Simulating Konrad Zuse's Computers

A working Java simulation of the Z3

Raul Rojas

Many programmers have probably never heard of Konrad Zuse, and American books on the history of computing make only peripheral references to his work. However, Zuse is popularly recognized in Germany as the "father of the computer," and his Z1, a programmable automaton built from 1936 to 1938, has been called the world's "first programmable calculating machine."

Konrad Zuse was born in Berlin in 1910 and died in December of 1995. He started thinking about computing machines when he was a civil engineering student. Zuse decided to build his first prototype exploiting two main ideas—that the machine would work with binary numbers and that the computing and control unit would be separated from the storage (this would be later called a "von Neumann architecture"). In 1936, Zuse completed the memory of the machine he had planned. It was a mechanical device, but not of the usual type. Instead of using gears (as Babbage had done in the previous century), Zuse implemented logical and arithmetical operations using sliding metallic bars. Figure 1 is a photograph of the reconstruction of the Z1 that can be seen today in Berlin's German Technology Museum.

In 1938, Zuse started building the Z3, a machine consisting purely of electromechanical relays but with the same logical structure as the Z1. It was ready and operational in 1941, four years before the ENIAC. In what follows, because Z1 and Z3 were practically equivalent from the logical and functional points of view, I refer only to the Z3.

Although functional copies of the Z3 were built in Berlin and Munich, the main work was done by Zuse himself, working, as he had done decades before, mostly from the top of his head.

My students and I decided to write a working Java simulation of the Z3. We obtained photocopies of sketches of the main circuits, which we validated using a CAD system. It required detective work in some instances, since we had to combine bits and pieces from documentation that Zuse had amassed over the years, but had never organized (see "Patentanmeldung Z-391," by K. Zuse, in *Die Rechenmaschinen von Konrad Zuse*, edited by R. Rojas, Springer-Verlag, 1998). With the help of several students, the circuits were validated over the course of two years. My student, Alexander Thurm, later wrote the Java version of the Z3 as soon as Java was first released.

In this article, I'll discuss the block architecture of the first computers built by Zuse, the Java simulation of the machines we implemented, and the surprising fact that Zuse's machines are universal, although they lack a branching instruction. The Java simulation and its source code are available over the Web at http://www.zib.de/zuse/.

Architectural Overview of the Z3

The Z3 is a floating-point machine. Whereas other early computing automatons worked with fixed-point numbers, Zuse decided early on to adopt what he called "semilogarithmic" notation, which corresponds to the modern floating-point representation.

Figure 2 is an overview of the main building blocks of the Z3 (with German labels). The Z3 consists of a binary mem-

Raul is a professor of computer science at Freie Universität Berlin. His main field of research is the theory and application of artificial neural networks. He is the author of Neural Networks *(Springer-Verlag, 1996). Raul can be contacted at rojas@inf.fu-berlin.de.*

(continued from page 64)
ory unit (capable of storing 64 floating-point numbers), a binary floating-point processor, a control unit, and I/O devices. The memory and the arithmetical unit are connected through a data bus, which transmits the exponent and mantissa of the floating-point representation. The control unit contains the microsequencers needed for each instruction. Control lines going from the control unit to the processor, the memory, and the I/O devices enforce the correct synchronization of all units. The tape reader provides the opcode of each instruction as well as the address for memory accesses. The input is done through a decimal keyboard; a result is shown in a decimal array of lamps.

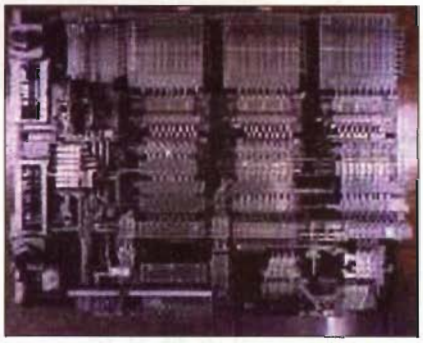

Figure 1: *The reconstructed Z1.*

The floating-point representation of the Z3 used 1 bit for the sign, 7 bits for the exponent (in two's complement coding), and 14 bits for the normalized mantissa. It was, in fact, somewhat similar to today's IEEE 754 Standard. The problem with normalized floating-point notation is that special conventions have to be used to deal with the number zero. The minimal exponent was used to code zero, the maximal, to code infinite numbers (for details, see "Konrad Zuse's Legacy: The Architecture of the Z1 and Z3," by R. Rojas, *IEEE Annals of the History of Computing*, 1997).

Instruction Set

The program for the Z3 is stored on punched tape. One instruction is coded using 8 bits for each row of the tape. The instruction set of the Z3 consists of the nine instructions in Table 1. Memory operations encode the address of a word in the lower 6 bits. The operating frequency of the Z3 was about 5 Hz.

