Konrad Zuse’s Z3 Machine Architecture

by

Michael Beaver

Dr. Patricia Roden

CS 311

31 Oct. 2013

Konrad Zuse’s Z3 Machine Architecture

The history of computing is marked with significant developments and advances in computer architecture and organization. Konrad Zuse, a German civil engineer, was among the first generation of computer developers. His first machine, the Z1, was not a profound success, but his third iteration, the Z3, anticipated many of the advances of his contemporaries, namely John von Neumann, and modern computer scientists. Indeed, the Z3 is a significant anomaly in the history of computing. Although Zuse’s Z3 was developed independently of the early great computer pioneers, it is a historically significant contribution to the development of computers.

**A Brief History**

Konrad Zuse’s foray into designing and constructing computers began in 1936. By 1941, he completed the “first fully operational digital computer,” the Z1 [1]. The Z1, however, was far from perfect: the Z1 was fully mechanical, and the arithmetic-logic unit was prone to jamming. Zuse solved this problem by designing the Z3 to be a machine consisting completely of electromechanical relays [1]. Unfortunately, Zuse was working in wartime Berlin, Germany. Although Berlin thrived in the early years of the war, Zuse had trouble securing financial support from the Nazi government. Without substantial support, Zuse resorted to constructing his machines wherever he could, even in his parents’ living room [2].

Zuse was working as a civil engineer, and the Z1 and Z3 were meant to aid in his work. Often engineers needed to create mathematical tables and perform complex calculations, which were time-consuming and error-prone. Thus, Zuse created the Z1 to expedite the process [4,6]. Interestingly, Zuse’s motivation was not unique; in the United States, engineers were constructing machines, such as the ENIAC, to perform similar tasks. However, Zuse’s Z1 was operational in 1941, four years before the ENIAC was completed. Despite the Z1’s mechanical faults, Zuse retained the overall logical structure for his Z3 machine, with the addition of a square root function [5,6]. Indeed, in his memoirs, Zuse stated that the “basic circuits of the Z1 and the Z3 were equivalent” [4].

Unfortunately, the nature of war kept Zuse’s work obscure for many years. It was not until 1951 that his work became public [1]. Despite this, Zuse’s Z3 machine anticipated many of the components we see in modern computers, especially in the work of John von Neumann and his colleagues. Namely, the Z1 and Z3 employed floating-point numbers, binary representation, an addressable memory for data storage, “distinct decimal input and output units,” an arithmetic-logic unit “with a carry look-ahead adder,” and a control unit that incorporated a primitive two-stage pipeline [6]. In fact, Zuse built the Z1 and Z3 with the intent of using binary representation and having a memory separate from the control unit. Zuse independently developed these concepts years before von Neumann drafted his seminal EDVAC report. Indeed, the Z3 has been called the “first *programmable* calculating machine in the world” [4].

**The One and Only Data Type**

The Z3 has only one data type: floating-point numbers. Zuse actually refers to these numbers as “semi-logarithmic” [1]. Fixed-point data can be cumbersome and costly (the ENIAC is a notable example); floating-point data offers a certain amount of precision. What is interesting is that Zuse’s implementation anticipates the current IEEE-754 standard. In the Z3 machine, floating-point data is represented by 22 bits. The first bit is the sign bit, the next seven bits form the exponent (in binary two’s complement), and the remaining 14 bits form the normalized mantissa [5,6]. The range for the exponents, then, is from -64 to 64, and the normalized mantissa dictates that “the first digit before the decimal point . . . must always be a one” [4]. Thus, the mantissa effectively represented 15 bits of data. It is interesting to note, according to Raúl Rojas, a Zuse scholar at the Free University of Berlin, Donald Knuth attributes to Zuse the invention of normalized floating-point numbers [4].

