycle of the Z3 was as long as one addition step. All other operations were implemented using combinations of additions and subtractions, which could operate on shifted arguments. Shifting allows us to multiply and divide numbers faster. In this case, "fast" means 3 seconds for a multiplication.

The original Z3 was destroyed during the war. However, in 1960, Konrad Zuse built a reconstruction for Deutsches Museum in Munich, assisted by engineers from his computer company. The machine was built to strengthen Zuse's patent application from 1941, which was still under review and which had been hotly contested by computer companies, especially IBM. Eventually, the patent application was rejected, as happened in the USA with the ENIAC patent application. The reconstructed Z3 was later donated to Deutsches Museum in Munich.

The problem with the first reconstruction of the Z3 was that no additional documentation was written. It can be safely said that only Konrad Zuse and two of his assistant engineers knew how the machine worked. After the Zuse KG disappeared in the late 1960s, only Konrad Zuse remained a witness of the Z3 technological history. Moreover, all published documents, even Konrad Zuse's memoirs, are very sketchy regarding the machine. One of the diagrams printed in the autobiography, for example, is of the mechanical data flow in the Z1 but is labeled as a diagram of the Z3. Therefore, Germany now had a reconstruction of the Z3, but only few individuals understood how it once worked.

The Z3 went unnoticed in other countries for many years. The main publications describing aspects of the Z3 were written in German and only a few were translated. Most historians of computing in the USA and United Kingdom did not take notice of the Z3 until new studies and appraisals started appearing in the 1970s and 1980s (Ceruzzi, 1983).

## 13.3 The Konrad Zuse Internet Archive

In retrospect, we can date back our Z3 reconstruction project to 1994. At the time I published a paper asking the old question of who had invented the computer (Rojas, 1994). By coincidence, in that same year, Konrad Zuse visited Freie Universität Berlin, and I was able to discuss the matter with him. One of the frustrating aspects of the debate in Germany about the invention of the computer was, at the time, the lack of detailed information about the exact structure and capabilities of Zuse's machines. Konrad Zuse was very mellow, even when I questioned if the Z3 could be called a computer. His supreme confidence put him above all such disputes. He always thought of himself as the one and only inventor of the computer. But he was intrigued by my questions and promised to send the Z3 patent application of 1941, a document few people had seen and still fewer had fully understood.

The patent application for the Z3 arrived a few weeks later. The document is difficult to read. The whole design of the machine is explained using relays and not current digital design conventions. There is no good overview of the machine. The reader has to go through individual circuits, one by one, trying to connect them into an organized whole. The patent application itself also contains numerous misprints that make it even more difficult to understand the machine. Zuse kept some details of the construction for himself because, as he said during one of our interviews, he did not want anybody at the patent office to get "too clever."

Understanding the patent application of the Z3 was the first obstacle faced by our project, but it could be mastered using modern circuit simulators. Several students taking a seminar on the history of computing at FU Berlin wrote code for simulating parts of the machine. In 1995, we had a functional emulation for Unix, which could recreate the way the machine was used (the "look and feel"), but without simulating the individual circuits. Later, at the University of Halle, Alexander Thurm wrote his undergraduate thesis working under my direction. This was the breakthrough we needed. The circuits in the patent application were deciphered one by one by me, and Alexander Thurm implemented the inner workings of the Z3 using the new programming language Java, just released a few months earlier. Our idea was to write a simulation that could run on any kind of computer. The project was successful: the complete Java simulation of the Z3 was operational and available on the Internet in early 1996. Many programmers from all over the world took notice and the website devoted to the simulation registered thousands of hits over several

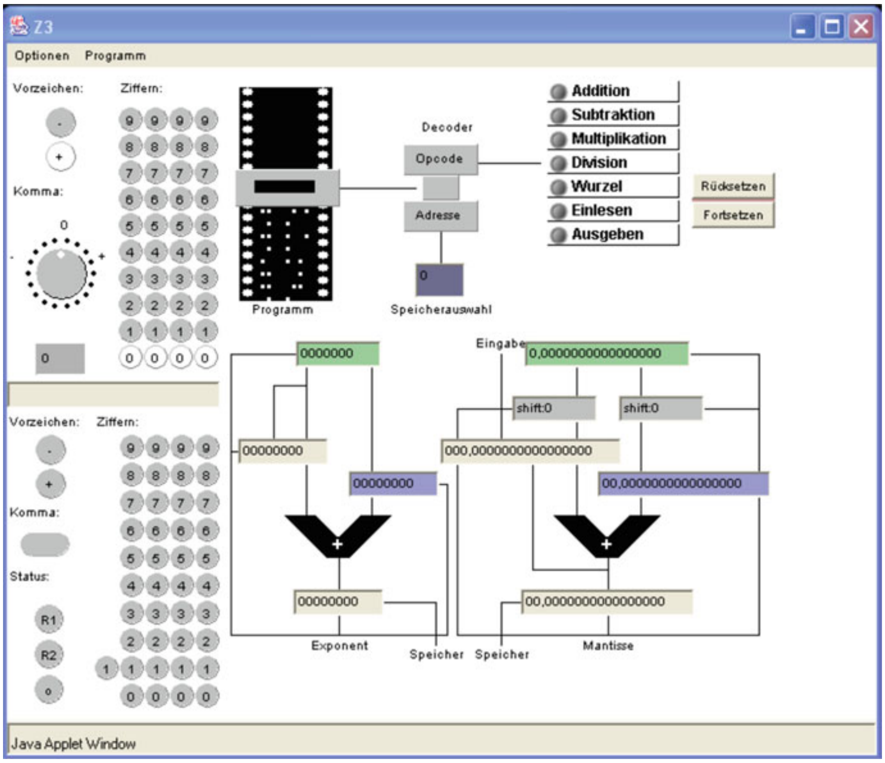


**Fig. 13.1** A screenshot of the Java simulation of the Z3. The left side shows the input console (upper left) and the output display (lower left). The tape reader is visible, as well as the ALUs for exponent and mantissa

months. A description of the simulation was published later in Dr. Dobb's Journal, a magazine for hardcore programmers usually not very interested in the history of computing (Rojas, 2000).

Figure 13.1 shows a screenshot of the Java simulation. The user could start the simulation using an Internet browser, enter numbers by clicking on the decimal console, and start a program contained in a virtual tape or using the Z3 as a calculator. The speed of the simulation matched the speed of the original machine.

The next step after having gotten the simulation to work, was to publish a clear and commented description of the circuits. The first complete diagram of the machine appeared in German in 1996 (Rojas, 1996b) and later on in English in 1997 (Rojas, 1997). The paper has been translated into Spanish, Polish, and Russian (see Chap. 5).

Konrad Zuse's patent application of 1941 was first published in 1998 in a book that received much attention in interested circles in Germany (Rojas, 1998a). The patent application was made available for the first time, together with long footnotes and an accompanying description at several levels of detail. All circuits were redrawn, and the book included the first really simple yet complete block diagram of the machine, as well as the sequence of microinstructions associated with each operation. As we now know, the Z3 contained micro-steppers that could be rewired to implement complex operations. It was a kind of microcode before microcode.

Soon after the success of publishing the patent application, we decided to start a more ambitious cataloging project of other documents. From 1999 to 2002, we scanned and published on the Internet the main documents and notebooks produced by Konrad Zuse from 1935 until his death. The notebooks are hard to read. They usually include just diagrams and stenographic comments (we had to find people able to read stenography). The material was scanned using the photocopies held by the Heinz Nixdorf MuseumsForum in Paderborn. This paper archive was set up by the Gesellschaft für Mathematik und Datenverarbeitung when Konrad Zuse was still alive. The originals for the photocopies are now housed at Deutsches Museum in Munich.

We brought the most important part of the Paderborn archive online. The Konrad Zuse Internet Archive (at http://zuse.zib.de/) registers almost 100,000 visitors each year. Many gigabytes of documents have been downloaded from the website. Although we stopped scanning documents in 2002, the collection has grown, because we also include theses, comments, and documents written by other people about the work and life of Konrad Zuse. We have also published several circuit simulations, mentioned in what follows.

## 13.4 Reconstruction of the Addition Unit

The second step toward a reconstruction of the Z3, after having understood and simulated the circuits, was to build a subset of the addition unit using relays. We decided to use smaller relays than those used by Zuse because we wanted to mount the final product on a frame. The connections were made using a printed board instead of loose cabling. This saved hours of soldering and manual work. Other than that, the addition unit was a one-to-one reconstruction of the original addition circuit of the Z3 and was mounted on a panel. We included two registers in the panel, one for each integer argument (limited to 12 bits). Interested users can press buttons to set the two binary arguments. The unit then computes the sum or difference, according to the settings of a radio button. Each relay shows its on or off state with a small red LED.