Programming Model

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 I simply call "R1 and R2." These two registers contain the arguments of arithmetical operations. You can write any sequence of instructions, but you have to keep track of the state of the machine's registers.

The important point to remember is the following: The first load operation in a program transfers the contents of an address to R1. Any other subsequent load operation transfers a word from memory to R2. R2 is cleared after an arithmetical instruction, whereas the result is stored in R1.

The Processor

Figure 3 is a simplified representation of the arithmetical unit of the Z3. There are two parts: The left side is used for operations with the exponents of the floating-point numbers, the right side is used for operations with the mantissas. *Af* and *Bf* are registers used to store the exponent and mantissa of what, from the programmer's point of view, is register R1. I 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 programmers. The two arithmetic logical units (ALUs), *A* and *B*, are used to add or subtract exponents and mantissas, respectively. The result of the operation in the exponent part is put into *Ae*. In the mantissa part, the result of the operation is put into *Be*. The pair *[Ae:Be]* can be considered an internal register invisible to programmers. In part *B*, a multiplexer allows selection of *Ba* or the output of the ALU as the result of the operation. The multiplexer is controlled by a relay *Bt* (if *Bt* is equal to zero, 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. The structure of part *B* of the arithmetical unit is 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 displace the mantissa up to two positions to the right and one position to the left. This amounts to a division of *Bf* by 4 or a multiplication by 2. The second shifter can displace the mantissa in *Af* from 1 to 16 positions to the right and from 1 to 15 positions to the left. These shifts are needed for addition and subtraction of floating-point numbers. Multiplication and division with powers of two can therefore be performed when the operands for the next arithmetical operation are fetched and, in this sense, do not consume time.

The basic primitive operation of the datapath is the addition or subtraction of exponents or mantissas. When the relay *As* (*Bs*) is set, the negation of the second argument *Ab* (*Bb*) is fed into the ALU. Therefore, if the relay *As* 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.

Simulation of the Z3

Our reconstruction is a functional simulation of the Z3 in the sense that the numerical algorithms were actually implemented in the same way as in the original machine. We preserved the main structure of the machine and the direction of information flow in the code. The microcode used is therefore the same, though this is a software version of the machine.

Figure 4 is an overview of the simulation. The processor can be seen to the right. The numerical input keyboard is located to the upper left. The numerical display is in the lower left side of the image. The Z3 could be used as a desktop calculator. The buttons to the upper right show the operations that can be performed with the machine. After entering two numbers ("Einlesen" or Input), for example, they can be added or multiplied. The result can then be shown on the display ("Ausgeben" or Output).

There is another window (not shown here) where users can peek at the contents of the 64-word memory and can flip bits at will. This lets users change the memory contents easily to perform some experiments using the same punched tape.

The simulation was programmed using Java threads. There is a thread for most of the parallel operating components. The machine is always running, even when no operation has been started. In each cycle, the machine adds/subtracts the contents of its registers and discards the result if it is not needed. The actual machine was therefore loud. The simulation, when set in the tracking mode (which can be selected from a pulldown menu), shows the flow of information using an animated sequence of points.

The program for the Z3 is contained in the punch tape in Figure 3. When started in automatic mode, the program runs, the tape advances, and the registers are loaded and cleared as needed. The tape's

Type	Instruction	Description	Opcode
I/O	Lu	Read keyboard	01 110000
	Ld	Display result	01 111000
Memory	Pr z	Load address z	11 zzzzzz
	Ps z	Store address z	10 zzzzzz
Arithmetic	Lm	Multiplication	01 001000
	Li	Division	01 010000
	Lw	Square root	01 011000
	Ls1	Addition	01 100000
	Ls2	Subtraction	01 101000

Table 1: *Instruction set and opcodes of the Z1 and Z3.*

advance mechanism is a Java thread, in this case, that communicates with the control thread by sending the next instruction.

Since we started with the first version of Java, we went through all the problems that early developers had to tackle. There were no advanced development tools, and we found several inconsistencies between the UNIX and Windows Java browsers, mainly related to the way they display images. Sometimes we had to write code just to get around these problems. Animation was an especially difficult endeavor at the beginning, since there was no third-party software to help. The animation was done by having different sets of images that are switched as needed; for example, when a button is pressed.

However, in spite of all these problems, Java proved to be a reliable platform for our simulation. The simulation has been available for almost five years now, and the homepage has been visited by hundreds of interested individuals.