Zuse faced a problem with his normalized floating-point representation: representing zero and infinity. Since zero cannot be expressed using a normalized mantissa, it has to be indicated by the exponent; thus, Zuse uses the exponent -64 to indicate a value of zero. Similarly, expressing infinity using normalized mantissas is problematic. Zuse solves this by using the exponent 63 to indicate an infinitely large value. Using Zuse’s conventions, the smallest representable number is 2-63, and the largest value representable number is 1.999 x 262 [4].

**The Memory Unit**

Considering Zuse was working on the Z3 during the early years of the Second World War, the resources needed to construct a large memory were scarce. Because of this scarcity, the memory of the Z3 was not meant to store programs internally. Instead, the Z3 consisted of a “binary memory unit” which could store 64 floating-point numbers (see figure 1) [5]. In addition, the Z3’s memory operations used the lower six bits to encode the address of a memory word; this affirms the memory unit’ maximum capacity of 64 words [4]. Using this information, we may infer that the Z3 is word-addressable, but there appears to be no evidence that Zuse actually used this terminology. Also, it appears that Zuse did not specify endianness for the Z3; the Z3 did not really have any need for endianness. At the very least, if we must specify endianness, big endian is the most relevant to the Z3.

Of particular interest, however, is the ingenuity with which Zuse was forced to work. According to Rojas, the Z3 processor employs 600 relays; the memory unit, on the other hand, employs three times as many relays. The scarcity of resources in wartime Germany “*forced* [Zuse] to think and rethink the logical structure of his machine” to make optimal use of his hardware resources [4]. Thus, the Z3’s memory unit is a product of the wartime rationing and clever thinking. Ultimately, the limited resources available to Zuse forced him to create a small memory unit not meant to store programs internally.

**The Registers**

From the programmer’s perspective, the Z3 has two floating-point registers, but the arithmetic-logic unit (ALU) has additional “invisible” registers. The main registers have no formal names, but we may use the convention established by Rojas: we will refer to these registers simply as “R1” and “R2.” These registers are primarily used for arithmetical operations [5]. Of importance is that the order of load operations affects program behavior. The first load operation loads a data-word of memory at the specified address into R1, and subsequent load operations load a data-word of memory at the specified address into R2. Arithmetical operations then implicitly use the contents of R1 and R2 as the operands. The resulting value is stored in R1, and the contents of R2 are cleared [5,6]. The register R2 also serves another purpose: it is a temporary store for the conversion of the “decimal input to a binary representation” [4].

The ALU is divided into two parts, *A* and *B*, each with its own set of registers. The registers *Af* and *Bf* store the exponent and mantissa, respectively, of what is essentially register R1. For our sake, we will use Rojas’ convention of referring to R1 as the register pair *[Af:Bf]*. Similarly, the register pair *[Ab:Bb]* performs the same function for register R2. A third register pair *[Aa:Ba]*, which is invisible to the programmer, contains the exponent and mantissa of a temporary floating-point register. What is more, the results of an addition or subtraction of exponents or mantissas are stored in special, “invisible” registers. The result of an operation performed on exponents is stored in *Ae*, and the result of an operation performed on mantissas is stored in *Be*. Thus, the register pair *[Ae:Be]* constitutes yet another register invisible to the programmer [4].

The Z3 has several registers, each with differing lengths and purposes (see table 1). Since Zuse did not name his registers, we have adopted Rojas’ naming conventions. Indeed, it is unlikely that Zuse would have needed to name his registers since only two of them, R1 and R2, are ever accessible to the programmer. What is more, the programmer cannot directly reference the registers by name in the arguments of instructions. Instead, the Z3’s instructions that rely on registers R1 and R2 implicitly use them in a particular order, as we will see later. A majority of the Z3’s registers, however, are dedicated to the ALU for computation; hence, these are neither explicitly named nor visible to the programmer. Indeed, according to Rojas, these registers need to be of variable length to adjust for precision in computation with floating-point numbers [4].