The addition unit works in three steps: first, the binary arguments are added, without carry, then the carries for all positions are computed (using carry-look-ahead), and in the third and final step, carries and partial sum are combined into the final result. A switch allows the user to activate each step one after the other, in order to be able to follow the computation. Figure 13.2 shows a picture of the reconstructed addition unit. The circuits were drawn by Frank Darius, the components were selected by Georg Heyne, and Cüneyt Göktekin wrote the corresponding Java simulations. As can be seen in the picture, the panel contains

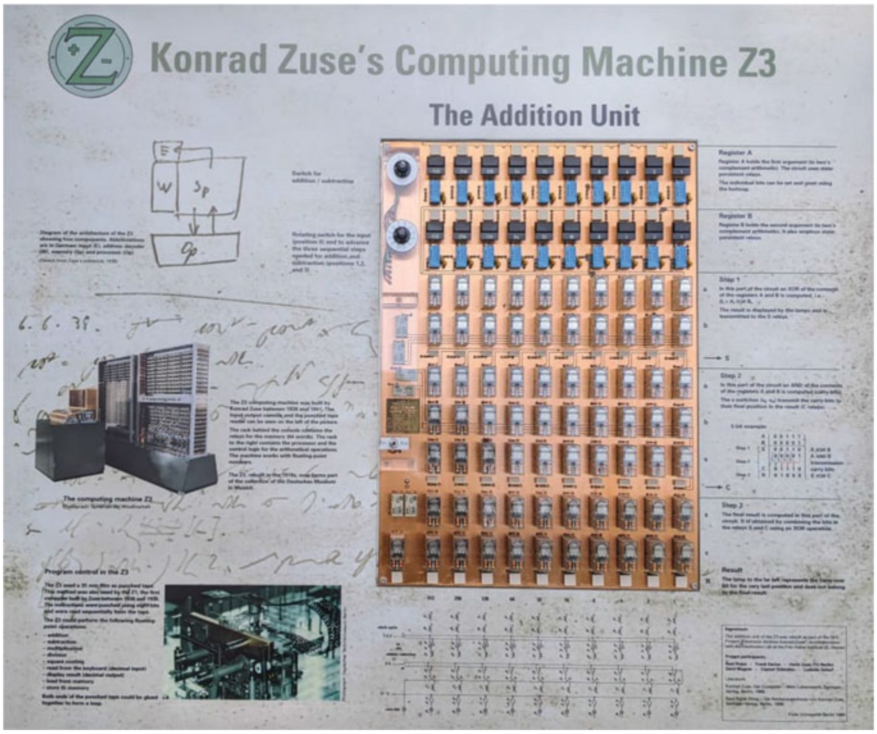


**Fig. 13.2** Panel with the reconstructed addition unit of the Z3. The text around the circuit explains the operation of the adder. The upper two rows of buttons in the printed board allow the user to set two numbers for addition or subtraction. All results are shown with LEDs

information about the addition unit and the circuits. Twelve panels were produced, 10 for German universities, one for the University of Pennsylvania (home of the ENIAC), and one for Computer Museum in Mountain View, California. Figure 13.3 shows a Java simulation of the addition circuit (available also at the Konrad Zuse Internet Archive). The simulation can run addition or subtraction forward or in reverse (using the arrows), so that interested viewers can better grasp the addition method.

We mention the reconstruction of the addition unit because it was a "proof of concept" and because it illustrates the idea that guided us later during the reconstruction of the complete Z3—we did not want to just build a replica of the machine. The Z3 in Munich was already there. We wanted to build a machine that would contain the original circuits, but would be "transparent," one that could be understood by interested people. We wanted to show the data flow in the machine using LEDs, and to make possible single-step operation. This would make the Z3 understandable for anyone who has taken a first-year course in logic design. It would save for posterity not just the flesh (the hardware) of the machine, but its spirit.

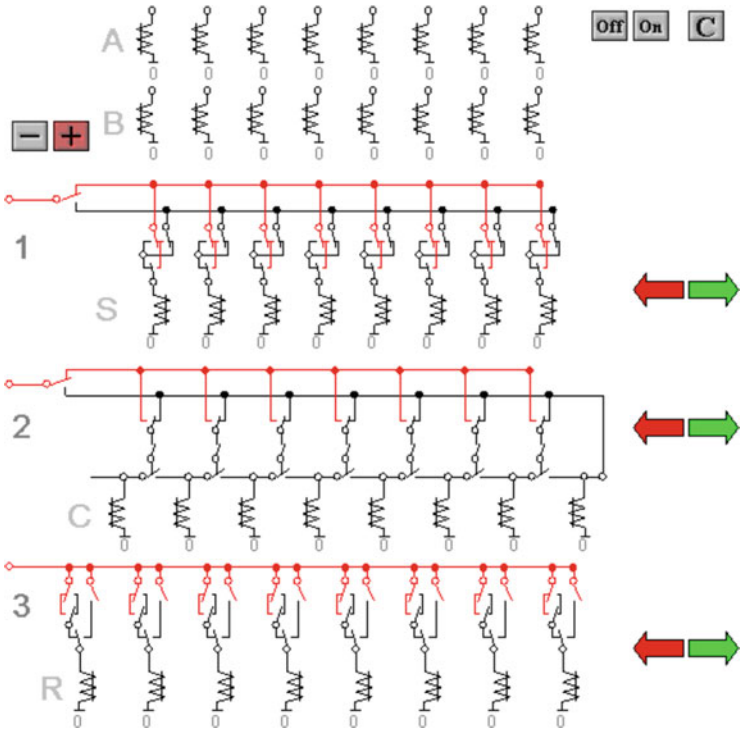


**Fig. 13.3** Java simulation of the circuit for the addition unit (available on the Internet at the Konrad Zuse Internet Archive http://zuse.zib.de/ until browsers discontinued Java applets)

## 13.5 Full Reconstruction of the Z3

After the addition unit was completed, we started thinking about a more ambitious project, the reconstruction of the whole machine. Horst Zuse, the son of Konrad Zuse, who had already closely followed our work with the addition unit, had been thinking in the same way and it was natural that we should attempt to build the machine together. Horst Zuse started a very serious and successful fundraising effort. Especially through the generosity of private donors and some institutions, it was possible to raise circa 75,000 US dollars, which went into the reconstruction effort. In collaboration with Horst Zuse, we proceeded to do a full design of the machine to be built.

Reconstructing the Z3 was largely a vast team effort. Raul Rojas wrote down the block diagram of the circuits that would be implemented, taking them straight out of the patent application. We decided to avoid building any mechanical parts (the console, for example). Emphasis would be put on the logical circuits. The mechanical console would be simulated with a computer display. The user would interact with the virtual console by pressing buttons with the mouse. The tape puncher would be also simulated with a computer. The reconstruction would therefore consist of two large panels, one for the processor and another for the memory, connected through a cable to the computer in which the console is simulated. This arrangement should allow the machine to run for years to come (at the time of this writing, it has been operational for 22 years).

Frank Darius produced detailed diagrams for each circuit. Since the cabling was done using a printed circuit board and modern relays, some adaptations had to be made. For example, where Zuse had used a relay with multiple connections, sometimes we had to substitute using the available components. However, much effort was made to limit the number of changes. The circuits in the reconstructed Z3 preserve, as well as possible, the original design of the Z3. Georg Heyne selected the components to be used and led the layout and fabrication team at the Max-Planck-Gesellschaft in Berlin. Peter Zilske acted as a consultant, and Torsten Vetter coupled the reconstructed Z3 with the console computer. Wolfram Däumel did the layout of the printed circuits and Lothar Schönbein soldered components. Cüneyt Göktekin wrote a full Java emulation of the console, so that the Z3 could be operated as the original was, but using the virtual console.

Figure 13.6 shows a photograph of the front and back views of the memory panel. Figure 13.4 shows the front view of the processor and Fig. 13.5 the rear view. The relays are visible, as are the connections made using ribbon cable or the lines printed on the board.

In Fig. 13.4 the relays are clearly visible, arranged in rows. Each row is a register or a partial result. Information flows from top to bottom when the processor is working. After an addition step, the result is stored back to the top register and the Z3 continues. Each relay has a small red LED glued to its lower edge.