Simulating Conditional Branching

This article would not be complete without demonstrating a clever programming hack. The main defect of the Z3 was the absence of a conditional branch in the

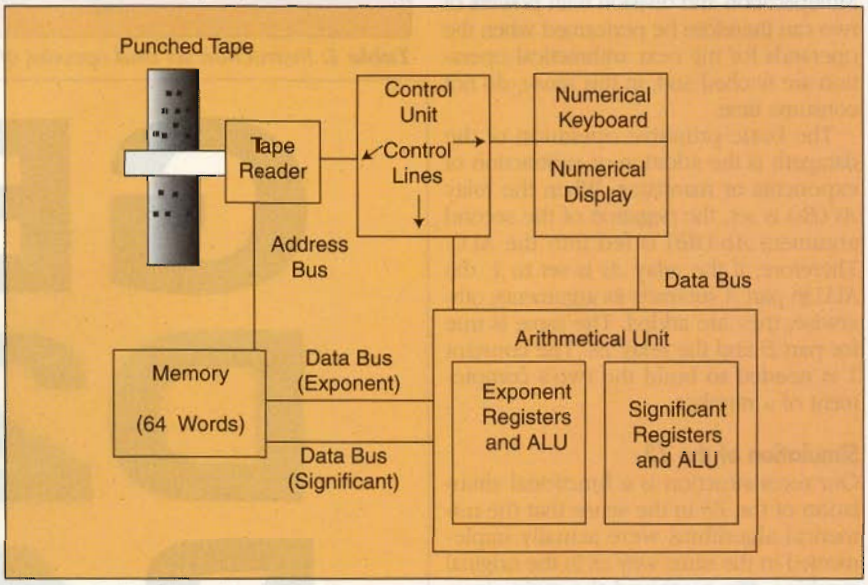

Figure 2: *Building blocks of the Z3.*

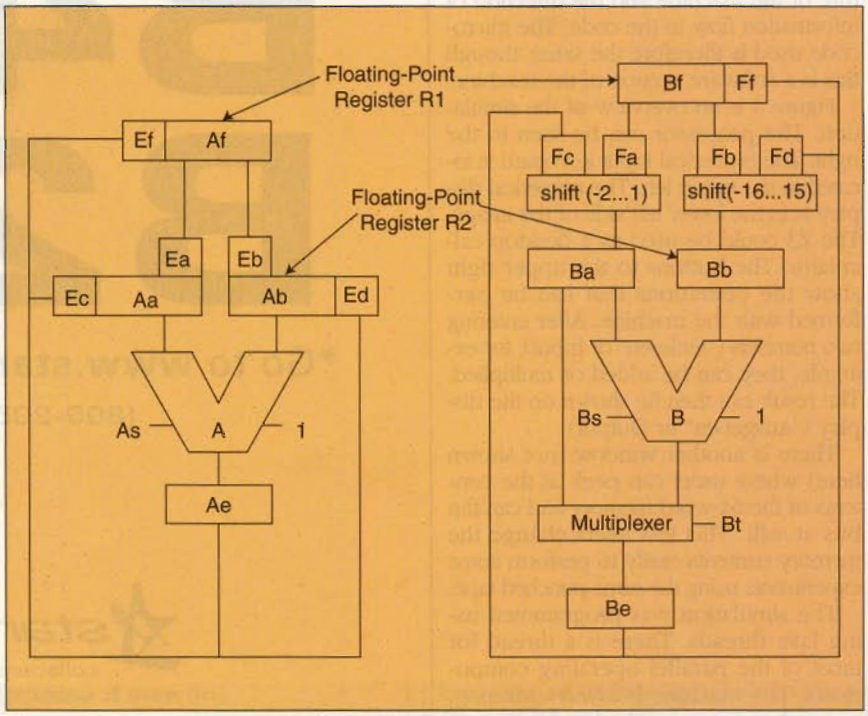

Figure 3: *The registers and datapath.*

instruction set. Nevertheless, it can be proved that a machine capable of executing a single loop and the basic arithmetic operations is equivalent to any computer with a limited addressing space.

The program loop can be obtained in the Z3 by just gluing together both ends of the punched tape. The loop will be performed repetitively until a halting condition is reached.

The Z3 can execute arithmetical expressions and store the results to memory, that is, expressions of the form $a := b\ op\ c$ can be compiled combining some primitive instructions (where *a*, *b*, and *c* denote memory locations and *op* is any of the basic arithmetic operations). We want to show that conditional branching can be simulated by using only this kind of expression in a program loop.