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table 1 The Z3’s Registers [4] | | | | |
| Register Name | Part *A* Register (ALU) | Size in Bits | Part *B* Register (ALU) | Size in Bits |
| R1 | *Af* | 7 | *Bf* | 17 |
| — | *Aa* | 8 | *Ba* | 19 |
| R2 | *Ab* | 8 | *Bb* | 18 |
| — | *Ae* | 8 | *Be* | 18 |

**The Instruction Set, the Instruction Format, and the One Addressing Mode**

The Z3’s instruction set is fairly limited by virtue of its punched tape medium. In fact, the Z3’s instruction set consists of only nine instructions (see table 2). Each instruction is coded by eight bits on each row of the tape, and memory operations use the lower six bits to encode memory addresses [5]. If we examine the instructions, we notice that Pr *z* and Ps *z* require the coding of an address. Notice that the bits of the address are made explicit. The first Pr *z* of the program loads the data word at address *z* into R1, and subsequent load operations load into R2. Arithmetical operations clear R2 and store the result in R1 [5,6]. However, the important insight is that the Z3 supports *only* direct addressing. There is neither an index register nor a base register, and the circuitry to load data does not consider immediate operands. What is more, the load operation does not specify a destination register; the destination is implied to be R1. Similarly, the store operation does not specify a source register; R1 is the implied source [6].

The Z3’s instructions fall into categories, do not specify operands, and the opcodes are of varying length (see table 2). The Z3’s primary operations are arithmetical operations. These operations all assume that R1 and R2 hold the operands for the operations; the result is always stored back into R1. Also, the instructions’ opcodes vary in length, from two to five bits [4]. For example, the opcode for Pr *z* is two bits (112), and the opcode for Ld is five bits (011112). Since there are nine instructions, we need to have at least four bits per operation. However, the inclusion of memory operations (memory addresses are six bits) increases the needed number of bits per operation. Since Zuse only uses two bits for the opcode of the memory operations, we only need eight bits. Also, since the programmer does not have to explicitly state operands (except in the case of Pr *z* and Ps *z*), there is no need to have longer instructions.

Of course, certain operations take more cycles to complete than others. For example, the number of cycles for Lu is variable depending on the size of the exponent of the input. The exponent dictates the number of multiplications needed to convert the value from decimal to binary representation. For example, an exponent of +8 would require 9 + 4x8 = 41 cycles to complete. Addition and subtraction, on the other hand, are much simpler operations. However, they require more than one cycle because “care has to be taken to set the size of the exponent of both [floating-point] arguments to the same value,” which necessitates additional comparisons and shifts [4]. Interestingly, we may estimate the Z3’s operating frequency by considering the number of cycles and estimated calculation times. Zuse claimed that multiplication took approximately three seconds to complete. Using this information, we may conclude that 16 cycles over three seconds approximates to 5.33 Hz [4].

Perhaps the store instruction (Ps *z*) is the most exciting. In the best case, a value may be stored into memory in zero cycles. In this case, the result of an arithmetical operation may be “redirected to the desired memory address” [4]. In this way, the cycles for the store instruction and the last cycle of the arithmetical operation overlap [4]. In the other case, storing a value to memory would require no more than one cycle to complete.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table 2 The Z3’s Complete Instruction Set [4] | | | | |
| Instruction Type | Instruction Name | Description | Opcode | Cycles to Complete |
| I/O | Lu  Ld | Read keyboard  Display result | 01 110000  01 111000 | 9 to 41  9 to 41 |
| Memory | Pr z  Ps z | Load address z  Store address z | 11 z6z5z4z3z2z1  10 z6z5z4z3z2z1 | 1  0 or 1 |
| Arithmetic | Lm  Li  Lw  Ls1  Ls2 | Multiplication  Division  Square root  Addition  Subtraction | 01 001000  01 010000  01 011000  01 100000  01 101000 | 16  18  20  3  4 or 5 |