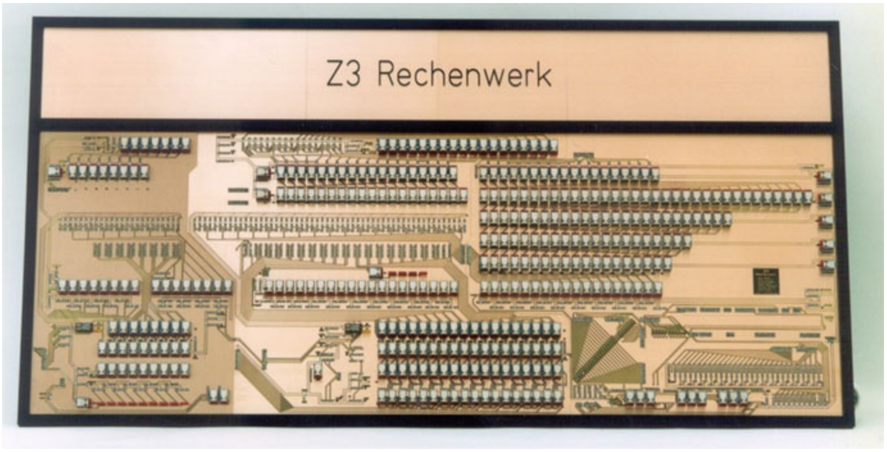


**Fig. 13.4** Frontal view of the processor (Photograph: G. Heyne)

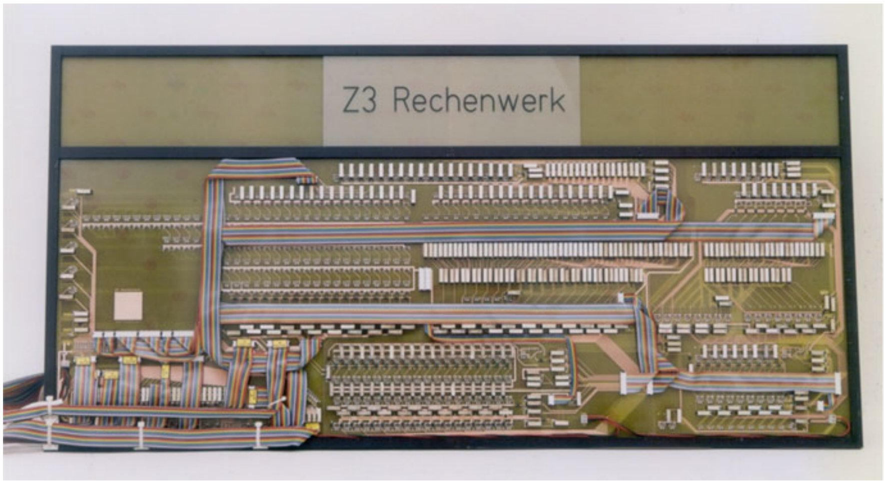


**Fig. 13.5** Rear view of the processor (Photograph: G. Heyne)

In Fig. 13.5 the ribbon cables are part of the bus in the Z3. It was easier to wire parts of the machine using them. This limited the number of layers in the printed circuit to two. Relays are also visible in the rear of the processor. These are the relays used for opening or closing the data path.

Figure 13.6 shows the Z3 memory. Each memory word is a row of relays. Each relay has a red LED underneath. The tree of lines is the decoding tree needed to access one memory address and the circuit is the original one used by Zuse. Only the arrangement is new: it illustrates the decoding tree structure of the circuit. Small LEDs light up the path to a memory address that has been selected. The figure also shows the rear view of the memory panel. Additional relays are also visible here. The FPGAs on top allow us to connect the memory unit to a computer using a serial cable. The memory unit was completed before the processor was ready, and we could demonstrate the use of the Z3 memory using the virtual console without the processor.

Of course, the original Z3 had no lights for signaling the state of components. This is a pedagogical addition that we think greatly increases the value of the machine. This Z3 was built to be understood. When the machine runs, the flickering lights show which operation is being performed and the corresponding data flow. It is an impressive experience to see the Z3 crunching numbers with its primitive relays.

The clocking for the whole Z3 can be modified from the connected computer, so that the machine runs at single step or at different speeds: faster than the original or as fast as the original. This feature was important for debugging purposes when the machine was being built and is used at the museum when the operation of the Z3 is explained.

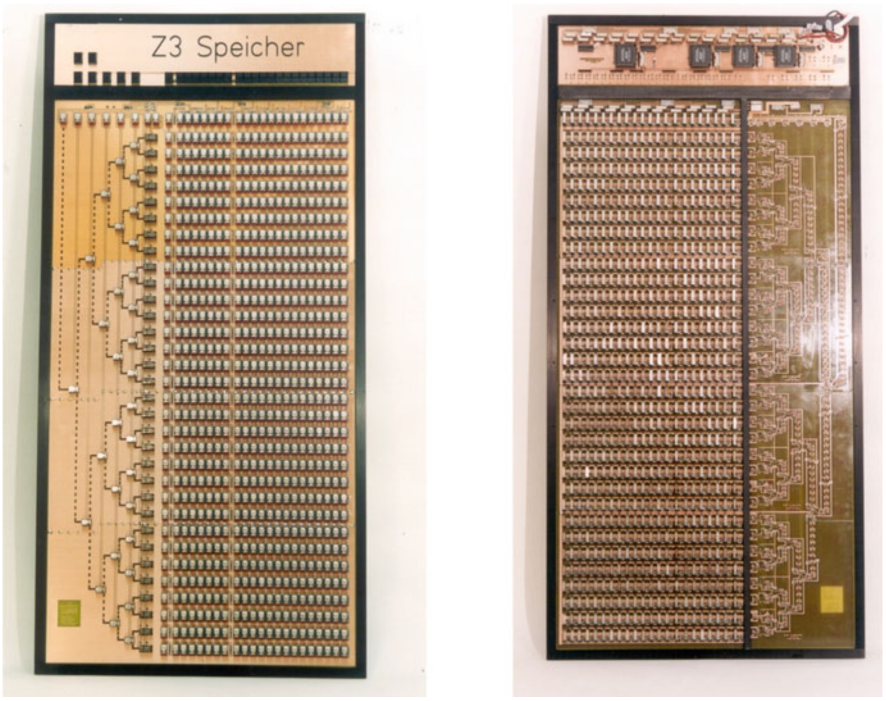


**Fig. 13.6** Frontal and rear views of the memory panel (Photograph: G. Heyne)

The reconstructed Z3 is resilient. Two years after we finished building the main circuits, only one memory bit had failed. The corresponding relay was changed, and no other error has been reported. This proves that our decision to eliminate all moving parts and simulate the console with a computer screen was sound.

During a ceremony in March 2003, the Z3 was bought by the Konrad-Zuse-Museum in Hünfeld, and private donations were paid back to the donors.

## 13.6 The Virtual Z3

Anyone interested in the architecture of the Z3 can now consult the original sources stored in the Konrad Zuse Internet Archive. It is also possible to run simulations of individual circuits. Our last addition was a 3D simulation of the Z3 kept in Munich at Deutsches Museum. Dr. Martin Kurze and his student Alexander Knabner programmed an interactive VRML model of the Z3. The Internet viewer can navigate in a virtual room and see the machine from different perspectives. The "skin" of the model is actual photographs of the Z3 reconstruction in Munich. The user can enter numbers and start the program tape by pressing buttons in the 3D console model (Fig. 13.7).

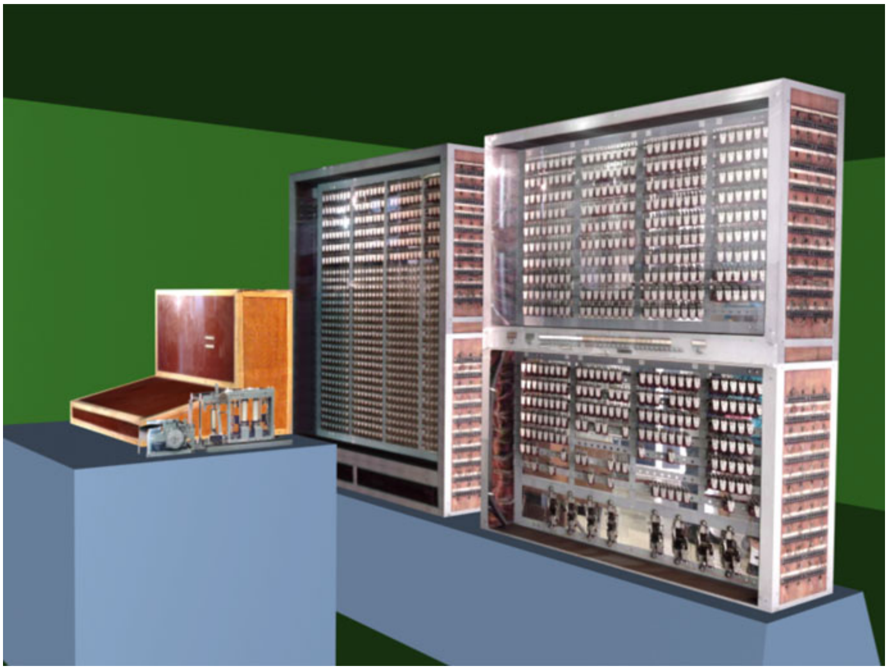


**Fig. 13.7** The 3D model of the Z3

This VRML simulation of the Z3 complemented our hardware reconstruction. Although the machine was in Munich, it was almost lost to history since, especially after Zuse's death, nobody else completely understood how it worked. Now the Z3 lives and computes again.

## 13.7 Conclusions

In 1997, when the first Java simulation of the Z3 was shown at Martin Luther University by our student Alexander Thurm, one colleague asked the very same question we have been hearing for years: "What is that good for?". Explicitly or implicitly, it is assumed that there is nothing new to learn and discover by studying and reconstructing historical machines. Computer scientists disregard, often with contempt, whatever results were obtained more than 10 years ago. In a field that is advancing exponentially fast, according to Moore's law, there is a tendency to think there is no time to look back. Our answer, then and now, is the following: preserving these old machines and understanding how they worked means preserving an important part of our cultural heritage. As computer scientists, we should be interested in knowing how our predecessors thought and by what means they achieved the same objectives we have now. We are not "people without history," as the expression goes.

As scientists, we are interested in living not dead artifacts. The latter is any machine in a museum, collecting dust, and which nobody knows how it once worked. The living artifact is the machine whose inner workings we understand. It is the machine whose intellectual content has been saved for posterity. It is the machine that by working proclaims the greatness of its invention. Rebuilding older computers means bringing them alive for future generations, and now more than ever, since we can recreate old machines in virtual environments where everybody can use and understand them.