In any program containing branches, there are sections of code that are executed sequentially and terminated with a branch to another section. Let us enumerate these code sections using binary numbers. Without loss of generality, assume that there are 15 sections or less—we can then use 4 bits and label the sections as follows: 0001, 0010, ..., 1111. Our strategy is to jump from one section to another by storing the complement of the desired section number in the four memory locations. We can indicate that we desire to branch to section 3 (in binary 0011), for example, by setting $s3=1, s2=1, s1=0, s0=0$. Because we are executing a closed loop repetitively, the desired section of code will at some point arrive to the reading head. However, we must ensure that all other sections of code being read until the desired section appears (which are always being executed), do not store the results of their operations in memory. In this way, it does not matter how many operations are performed until the desired section is reached, since the state of the memory is not changed.

Implementing this idea requires putting a guard at the beginning of each code section. This is done using the auxiliary memory location *t* and computing at the beginning of each code section with the 4-bit binary label *abcd* the expression $t=((s3-a)(s2-b)(s1-c)(s0-d))^2$. Because this computation involves only basic arithmetical operations and fixed memory addresses, it can be performed by the Z3. Now, the variable *t* is zero if we are in the desired code section and one if not. We can therefore rewrite all expressions of the form $a=b\ op\ c$ as $a=at+(1-t)(b\ op\ c)$. If we are in the desired code section, memory location *a* is set to the new value $b\ op\ c$ (since $1-t=1$). If we are not, memory location *a* remains unchanged (since $1-t=0$).

Of course, we must take care of putting at the beginning of each section the code necessary for computing *t*, and we also take care of writing only programs that use expressions of the form given earlier. Each section of code is closed by a branch to another code section. Because the results of arithmetical operations can be used to set the values of the variables, all kinds of conditional branches can be executed. It can be proved that a Turing Machine with a tape of limited size can be simulated by the Z3 using this approach. For the details see "How to Make Zuse's Z3 a Universal Computer," by R. Rojas, *IEEE Annals of the History of Computing*, 1998. Thus, the Z3 can in fact simulate any other computer.

Zuse decided early on to adopt what he called "semilogarithmic" notation

Only one problem remains: Since the program loop is being executed repetitively, how do we stop the machine? This can done easily in the Z3 by causing an arithmetical exception. We can reserve a section of code as the stop section. When this section of code is called (by setting the locations to the appropriate section number), the auxiliary memory location *t* will be zero in this code section. The only operation that we include in this section is $0/t$. Whenever *t* is zero, the machine stops and signals the arithmetical exception 0/0. If *t* is not zero the machine just goes through this computation and proceeds to the next section. Had Zuse not included arithmetical exceptions in his Z3, we would not be able to stop the loop, and this whole approach would not work.

The result seems counterintuitive, until we realize that operations such as 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 magic of the transformation is making the hardware branchings visible to the programmer.

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. That's why I prefer to speak not of the "first computer" but of the "first computers" of the world, in plural, referring by this to the American, British, and German machines, which were all built almost simultaneously at the dawn of the computer age.

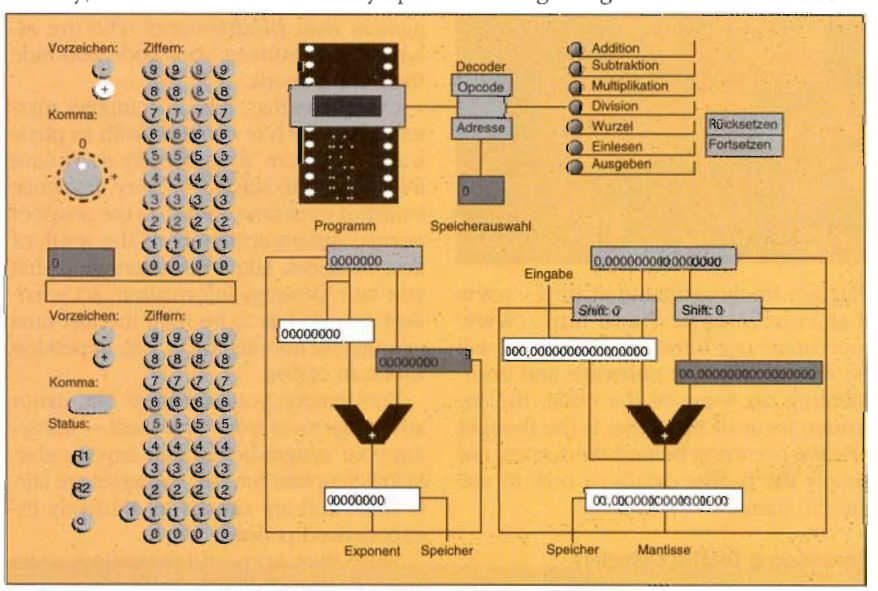

Figure 4: *Overview of the simulation screen.*

DDJ