For the sake of illustration, assume we wish to program the Z3 “to compute a polynomial using Horner’s method: *x*(*a*2 + *x*(*a*3 + *xa*4)) + *a*1” [4]. Let us assume that we have already stored the constant values *a*4, *a*3, *a*2, and *a*1 into the memory addresses 0001002, 0000112, 0000102, and 0000012, respectively. Also, let us assume that the value *x* is stored at address 0001012. We may use the following program (see table 3). However, we should note that the program would not be written using the instructions’ names; rather, it would be written in machine code.

|  |  |  |
| --- | --- | --- |
| Table 3 A Z3 Program to Compute a Polynomial using Horner’s Method [4] | | |
| Instruction | Actual Coding | Description |
| Pr 000100 | 11 000100 | Load *a*4 into R1 |
| Pr 000101 | 11 000101 | Load *x* into R2 |
| Lm | 01 001000 | Multiply R1 and R2, result into R1 |
| Pr 000011 | 11 000011 | Load *a*3 into R2 (recall subsequent loads load into R2) |
| Ls1 | 01 100000 | Add R1 and R2, result into R1 |
| Pr 000101 | 11 000101 | Load *x* into R2 |
| Lm | 01 001000 | Multiply R1 and R2, result into R1 |
| Pr 000010 | 11 000101 | Load *a*2 into R2 |
| Ls1 | 01 100000 | Add R1 and R2, result into R1 |
| Pr 000101 | 11 000101 | Load *x* into R2 |
| Lm | 01 001000 | Multiply R1 and R2, result into R1 |
| Pr 000001 | 11 000001 | Load *a*1 into R2 |
| Ls1 | 01 100000 | Add R1 and R2, result into R1 |
| Ld | 01 111000 | Display the result (value in R1) |

**The Control Unit** **and the Arithmetic-Logic Unit**

The Z3’s architecture separates the processor from the memory. The arithmetic-logic unit (ALU) is connected to the memory via a data bus, which transmits the exponents and mantissas of the floating-point data (see figure 1). The control unit contains microsequencers for each instruction; the control lines from the control unit ensure the proper synchronization between all of the Z3’s components. Zuse incorporated special hardware to handle zero and infinite values. Special hardware monitors the data that is loaded into the processor and trips exception flags as appropriate. Additionally, the Z3 handles underflow and overflow by monitoring the resulting exponent values of all arithmetical operations. Exceptions are also captured by “looking ahead” on the bus [4].

The ALU is divided into two parts that work with different parts of the floating-point numbers. The left side of the ALU, referred to as *A*, works with the numbers’ exponents, and the right side of the ALU, referred to as *B*, works with the numbers’ mantissas. Results of arithmetical operations are separated into their exponent and mantissa parts in the ALU’s invisible register pair *[Ae:Be]*. The exponent is stored in *Ae*, and the mantissa is stored in *Be*. Each side of the ALU also features switches that open or close the data bus. In addition, there are shifters for quick division and multiplication by powers of two; these shifters are particularly useful for addition and subtraction of floating-point numbers. The data path has the basic operations of addition and subtraction of exponents and mantissas. When the relay *As* is set, the second argument *Ab* is negated and sent to the ALU. When the relay *Bs* is set, the same is done for the argument *Bb*. Thus, when the relay *As* is set, part *A* of the ALU subtracts its arguments; otherwise, an addition is performed. The same is true for the relay *Bs* and part *B* of the ALU. Finally, the constant one is used to construct the two’s complement representation of results of arithmetical operations [4].