Rebuilding old computers is first and foremost preserving culture (Fig. 13.8). And there is also an important educational component. Our experience is that computer science students often find seminars about the history of computing boring—one of those requirements they just have to pass. However, when they have to rebuild the machine by writing a functional simulation, they soon find out how challenging the task is and how wonderful such old machines are. We have not known a student, who having written a simulation of the Z3, Z1, ENIAC, or Turing's APL is not full of admiration for the inventors of these machines.

Rebuilding historical machines gives students a lesson they will never forget: we all truly stand on the shoulders of giants.

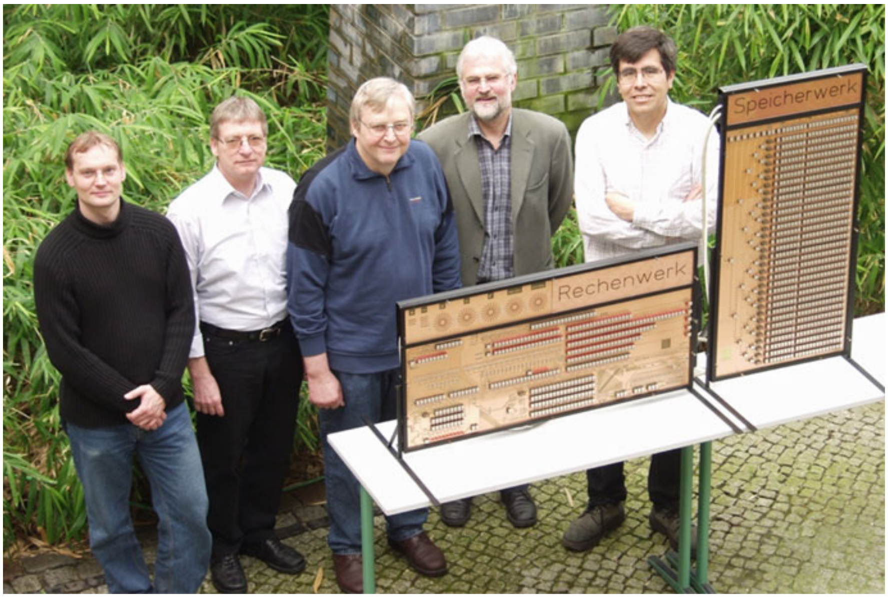


**Fig. 13.8** Five members of the Z3 reconstruction team. From left to right: Torsten Vetter, Lothar Schönbein, Peter Zilske, Georg Heyne, and Raúl Rojas. Frank Darius, Cüneyt Göktekin, and Wolfram Däumel are not in the picture (Photograph: G. Heyne)

**Acknowledgments** We would like to thank Horst Zuse, who made possible the reconstruction project through his energy and untiring enthusiasm. Private donors financed the reconstruction. Several institutions were also involved: the Konrad Zuse Gesellschaft, the Deutsche Forschungsgemeinschaft, the Konrad Zuse Zentrum, and the FU Berlin. Alexander Thurm wrote the Java functional simulation. Other individuals who contributed are mentioned in the text. They all made possible the success of our project. Many person-years went into the design of the reconstructed Z3. For us, the more astonishing realization at the end of the project was that it took us almost as long as it took Konrad Zuse to build the original Z3, although we had modern technology at our command.

## References

Ceruzzi, P.E. 1983. *Reckoners: The prehistory of the digital computer, from relays to the stored program concept, 1935–1945*. Westport Connecticut: Greenwood Press.

Petzold, H. 1992. *Moderne Rechenkünstler: die Industrialisierung der Rechentechnik in Deutschland*. Munich: C.H. Beck.

Rojas, R. 1994. On basic concepts of early computers in relation to contemporary computer architectures. In *IFIP 13th World Computer Congress*, 324–331.

Rojas, R. 1996b. Die Architektur der Rechenmaschinen Z1 und Z3 von Konrad Zuse. *Informatik-Spektrum* 19(6): 303–314. https://doi.org/10.1007/s002870050041.

Rojas, R. 1997. Konrad Zuse's legacy: The architecture of the Z1 and Z3. *IEEE Annals of the History of Computing* 19(2): 5–16. https://doi.org/10.1109/85.586067.

Rojas, R. 1998a. *Die Rechenmaschinen von Konrad Zuse*. Berlin: Springer-Verlag. https://doi.org/10.1007/978-3-642-71944-8.

Rojas, R. 2000. Simulating Konrad Zuse's computers. *Dr Dobbs Journal* 316: 64–69.

Rojas, R. 2001. Konrad Zuse – War der Erfinder des computers doch kein Musterschüler? Telepolis.de

Rojas, R., and U. Hashagen. eds. 2001. *The first computers: history and architectures*. Cambridge, MA: MIT Press

Rojas, R., Darius, F., Göktekin, C., and G. Heyne. 2005. The reconstruction of Konrad Zuse's Z3. *IEEE Annals of the History of Computing* 27(3): 23–32. https://doi.org/10.1109/MAHC.2005.48.

Zuse, K. 1970. *Der computer – Mein Lebenswerk*. Landsberg: Verlag Moderne Industrie.

# Chapter 14
# Epilogue

*Due to the nature of the method shown, science presents itself as an intertwined circle in which the beginning, the simple reason, loops back to the end, the mediation. This circle is a circle of circles because each individual link, as an animated part of the method, is a self-reflection that, while returning to the beginning, is at the same time the beginning of a new link.*[1]

As we conclude our tour through the intricate workings of Konrad Zuse's computer architectures, we bear witness to his most innovative decade—a span of nearly 10 years of unprecedented creativity. I hope that with each passing chapter, the reader could gain a clearer understanding of the progression of this story: from Zuse's earliest musings about computation and binary machines to the completed Z4 and Plankalkül. What truly stands out in this long odyssey is Zuse's early development of the floating-point architecture, of "von Neumann" type, which he deemed essential for his "computer for the engineer," and his steadfast dedication to it in all subsequent iterations, from the Z1 to the Z4, culminating in the proposal for a high-level programming language.

Zuse was not a theorist like Turing. He had a very pragmatic approach to computation, which he once simply defined as "obtaining new results from given data." To me, this suggests that once a formula or set of formulas is given, the computation proceeds without the need to consider alternative paths. The example he uses over and over again in various documents is the computation of determinants or the computation of static force arrangements. Therefore, he did not include the conditional branch in any of his early machines until the ETH asked him to add it to the Z4. The absence of the conditional branch from the instruction set is indeed puzzling, especially considering that the hardwired microcode of the Z1, and all the way up to the Z4, was full of conditional tests and alternative computations. In the Plankalkül draft, Zuse argues that the inclusion of a conditional branch instruction

[1] Hegel, *Wissenschaft der Logik*.

R. Rojas, *Konrad Zuse's Early Computers*, History of Computing,
https://doi.org/10.1007/978-3-031-39876-6_14

in the instruction set would have required more storage, so that the program could be executed from the memory unit instead of from a punched tape (Zuse, 1972). It is worth noting that even Babbage, in his design of the Analytical Engine, considered the idea of rewinding the punched cards to implement conditional loops (Rojas, 2021b).

## 14.1 A Hierarchy of Architectural Levels

Now that we have a clear overview of the machines that Zuse designed up to 1945, it is easy to see that he completed a theoretical program that he did not publish but left sketched in his notebooks (at the time that he was finishing the Z1).

In his autobiography, Konrad Zuse refers to some important entries in this notebook. On June 19, 1937, in the midst of constructing the rods and plates for the Z1, he wrote in his shorthand notes: "Realization that there are elementary operations in which all arithmetic and mental operations can be expressed. A primitive type of mechanical brain consists of a memory unit, a control unit, and a simple device capable of handling simple chains of two to three conditions. With this type of brain, it should theoretically be possible to solve all mental problems that can be captured by mechanisms, regardless of the time required." (Figure 14.1 shows the drawing that goes with this quote).

What the young Zuse means by this, is that every computing machine consists of logic gates, and every computation can be broken down into a combination of elementary logic operations. In any given circuit, each bit can be computed, stored temporarily, and be reused for subsequent operations. It is well known that the gates AND, OR, and NOT provide a basis for logic operations, meaning that any computer can be constructed with these building blocks. If one wants to use only one type of gate, one can use NAND or NOR. The other operations can be obtained from these basic components. Zuse also searched for this elementary operation and thought that the equivalence of two bits (i.e., the negation of XOR) could be such a universal gate.

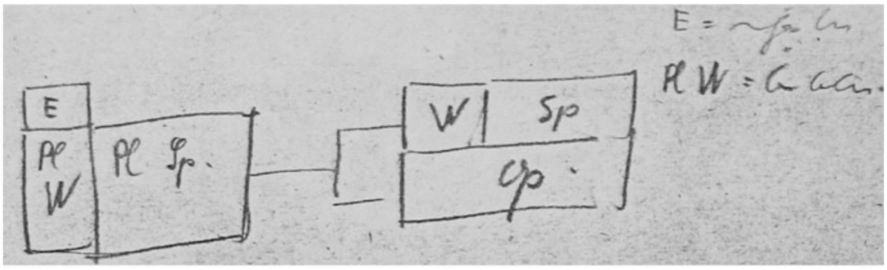


**Fig. 14.1** Drawing of the basic computing machine. The instructions from the program (E) are selected one by one by a controller (PL W) from the program memory (PL Sp). The instructions are executed by the processor (Op) that can access the data memory (Sp) through the address decoder (W)

He later realized that this was not the case. However, the key idea is that one can build a computer with a minimal processor that actually performs only simple logic operations of two bits in each cycle, stores the result, and then starts the next logic operation. For this purpose, one only needs a very long memory for individual bits, a processor that can execute atomic logic operations, and a long program, since any complex computation (such as the sum of two 32-bit numbers) must be broken down into many elementary operations. Instead of processing 32-bit words in parallel, the operations are performed sequentially, i.e., bit by bit. This can reduce the complexity of the computing machine to a minimum, if the execution time is not taken into account. This is the earliest description of what would later become Zuse's proposal for a "logistic computer."