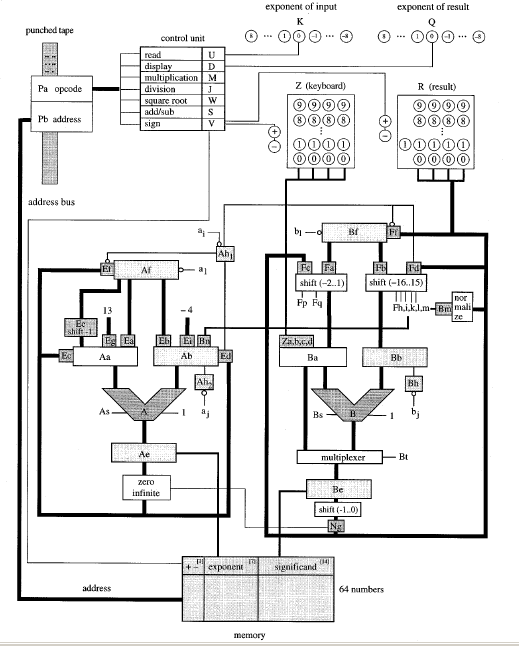
To ensure the correct operation is performed, the Z3 employs a “technique very similar to microprogramming” [4]. Once an opcode is read from the punched tape, it is decoded by the circuit *Pa*. If the opcode specifies a memory operation, the circuit *Pb* sets the address bus to the memory address specified by the opcode. Then, the control unit determines the microsequencing needed to carry out the instruction; special circuits are implemented for each operation in the instruction set. Once an instruction is decoded by the control unit, the arm of the control wheel moves and conducts electricity to activate the appropriate circuit. Rojas suggests that the control wheels provide a “comfortable platform for modifying the exact sequence of events during an operation” [4]. These control wheels correspond to microsequencers used in modern microprocessors; however, the Z3’s implementation is hardwired. Although the Z3 is not microprogrammed, its “extensive use of microsequencing” allowed for a great simplification of its circuitry [4].

In addition to its proto-microprogramming, the Z3 also implements a primitive form of two-stage pipeline. Each of the Z3’s cycles are divided into five stages: Stage I through V. Consider the typical fetch-decode-execute cycle implemented by most machines. Zuse carefully saved execution time by “overlapping the fetch stage of the next instruction with the write-back of the current one” [4]. Additions and subtractions are computed in parts *A* and *B* of the ALU during Stages I, II, and III. Stages IV and V are used to transport data through the machine [4]. In this sense, then, Zuse’s Z3 introduces one of the first instances of a very simple pipeline.

Zuse built the Z3’s adders to implement the carry look-ahead method. In normal binary addition, the mantissa requires 16 cycles to fully transmit the carry bits. Zuse solved this problem by performing addition and subtraction during Stages I, II, and III of a single cycle. Subtraction using two’s complement representation simply negates the second operand and performed an addition. A potential issue arises, however, when we wish to operate on zero or infinite values. Zuse solved this by performing these operations as normal and overriding the result with a “snooping circuit” [4]. Zuse also made sure to implement exception handling at the hardware level for undefined operations, such as 0/0 or ∞/∞. In such a case, the machine halts and illuminates the corresponding exception light on the output panel. If only one of the arguments is zero or infinite and the other argument is within bounds, then the Z3 always produces correct results. A final layer of exception handling detects underflow and overflow. Whenever an exponent value exceeds 63, overflow occurs—the value is infinitely large. Similarly, whenever an exponent is less than -64, underflow occurs—the value is zero [4].

The nature of the floating-point data necessarily complicates operations such as addition and subtraction. Particularly, to perform either addition or subtraction, the exponents of the numbers must be equivalent. If the exponents are equivalent, only the mantissas have to be added or subtracted. If the exponents differ, however, the mantissa of the smaller number is shifted to the right as necessary (with the exponent incremented appropriately) until the exponents were equivalent. The danger with this, of course, is that a value may be accidentally made zero after 17 shifts to the right. As would be expected, the signs of each number are compared before performing the addition or subtraction. If an addition or subtraction is requested for two values with the same sign, the requested operation is performed. However, if the signs differ, then the inverse operation is performed (e.g., subtraction instead of addition). Special circuitry then determines the appropriate sign for the resulting value [4].

Figure 1 The Z3’s Complete Architecture [4]



Addition and subtraction require only a few cycles; hence, these operations are implemented using relays, not control wheels. The arguments for addition and subtraction are assumed to be stored in registers R1 and R2. For addition, during the first cycle, the exponents are subtracted. During the second cycle, the mantissa with the larger exponent is loaded into register *Ba*, and the other mantissa is loaded into register *Bb*. The mantissa in *Bb* is shifted right as many times as necessary to equate the arguments’ exponents. During Stages I, II, and III of the second cycle, the arguments’ mantissas are added, and the processor tests if the result exceeds two. If the mantissa exceeds two, it is shifted once to the right, and the exponent is incremented once. For subtraction, up to five cycles are needed. The first two cycles are nearly identical to the addition cycles, except the mantissas are subtracted. The third cycle executes only when the mantissas’ difference is negative; the third cycle acts to make the resulting mantissa positive. The fourth cycle acts to normalize the resulting mantissa by shifting *Be* to the left as many times as necessary; the exponent is decremented as necessary. The full result is stored in R1, constituted by the register pair *[Af:Bf]*. The register *Af* holds the exponent, and the register *Bf* holds the mantissa [4].

Multiplication is implemented in the Z3 very similarly to how decimal multiplication is performed by hand. That is, multiplication is performed as a series of additions according to the multiplicand’s digits. Hence, multiplication takes 16 cycles to complete. The arguments’ exponents are added in the first cycle, and the result loops in part *A* of the ALU. Part *B* of the ALU handles the arguments’ mantissas. If the original arguments are within bounds, then the result is a number 1 ≤ *r* < 4. The final cycle corrects any values *r* that exceed two, similar to the addition algorithm [4].

The division algorithm is implemented similarly to the multiplication algorithm; obviously, division uses repetitive subtraction instead of addition. Unlike the multiplication algorithm, the division algorithm requires 18 cycles to complete. The result’s exponent is calculated simply by subtracting the exponents of the dividend and the divisor. However, dividing normalized mantissas is a little more challenging. If the dividend is greater than or equal to the divisor, the result is one. If the divisor exceeds the dividend, the result is zero. In the former case, the remainder is computed and recursively divided by the divisor. In the latter case, the remainder is the dividend; the recursive division caries on as in the previous case. Division of the mantissas yields a number ½ < *r* < 2; the seventeenth and eighteenth cycles check these conditions. If *r* < 1, the value is normalized by decrementing the exponent and shifting the mantissa once to the left [4].

Finally, we have the square root implementation, the “jewel in the crown of the Z3” [4]. A total of 20 cycles is needed to compute a number’s square root. Interestingly, the square root algorithm is used only to calculate the square root of numbers with even exponents. Odd exponents are decremented once, and the mantissas are shifted once to the left. All the bits in register *Bf* (the mantissa portion of R1) are used to compute the square root. The nineteenth cycle computes the final exponent, which is half the initial exponent. The square root result lies within bounds if the original argument (in R1) also lies within bounds [4].

**Input and Output**

While the Z3 reads its programs via punched tape, the user is able to interact via an input keyboard and an output lamp board. These I/O devices are connected to the processor via a data bus. To provide input or receive output, the system must be halted by executing the corresponding instructions [4,6]. The user is able to input mantissa values by using a decimal keyboard; the exponent is specified by selecting the appropriate key of a set labeled -8 through +8. The Z3 is limited to accepting values between 1 x 10-8 and 9,999 x 108. The Z3 also cannot print its output; instead, a lamp board is used to indicate output values. Data is outputted via an “array of lamps representing the digits from zero to nine” [4]. The output is limited to a maximum value of 19,999; the smallest value is 00001. The output’s exponent ranges from -8 to +8 [4].