In the summer of 1938, only one year after writing the aforementioned note, Zuse returned to the problem and proposed a hierarchy of control mechanisms. Instead of writing programs directly for the minimal machine, one could write programs using high-level instructions, which are then interpreted by the hardware as a sequence of microinstructions. For example, the multiplication operation can be transformed into individual additions of the shifted multiplicand. An addition can also be transformed into a sequence of operations, such as XOR of pairs of bits, carry calculations, and another XOR of the intermediate results. Zuse wrote in 1938:

$E_0$ = Counter for main plan
$E_1$ = Counter for sub-plan.
(...)
For external commands, the currently operating controller $E_i$ provides the next command. Internal commands cause a change in the relevant counter, with $E_1$ (sub-plan counter) continuing from the desired instruction, and E0 continuing where it last left off.

This is actually what Zuse did in the Z3 and Z4. The $E_0$ controller selects the arithmetic operations in the program, one by one, and the controller $E_1$ refers to the rotary dials that reduce each arithmetic operation to a sequence of microinstructions. The same was achieved in the Z1, but using a stack of addressable metal plates (Fig. 14.2).

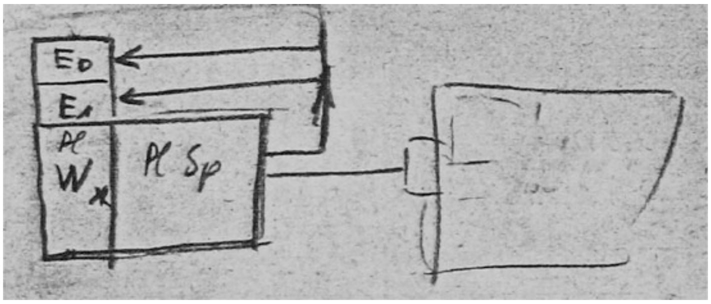


**Fig. 14.2** In this drawing (Zuse notebooks, 1938) the program executed at the level $E_0$ transforms into instructions for the lower level $E_1$ that are executed by the processor. Both controllers, $E_0$ and $E_1$ keep track of their sequence of operations

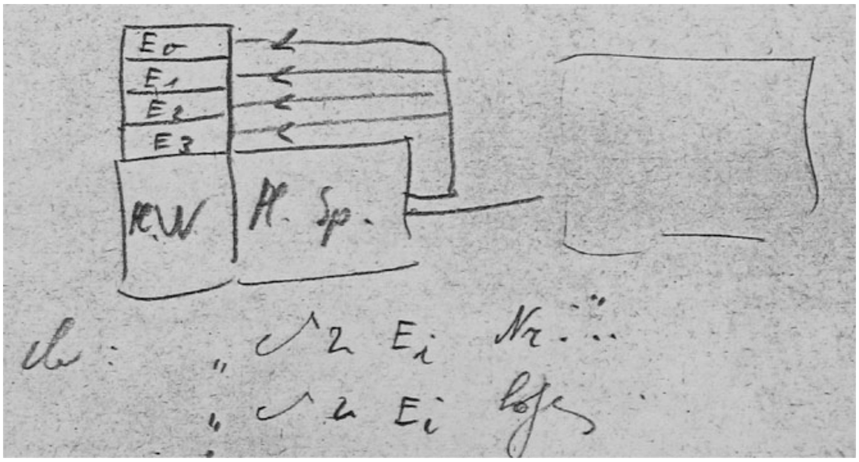


**Fig. 14.3** A full hierarchy of execution levels can be implemented. The highest-level description of the computation at $E_0$ is transformed into instructions for level $E_1$. This level is transformed into instructions for level $E_2$ and so on

Continuing in the same notebook, Zuse then proposes a complex hierarchy of controllers and levels of description for the computation at hand. The highest-level description is represented by "$E_0$". The instructions in the controller $E_0$ transform into instructions for the controller $E_1$, the instructions for $E_1$ into instructions for the controller $E_2$ and so on. Zuse calls this "multiple nesting." Note also that in the diagram attached to the note, Zuse now fuses the program and data memory into a single memory unit (Fig. 14.3).

This hierarchy of execution levels is exactly the concept that guided Zuse's work around the Plankalkül and the logic machine. The latter would be the minimal processor, while even Plankalkül had at least two levels: in the "implicit form," the programmer only specified the computation using a predicate logic or set-theoretic description (such as: "select all $x$ that fulfill $f(x)$"). This form was then transformed into the "explicit form," which is the required imperative program that represents (or implements) the desired computation. The implicit form would be the $E_0$ level in the diagram and the explicit form would be the $E_1$ level. Then we would need an interpreter $E_2$ for the individual Plankalkül imperative statements, a level $E_3$ for the execution of arithmetic operations, and so on. At the end of the chain, we would have the logic operations provided by the logic machine. It is evident that toward the end of his most creative period, Zuse successfully completes the theoretical program he initially sketched in 1937/38.

If ever there was a "discursive circumnavigation," like the one described by Hegel in the quote that opens this chapter, it is this one. By 1945, Zuse develops the highest level for the specification of computations (the predicate calculus form of Plankalkül) and grounds everything on a minimal computer actually quite similar to a Turing machine (without being equivalent).

Alan Turing would have been proud of the young Zuse.

## 14.2 Celebrating the Z1 in 2038?

But there is one more thing: I hope that we will be able to resurrect the Z1 before 2038, one hundred years after its creation. Not with metal plates, but as a virtual 3D reconstruction in the computer. What we need to do is to capture photographically every single component of the machine, so that we can write a 3D simulation of the interplay of all parts. Allow me to elaborate.

The reconstruction of the Z1 in the German Museum of Technology in Berlin was built using many layers of mechanical components (see Fig. 14.4). The different layers can be removed, starting from the top, one by one. We can position a digital camera above the machine and take pictures of every single component (except bolts and screws), before removing them from the Z1. To have a 3D reference, we could place red dots on each photographed component to capture its exact placement. The camera can be calibrated so that pixel values can be reliably associated with 3D-Euclidian coordinates. The thickness of each component can be manually measured.

Once a component has been removed from a layer, it is placed on a stage below a second camera. An image is taken. The background will have a salient color (blue, for example) so that all the edges of the components can be detected automatically. Since Zuse used mainly flat plates and some rods, in most cases there is no need for a 3D measurement. The 2D image from the camera is sufficient. Moreover, the dimensions of the parts were drawn in the 1980s using a grid, and this grid provides a very good initial reference for the dimensions of each part.

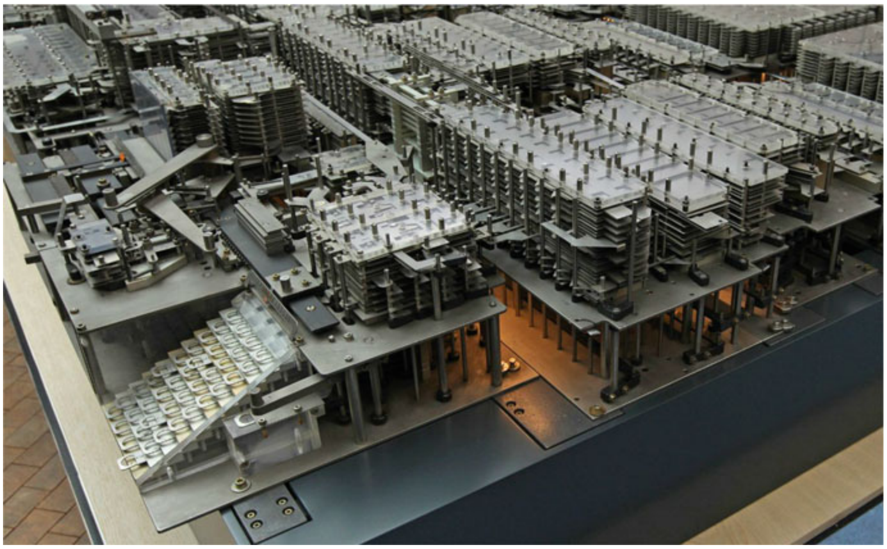


**Fig. 14.4** The reconstructed Z1 in the German Museum of Technology in Berlin

Levers and rods will be handled as special cases. Some levers are composite assemblies of primitive shapes. Vertical rods will be just defined by hand (their radius is the same everywhere, only their height is variable). It is more difficult to capture the levers in the lowest part of the machine. These levers are used to carry the machine cycling pulses. They are few and could be measured manually.