Interestingly, the input and output instructions are the Z3’s most complex instructions. Because the input and output are in decimal, the input values have to be converted to binary for internal representation and reconverted to decimal for output. The decimal input has to undergo four iterations of storing and multiplication by 10. Each input digit is read sequentially and stored in bits 10, 11, 12, and 13 of register *Ba*. The number in *Ba* is then multiplied by 10; this is repeated until the number is fully entered. After the decimal value is fully entered, it is converted into binary representation. The exponent of the binary representation, according to Rojas, is formed “indirectly via shifts resulting from multiplication by 10” [4].

Handling the exponent is the challenging part. According to Rojas, if an exponent *e* is positive, the mantissa is multiplied by 10, *e*-times. If the exponent is negative, the mantissa is multiplied by 0.1, | *e* |-times. Multiplication by 10 is easily achieved by a few shifts. Indeed, it takes only four cycles for each multiplication; hence, it takes 32 cycles for the exponent +8. Thus, considering it takes a minimum of nine cycles to perform a read operation, it takes a maximum of 41 cycles to input a decimal value with exponent +8. For negative exponents, the multiplication by 0.1 is also achieved by using shifters and adders. However, the process is complicated since 0.1 is periodic in binary representation [4].

The output is realized by “multiplying or dividing iteratively by 10” [4]. For instance, if R1 contains a positive exponent, the number is multiplied by 0.1 until the binary exponent equals two and “the first left four bits of register *Bf* contain a number between zero and nine” [4]. This number is one of the decimal digits displayed on the lamp board. The value is then subtracted from the mantissa in *Bf*, and the process continues for the remaining digits. The process is similar for negative exponents; instead, the multiplication uses the constant 10 [4].

**Unusual Features (or Lack Thereof)**

The Z3’s most prominent unusual feature is that it read programs via punched tape. Of course, it was not unusual for early computing machines to read programs and data from punched cards. Like the early Harvard systems, the Z3 reads programs from punched tape and stores data in a separate memory. The Z3’s unusual choice of tape, however, arose out of the scarcity of resources in wartime Germany; in fact, Zuse relied specifically on 35mm movie film [6]. It was considered too expensive to construct a large memory to store both programs and data, so Zuse used external programs. Just like the punched cards of the Harvard-style machines, the Z3’s tape contained the opcodes for each instruction and memory addresses [4].

Somewhat related to the punched tape feature, the Z3 mysteriously lacked conditional branching in its instruction set. Although a loop cannot be written using instructions, a programmer may glue together the ends of the program tape to create a physical loop. This loop executes until a halting condition is reached. Interestingly, Rojas argues that the Z3 can simulate a Turing Machine with “a tape of limited size”; by extension, the Z3 could theoretically simulate any other computer [5]. Halting the loop is the problem with this infinite looping scheme. Thankfully, the loop may be easily halted by raising an arithmetical exception. In fact, Rojas notes that if Zuse had not included arithmetical exceptions, then we would not be able to stop the loop. Consequently, we would not be able to use this particular method of looping [5].

**A Casualty of War**

Zuse was a civil engineer, and his Z3 machine was tailored to suit his engineering needs. He completed the Z3 in the early years of the war [4,6]. Unfortunately, Zuse’s Z3 machine was another casualty of the war. Shortly after the original machine was completed, it was destroyed in the Allied bombings of Germany. However, Zuse worked from 1961 to 1962 to recreate the working model of the Z3 currently on display in the Deutches Museum in Germany [3]. In effect, the Z3 saw little to no practical application during its original, short lifetime. It would not be unreasonable to postulate that Zuse briefly used the machine for civil engineering applications before moving on to construct the Z4 machine. In its reincarnated form, at least, the Z3 serves as an educational tool for those interested in the early developments of German computing.