A 2D-CAD model can be automatically generated from the images of the components. The position of each part in the machine is given by the image taken before with the ceiling camera. A computer can put together the CAD models of all the parts and generate a genuine and accurate virtual model of the Z1. Some manual intervention might be required to make all parts fit, but it would be minimal. In fact, we have already simulated subcomponents of the Z1, such as the addition unit (see Fig. 14.5), using Zuse's patents' applications as a reference (which unfortunately do not cover the complete and exact design of the Z1).

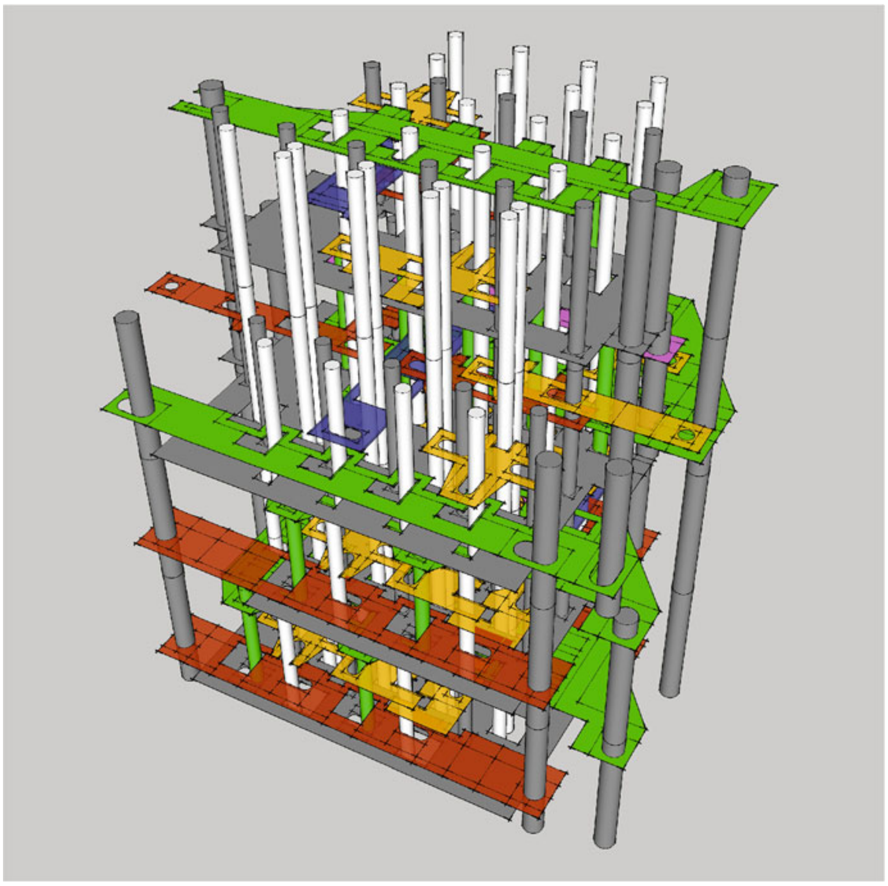


**Fig. 14.5** Virtual simulation of a mechanical binary adder using Zuse's mechanical components (Mischek, 2012)

The most challenging aspect of the project is dismantling the machine, layer by layer, with the utmost care to ensure successful reassembly. The generation of the CAD model could be created simultaneously with the mechanical work. The time required to capture all the parts is perhaps twice the time required to disassemble and reassemble the machine.

Then we could write a simulation. The different parts of the Z1 can adopt one of two positions (0 or 1). Only the pins used for connecting layers can have more than two positions. The important thing is that a simulation can be derived from the CAD model, without having to simulate the physics of the mechanical parts, because all the movements represent the data processing logic. For example, if we know that a bit that was 0 has been reset to 1, we know that the corresponding mechanical part has moved from its 0-position to its 1-position. Therefore, we only need to put together a model of the logical connections of all the parts in the machine (the detailed computational model, in fact) and then the movement of the virtual parts can be generated by the model, following the cycles of the machine.

Thus, it would be feasible to simulate the 30,000+ parts of the machine (for 6000 gates) in real time. Users would be able to input their own numerical data and actively interact with the Z1 simulation. Initially, we may target a desktop application, but it may also be possible to run the simulation from a web browser. While the primary goal of this project would be to create a simulation, there is the intriguing possibility of producing miniature or full-scale 3D models of the Z1 for museums around the world. These models, while non-functional, could be printed using materials such as titanium or other metals.

We could thus celebrate the Z1, the first German computer, by doing something extraordinary: reviving it. I hope that, when this happens, this book will be remembered as an important milestone toward a greater understanding of Zuse's early computers and for their preservation as an invaluable part of the world's cultural heritage. I sincerely hope that some future readers will rally around this wonderful endeavor.

## 14.3 Acknowledgments for Figures

The great majority of the illustrations for this book were drawn by the author or produced for the author by illustrators. Some diagrams are from our public domain repository *Konrad Zuse Internet Archive*. Some photographs were part of our projects with the Archive or the reconstruction of the Z3. The table below provides an overview of the individual sources of certain photographs and diagrams.

| Figure | Page | Attribution |
|---|---|---|
| | xi | Konrad Zuse Internet Archive: http://zuse.zib.de/ |
| 1.2 | 3 | Zuse (1970) |
| 1.3 | 4 | Zuse (1970) |
| 1.4 | 6 | Deutsches Museum, Munich, Archive CD-57258 |
| 2.1 | 25 | Wikimedia Commons, public domain |
| 2.2 | 23 | University Archives Princeton University |
| 2.3 | 25 | Original Author: User:San Jose Derivative Author: User:ArmadniGeneral (https://commons.wikimedia.org/wiki/File:Second-world-war-europe-1941-1942-map-en.png), "Second world war europe 1941–1942 map en", https://creativecommons.org/-licenses/by-sa/3.0/legalcode |
| 2.4 | 27 | Hopper et al. (1946) |
| 2.5 | 28 | Wikimedia Commons, public domain |
| 2.6 | 29 | Wikimedia Commons, public domain |
| 2.7 | 34 | Deutsches Museum, Munich, Archive CD-57196 |
| 5.1 | 90 | Konrad Zuse Internet Archive: http://zuse.zib.de/ |
| 8.1 | 136 | Deutsches Museum, Munich, Archive CD-67142 |
| 8.3 | 140 | ETH Library |
| 8.5 | 144 | Deutsches Museum, Munich, Archive CD-73322 |
| 7.1 | 121 | public domain |
| 7.2 | 122 | Wikimedia Commons, public domain |

## References

Bower, T. 1987. *The paperclip conspiracy – The hunt for Nazi scientists*. Boston: Little, Brown and Company.

Giles, G.J. 1985. *Students and national socialism in Germany*. Princeton: Princeton University Press.

Hopper, G.M., Aiken, H.H., Bloch, R.M., and R.L. Hawkins. 1946. *A manual of operation for the automatic sequence controlled calculator*, vol. 1. Cambridge, MA: Harvard University Press.

Mischek, J. 2012. 3D-Simulation des Additionswerkes der Z1 von Konrad Zuse. Master's thesis, Freie Universität Berlin.

Rojas, R. 2021b. The computer programs of Charles Babbage. *IEEE Annals of the History of Computing* 43(1): 6–18. https://doi.org/10.1109/MAHC.2020.3045717.

Zuse, K. 1936d. Patentanmeldung Z 23 139 IX / 42m: Verfahren zur selbsttätigen Durchführung von Rechnungen mit Hilfe von Rechenmaschinen [für Zuse]. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0990/.

Zuse, K. 1970. *Der computer – Mein Lebenswerk*. Landsberg: Verlag Moderne Industrie.

Zuse, K. 1972. *Der Plankalkül* (vol. 63). Sankt Augustin: Berichte der Gesellschaft für Mathematik und Datenverarbeitung.

# References

Aiken, H.H., and G.M. Hopper. 1982. The Automatic Sequence Controlled Calculator. In *The Origins of Digital Computers*, ed. B. Randell, 203–222. Monographs in Computer Science. Berlin: Springer. https://doi.org/10.1109/EE.1946.6434251.

Anonymous. 1947. Objective List of German and Austrian Scientists. Joint Intelligence Objectives Agency.

Anonymous. 1981. Documentation: Konrad Zuse and the Early Days of Scientific Computing at the ETH. ETH Zurich, 17.6.-15.7.1981, the documentation contains the list of commands for the Z4.

Anonymous. 1995. Erklärung des Fachbereichs 10, TU Intern, Newsletter.

Arefi, F., Hughes, C.E., and D.A. Workman. 1990. *Automatically generating visual syntax-directed editors*. *Communications of the ACM* 33(3): 349–360. https://doi.org/10.1145/77481.77487.

Babbage, C. 1864. *Passages from the life of a philosopher*. London: Longman, Green, Longman, Roberts & Green. https://doi.org/10.1017/CBO9781139103671.

Bauer, F.L. 2008. Über "Episoden aus den Anfängen der Informatik an der ETH" von A. Speiser. *Informatik-Spektrum* 31(6): 600–612. https://doi.org/10.1007/s00287-008-0291-8.