**Contributions to Computer Architecture**

Zuse’s original patent application from 1936 proves that he had already developed or anticipated the major developments of modern computing. In fact, Zuse did not limit his scope to sequential computers, such as von Neumann machines, and as early as 1936, he was already investigating array processing and the possibility of parallel processing. In particular, Zuse’s specification of an addressable memory separated from the processor is mirrored in von Neumann’s EDVAC draft. Although von Neumann and Zuse were working independently, Zuse was ahead in his computer development work. Unlike von Neumann, however, Zuse did not allow for instructions to be stored in memory and be randomly accessed, but he once claimed he considered storing instructions and data in memory together [1]. Despite working independently, Zuse’s work was both similar to and considerably different from the work of his American and British counterparts (see tables 4 and 5).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table 4 A Comparison of Architectural Features of Early Computers [4] | | | | | |
| Machine | Separate Memory and CPU? | Conditional Branching? | Soft or Hard Programming? | Self-modifying Programs? | Indirect Addressing? |
| Z3  ABC  Harvard  ENIAC  Manchester | Yes  Yes  No  No  Yes | No  No  No  Partially  Yes | Soft  Hard  Soft  Hard  Soft | No  No  No  No  Yes | No  No  No  No No |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table 5 A Comparison of Additional Architectural Features of Early Computers [4] | | | | | |
| Machine | Internal Coding | Fixed-point or Floating-point? | Bit-sequential Arithmetic? | Architecture | Hardware Technology |
| Z3  ABC  Harvard  ENIAC  Manchester | Binary  Binary  Decimal  Decimal  Binary | Floating  Fixed  Fixed  Fixed  Fixed | No  Yes  No  No  Yes | Sequential  Vectorized  Parallel  Dataflow  Sequential | Electromechanical  Electronic  Electromechanical  Electronic  Electronic |

Unfortunately for Zuse, no one seemed to take much interest in his pioneering work until 1951 [1,2]. Yet, in the abstract sense, Zuse’s Z3 is “equivalent to the computing model of today’s computers”; in the practical sense, however, the Z3 is woefully antiquated [5]. Ultimately, the Z3 was only one of several early developments in computing during the Second World War. Unlike the British and American efforts, however, Zuse’s work was not informed by other sources, and he lacked adequate financial and material support [4]. However, for his independent groundbreaking work, he has definitely secured his place in the history of computer development.

**Conclusion**

Konrad Zuse’s Z3 calculating machine is an important milestone in computer development. The Z3 independently introduced revolutionary concepts, such as a primitive pipeline and floating-point data representation. Unfortunately, Zuse and the Z3 were victims of their time; the Second World War greatly overshadowed Zuse’s contributions. Since Zuse was relatively unknown until 1951, it is understandable that he does not receive considerable recognition today. However, he deserves a great deal of attention for his contributions to the development of computers. Indeed, the Z3 marks a revolutionary anomaly in the history of computers where one man introduced and acted upon original concepts developed more fully by others at a much later date.

Works Cited

[1] Giloi, Wolfgang K. “Konrad Zuse: Reflections on the 80th Birthday of the German Computing Pioneer.” *ACM SIGNUM Newsletter* 33.2 (1998): 11-16. Web.

[2] Goodman, Seymour. “The Origins of Digital Computing in Europe.” *Communications of the ACM* 46.9 (2003): 21-25. Web.

[3] Petzold, Hartmut. “Computer Museum Series: Great Computing Museums of the World, Part Two: The Deutches Museum.” *Communications of the ACM* 53.5 (2010): 47-48. Web.

[4] Rojas, Raúl. “Konrad Zuse’s Legacy: The Architecture of the Z1 and Z3.” *IEEE Annals of the History of Computing* 19.2 (1997): 5-16. Web.

[5] ---. “Simulating Konrad Zuse’s Computers: A Working Java Simulation of the Z3.” *Dr. Dobb’s Journal* 25.9 (2000): 64-69. Web.

[6] Tarnoff, David L. “Using Early Instruction Sets to Introduce Computer Architecture.” *Journal of Computing Sciences in Colleges* 26.5 (2011): 80-87. Web.