Bauer, F.L. 2009. Helmut Schreyer – ein Pionier des "elektronischen" Rechnens. In *Historische Notizen zur Informatik*. Berlin: Springer. https://doi.org/10.1007/978-3-540-85790-7.

Bibel, W. 2014. Artificial intelligence in a historical perspective. *AI Communications* 27(1): 87–102. https://doi.org/10.3233/AIC-130576.

Bibel, W. 2020. On the development of AI in Germany. *Künstliche Intelligenz* 34(2): 251–258. https://doi.org/10.1007/s13218-020-00654-x. Open Access https://rdcu.be/b3oxS.

Bower, T. 1987. *The paperclip conspiracy – The hunt for Nazi scientists*. Boston: Little, Brown and Company.

Bromley, A.G. 2000. Babbage's analytical engine plans 28 and 28a – The programmer's interface. *Annals of the History of Computing* 22(4): 5–19. https://doi.org/10.1109/85.887986.

Bruderer, H. 2012. *Konrad Zuse und die Schweiz: Wer hat den Computer erfunden?* Munich: Oldenbourg Wissenschaftsverlag. https://doi.org/10.1524/9783486716658.

Bruderer, H. 2020. *Milestones in Analog and Digital Computing*. Vols. 1 and 2, 3rd ed., 2075. Cham: Springer Nature Switzerland. https://doi.org/10.1007/978-3-030-40974-6.

Burks, A.R., and A.W. Burks. 1988. *The First Electronic Computer – The Atanasoff Story*. Ann Arbor: The University of Michigan Press.

Burks, A.W., and A.R. Burks. 1981. The ENIAC: First General-Purpose Electronic Computer. *Annals of the History of Computing* 3 (4): 310–399, https://doi.org/10.1109/MAHC.1981.10043.

Butzer, P.L., M.M. Dodson, P.J.S. Ferreira, J.R. Higgins, O. Lange, and P. Seidler. 2009. Herbert Raabe's Work in Multiplex Signal Transmission and His Development of Sampling Methods. *Signal Processing* 90 (5): 1436–1455. https://doi.org/10.1016/j.sigpro.2009.11.018.

Campbell-Kelly, M., W. Aspray, N. Ensmenger, and J.R. Yost. 2013. *Computer: A History of the Information Machine*. 3rd ed. The Sloan Technology Series. https://doi.org/10.4324/9780429495373.

Ceruzzi, P.E. 1983. *Reckoners: The prehistory of the digital computer, from relays to the stored program concept, 1935–1945*. Westport Connecticut: Greenwood Press.

Copeland, B.J. 2004. *The essential turing: The ideas that gave birth to the computer age*. Oxford: Oxford University Press. https://doi.org/10.1093/oso/9780198250791.001.0001.

Czauderna, K.H. 1979. Konrad Zuse, der Weg zu seinem Computer Z3. In *Berichte der Gesellschaft für Mathematik und Datenverarbeitung*. Munich: Oldenbourg Verlag.

Delius, F.C. 2009. *Die Frau, für die ich den Computer erfand*. Berlin: Rowohlt Verlag.

Ensmenger, N. 2012. Is chess the drosophila of artificial intelligence? A social history of an algorithm. *Social Studies of Science* 42(1): 5–30. https://doi.org/10.1177/0306312711424596.

Giles, G.J. 1985. *Students and national socialism in Germany*. Princeton: Princeton University Press.

Goldstine, H.H., and A. Goldstine. 1996. The electronic numerical integrator and computer (ENIAC). *Annals of the History of Computing* 18(1): 10–16. https://doi.org/10.1109/85.476557.

Grosch, J., and H. Emmelmann. 1991. A tool box for compiler construction. In *Compiler Compilers. CC 1990*, ed. D Hammer. Lecture Notes in Computer Science, vol. 477, 106–116. Berlin, Heidelberg: Springer. https://doi.org/10.1007/3-540-53669-8-77.

Hachtmann, R. 2003. *Science Management in the "Third Reich". History of the General Administration of the Kaiser Wilhelm Society*. Göttingen: Wallstein Verlag.

Harel, D. 1980. On Folk Theorems. *Communications of the ACM* 23(7): 379–389. https://doi.org/10.1145/358886.358892.

Hodges, A. 2006. *Alan turing: The enigma*. New York: Walker & Company.

Hopper, G.M., Aiken, H.H., Bloch, R.M., and R.L. Hawkins. 1946. *A manual of operation for the automatic sequence controlled calculator*, vol. 1. Cambridge, MA: Harvard University Press.

Hsu, F.H., Anantharaman, T., Campbell, M., and A. Nowatzyk. 1990. A Grandmaster chess machine. *Scientific American* 263(4): 44–51.

Ibarra, O.H., S. Moran, L.E. Rosier. 1983. On the Control Power of Integer Division. *Theoretical Computer Science* 24(1): 35–52. https://doi.org/10.1016/0304-3975(83)90129-9.

Jarausch, K.H. 1993. The Expulsion of Jewish Students and Professors from Berlin University under the Nazi Regime. Lecture June 15.

Knuth, D.E. 1981. *The Art of Computer Programming – Seminumerical Algorithms*, Vol. 2. Boston: Addison-Wesley Professional.

Koren, I. 1993. *Computer Arithmetic Algorithms*. Englewood Cliffs: Prentice Hall.

Kurrer, K.E. 2010. Konrad Zuse und die Baustatik – Zur Vorgeschichte der Computerstatik (Teil I). *Bautechnik* 87 (11). https://doi.org/10.1002/bate.201010046

Lavington, S.H. 1975. *A History of Manchester Computers*. Manchester: Blackwell Publishers.

Materna, H. 2010. *Die Geschichte der Henschel Flugzeug-Werke in Schönefeld bei Berlin 1933-1945*. Bad Langensalza: Verlag Rockstuhl.

Meyer, B. 2000. Berlin – Stadt der Nobelpreisträger. Berlinische Monatsschrift 6. https://berlingeschichte.de/bms/bmstxt00/0004prol.htm.

Minsky. M.L. 1967. *Computation: finite and infinite machines*. Upper Saddle River, NJ: Prentice Hall. https://doi.org/10.5555/1095587.

Mischek, J. 2012. 3D-Simulation des Additionswerkes der Z1 von Konrad Zuse. Master's thesis, Freie Universität Berlin.

Pannke, K. 1927. Rechenvorrichtung. German Patent Office, Patent 447780.

Péter, R. 1967. *Recursive Functions*. New York: Academic Press.

Petzold, H. 1985. *Computing Machines – A Historical Study of their Manufacture and Use from the Empire to the Federal Republic*. Düsseldorf: VDI Verlag.

Petzold, H. 1992. *Moderne Rechenkünstler: die Industrialisierung der Rechentechnik in Deutschland*. Munich: C.H. Beck.

Petzold, H. 2004. Hardwaretechnologische Alternativen bei Konrad Zuse. In *Geschichten der Informatik: Visionen, Paradigmen, Leitmotive*, ed. HD Hellige. Berlin, Heidelberg: Springer. https://doi.org/0.1007/978-3-642-18631-8-5.

Poe, E.A. 1836. Maelzels chess player. *Southern Literary Messenger* 2: 318–326.

Randell, B. 1982a. From analytical engine to electronic digital computer: The contributions of ludgate, torres and bush. *Annals of the History of Computing* 4(4). https://doi.org/10.1109/MAHC.1982.10042.

Randell, B., ed. 1982b. *The Origins of Digital Computers, 3rd edn. Monographs in Computer Science*. Berlin: Springer. https://doi.org/10.1007/978-3-642-61812-3.

Range, J. 2016. Validierung und Visualisierung des Mikrocodes der Z3 von Konrad Zuse. Master's Thesis, Humboldt Universität Berlin.

Rojas, R. 1993. Who Invented the Computer? The Debate from the Viewpoint of Computer Architecture. In *Fifty Years Mathematics of Computation*, ed. W. Gautschi, Vol. 48, 361–366. Proceedings of Symposia in Applied Mathematics. https://doi.org/10.1090/psapm/048/1314871.

Rojas, R. 1994. On basic concepts of early computers in relation to contemporary computer architectures. In *IFIP 13th World Computer Congress*, 324–331.

Rojas, R. 1996a. Conditional Branching is not Necessary for Universal Computation in von Neumann Computers. *Journal of Universal Computer Science* 2(11): 756–767. https://doi.org/10.3217/jucs-002-11-0756.

Rojas, R. 1996b. Die Architektur der Rechenmaschinen Z1 und Z3 von Konrad Zuse. *Informatik-Spektrum* 19(6): 303–314. https://doi.org/10.1007/s002870050041.

Rojas, R. 1997. Konrad Zuse's legacy: The architecture of the Z1 and Z3. *IEEE Annals of the History of Computing* 19(2): 5–16. https://doi.org/10.1109/85.586067.

Rojas, R. 1998a. *Die Rechenmaschinen von Konrad Zuse*. Berlin: Springer. https://doi.org/10.1007/978-3-642-71944-8.

Rojas, R. 1998b. How to make Konrad Zuse's Z3 a universal computer. *IEEE Annals of the History of Computing* 20(3): 51–54. https://doi.org/10.1109/85.707574.

Rojas, R. 2000. Simulating Konrad Zuse's computers. *Dr Dobbs Journal* 316: 64–69.

Rojas, R. 2001. Konrad Zuse – War der Erfinder des computers doch kein Musterschüler? Telepolis.de

Rojas, R. 2014a. Konrad Zuse und der bedingte Sprung. *Informatik-Spektrum* 37: 50–53. https://doi.org/10.1007/s00287-013-0717-9.

Rojas, R. 2014b. The Z1: Architecture and Algorithms of Konrad Zuse's First Computer. https://arxiv.org/abs/1406.1886, arXiv eprint.

Rojas, R. 2016. The Design Principles of Konrad Zuse's Mechanical Computers. https://arxiv.org/abs/1603.02396, arXiv eprint.

Rojas, R. 2021a. The architecture of Konrad Zuse's Z4 computer. In *IEEE 7th IEEE History of Electrotechnology Conference*, Moscow, 43–47.

Rojas, R. 2021b. The computer programs of Charles Babbage. *IEEE Annals of the History of Computing* 43(1): 6–18. https://doi.org/10.1109/MAHC.2020.3045717.

Rojas, R., and U. Hashagen. eds. 2001. *The first computers: history and architectures*. Cambridge, MA: MIT Press.

Rojas, R., Göktekin, C., Friedland, G., Krüger, M., Langmack, O., and D. Kunik. 2000. Plankalkül: The First High-Level Programming Language and its Implementation. Technical Report B-3/2000, Freie Universität Berlin.

Rojas, R., Darius, F., Göktekin, C., and G. Heyne. 2005. The reconstruction of Konrad Zuse's Z3. *IEEE Annals of the History of Computing* 27(3): 23–32. https://doi.org/10.1109/MAHC.2005.48.

Rojas, R., Röder, J., and H. Nguyen. 2014. Die Prozessorarchitektur der Rechenmaschine Z1. *Informatik-Spektrum* 37(4): 341–347. https://doi.org/10.1007/s00287-013-0749-1.

Schreyer, H. 1939. Technische Rechenmaschine. Zuse Papers 004/002 www.zib.de/zuse.

Schreyer, H. 1941. Das Röhrenrelays und seine Schaltungstechnik. PhD thesis, TH Berlin.

Schweier, U., and D. Saupe. 1988. Funktions- und Konstruktionsprinzipien der Programmgesteuerten Rechenmaschine "Z1". Arbeitspapiere der Gesellschaft für Mathematik und Datenverarbeitung 321.

Shannon, C.E. 1950. Programming a computer for playing chess. *Philosophical Magazine* 41: 256–275. https://doi.org/10.1007/978-1-4757-1968-0-1.

Speiser, A. 2000. Konrad Zuse's Z4: Architecture, programming, and modifications at the ETH Zurich. In *The first computers - history and architectures*, eds. R. Rojas, U. Hashagen Cambridge, MA: MIT Press.

Stern, N. 1981. *From ENIAC to UNIVAC*. Bedford: Digital Press.

Teitelbaum, T., and T. Reps. 1981. The cornell program synthesizer: A syntax directed programming environment. *Communications of the ACM* 24(9): 563–573. https://doi.org/10.1145/358746.358755.

Tomayko, J.E. 1985. Helmut Hoelzer's Fully Electronic Analog Computer. *Annals of the History of Computing* 7 (3): 227–240. https://doi.org/10.1109/MAHC.1985.10025.

Turing, A. 1948. Document amt/k/1/77. Alan Turing Digital Archive.

Turing, A. 1953. Digital computers applied to games. In *Faster Than Thought*, ed. Bowden BV. London: Sir Isaac Pitman & Sons Ltd.

von Neumann, J. 1945. First Draft of a Report on the EDVAC. Unpublished draft.

von Neumann, J., and O. Morgenstern. 1944. *Theory of games and economic behavior*. Princeton: Princeton University Press.

Zermelo, E. 2010. *Ernst Zermelo – collected works*, 1st edn. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-540-79384-7.

Zuse, H. 2000. Konrad Zuse—Seine Rechenmaschinen. In *Konrad Zuse—Der Vater des Computers*, ed. Alex Jürgen, Flessner Hermann, Mons Wilhelm, Pauli Kurt, and Zuse Horst. Fulda: Verlag Parzeller.

Zuse, K. 1936a. Die Aufstellung der Rechenpläne. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0208/.

Zuse, K. 1936b. Die Rechenmaschine des Ingenieurs. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0209/.

Zuse, K. 1936c. Die Rechenmaschine des Ingenieurs. Mathematische Probleme. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0214/.

Zuse, K. 1936d. Patentanmeldung Z 23 139 IX / 42m: Verfahren zur selbsttätigen Durchführung von Rechnungen mit Hilfe von Rechenmaschinen [für Zuse]. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0990/.

Zuse, K. 1937a. Einführung in die allgemeine Dyadik. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0210/.

Zuse, K. 1937b. Einführung in die allgemeine Dyadik. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0211/.

Zuse, K. 1938. Die Rechenmaschine des Ingenieurs. Available online at the Zuse Internet Archive.

Zuse, K. 1939/1940. Chiffriermaschine. Zuse Archive at Deutsches Museum, images 202_4-01, 202_4-02, 202_4-03.

Zuse, K. 1940. Draft of a letter to Kurt Pannke. Zuse Archive at Deutsches Museum.

Zuse, K. 1941. *Patentanmeldung Z-391*. Berlin: German Patent Office.

Zuse, K. 1941-1942. Stenografische Notizen zum Schachspiel. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0746/.

Zuse, K. 1942a Modell S1/S2. Skizze der Gesamtanlage und Rechenschema. Available online at the Zuse Internet Archive.

Zuse, K. 1942b. Programmgesteuerte Rechenmaschine für Flügelvermessung. Available online at the Zuse Internet Archive.

Zuse, K. 1942c. Vorarbeiten zum Plankalkül. Schachprogramme. Zuse Papers 012/012.

Zuse, K. 1943a. Rechenplangesteuerte Rechengeräte für technische und wissenschaftliche Rechnungen. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0217/.

Zuse, K. 1943b. Rechenplangesteuerte Rechengeräte für technische und wissenschaftliche Rechnungen. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0217/.

Zuse, K. 1944a Gerät S2a Rechengerät für Flügelvermessung mit automatischer Ablesbarkeit der Messuhren. 2. Ausführung des Spezialmodells S2. Available online at the Zuse Internet Archive.

Zuse, K. 1944b. Mechanische Schaltgliedtechnik. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0226/.

Zuse, K. 1944c. Notizen zu den Schaltungen. Available online at the Zuse Internet Archive.

Zuse, K. 1944d. Patentanmeldung Z-394: Rechenvorrichtung. Available online at the Zuse Internet Archive.

Zuse, K. 1944e. Planfertigungsgeräte. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0220/.

Zuse, K. 1945. Theorie der Angewandten Logistik, 2. Buch. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0233/.

Zuse, K. 1946a. Kurze Beschreibung des mechanischen Speicherwerks der Firma Zuse. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0249/.

Zuse, K. 1946b. Zur Entwicklung von Rechengeräten bis zum Jahre 1945. Nennung von Namen und finanziellen Unterstützungen. Available online at the Zuse Internet Archive.

Zuse, K. 1946c. Zuse Calculators. Available online at the Zuse Internet Archive.

Zuse, K. 1948. Über den Plankalkül als Mittel zur Formulierung schematisch kombinativer Aufgaben. *Archiv der Mathematik* 1(6): 441–449.

Zuse, K. 1950. Patentschrift Nr.926449, Kombinierte numerische und nichtnumerische Rechenmaschine. Deutsches Patentamt, 11 Seiten.

Zuse, K. 1952a. *Bedienungsanweisung für Zuse Z4*. Zurich: ETH Zurich. https://doi.org/10.7891/e-manuscripta-98601.

Zuse, K. 1952b. Rechenvorrichtungen aus mechanischen Schaltgliedern. Deutsches Museum Digital. http://zuse.zib.de/.

Zuse, K. 1952c. Rechenvorrichtungen aus mechanischen Schaltgliedern. Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0961/.

Zuse, K. 1952d. Theorie der mechanischen Schaltglieder. Available online at the Zuse Internet Archive.

Zuse, K. 1953a Patentschrift nr. 872645: Verfahren zur Abtastung von Oberflächen und Einrichtung zur Durchführung des Verfahrens (für Zuse). Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0998/.

Zuse, K. 1953b Verfahren zur Abtastung von Oberflächen und Einrichtung zur Durchführung des Verfahrens. German Patent Office, Patent 872645.

Zuse, K. 1954. Patentschrift nr. 907948: Mechanisches Schaltglied (für die Zuse KG). Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-0999/.

Zuse, K. 1955. Patentschrift nr. 924107: Aus mechanischen Schaltgliedern aufgebautes Speicherwerk (für Zuse). Deutsches Museum Digital. http://digital.deutsches-museum.de/item/NL-207-1001/.

Zuse, K. 1970. *Der computer – Mein Lebenswerk*. Landsberg: Verlag Moderne Industrie.

Zuse, K. 1972. *Der Plankalkül* (vol. 63). Sankt Augustin: Berichte der Gesellschaft für Mathematik und Datenverarbeitung.

Zuse, K. ca 1945. Rechenpläne für das Rechengerät V4. Available online at the Zuse Internet Archive.