How to Make Zuse's Z3 a Universal Computer

Raúl Rojas

January 14, 1998

Abstract

The computing machine Z3, built by Konrad Zuse between 1938 and 1941, could only execute fixed sequences of floating-point arithmetical operations (addition, subtraction, multiplication, division and square root) coded 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. In this paper we show that in fact a single program loop containing these arithmetical instructions can simulate any Turing machine whose tape is of a given finite size. This is done by simulating conditional branching and indirect addressing by purely arithmetical means. Zuse's Z3 is therefore, at least in principle, as universal as today's computers which have a bounded addressing space. A side-effect of this result is that the size of the program stored on punched tape increases enormously.

Universal Machines and Single Loops

Nobody has ever built a universal computer. The reason is that a universal computer consists, in theory, of a fixed processor and a memory of unbounded size. This is the case with Turing machines which have infinite 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 size can perform. If more storage is required, more can be added without having to modify the processor (provided that the extra memory is still addressable).

It is the purpose of this paper to show that Konrad Zuse's Z3, a computing automaton built in Berlin between 1938 and 1941, could *in principle* be programmed as any other modern computer. This is a rather curious result, since the Z3 can only compute sequences of arithmetical 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 glued together, the Z3 is a machine capable of repeatedly executing *a single loop* of arithmetical operations which act on numbers stored in memory.

It is well known that any computer program containing conditional branches and the usual instructions of imperative languages (Pascal for example) can be programmed using a single WHILE loop [1]. Also, all conditional branches can be eliminated from the loop [2]. I showed in [4] that if the Z3 is extended with indirect addressing it can simulate a Turing machine. We will adopt the techniques used in those papers in order to show that a Turing machine can be simulated by a single program loop of a machine capable of computing the four basic arithmetic operations.

Our computing model is the following: there exist memory locations which will be denoted by lower case letters. We can only refer explicitly to memory addresses (there is no indirect addressing). Initially (for the sake of simplicity) we will restrict our programs for the Z3 to a language containing only statements of the form

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

where op represents 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 refers implicitly 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 number is displayed on the console the machine stops). For more details on the architecture of the Z3 see [5] and [6].

Simulating Branches

We show here how to simulate the operation of a CASE statement using a technique introduced in [2] and used previously for the development of the theory of recursive functions [3]. 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 $P_1, \ldots, P_n$ and that the variable $z \in \{1, 2, \ldots, n\}$ is used to select the section which 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. In order to implement this idea we transform each section of code P_j in equivalent code P'_j according to the following recipe: At the beginning of each section P_j a comparison is made and if $z = j$ the auxiliary variable t is set to zero, otherwise it is set to one. The variable t can be interpreted as a flag for the "selected section" since it will be only zero in P_z. Now all original statements in the program $P_1, \ldots, P_n$ of the form $a = b \ \mathsf{op}\ c$ are transformed to

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

and are compiled accordingly. Therefore the state of variable a will not be modified unless the computation is performed 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 $P'_1, \ldots, P'_n$ 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 to be expected from a CASE statement.

We must only show now that it is in fact possible to perform the computation

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

where z and j are integers. The simplest approach is to use the binary representation of z, which is stored using the auxiliary variables $z_1, z_2, \ldots, z_m$. 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 section of code j the *complement* of the binary representation of j is stored in the variables $c_1^{(j)}, c_2^{(j)}, \ldots, c_m^{(j)}$. The following arithmetical calculation at the beginning of each code section j sets the variable t to its correct value:

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

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

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

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

Simulating a Turing Machine

A Turing machine (TM) is defined by a table of state transitions: given the current state Q and the tape symbol at the current position *pos* of the read/write head, we read from the table and find the new state Q', the symbol to be written o and the direction *dir* of motion of the read/write head ($+1$ or -1). The new position of the head is given by $pos = pos + dir$. Before simulating a Turing machine, the memory of the Z3 is prepared. All necessary tables are loaded at specific addresses as are the initial contents of the TM tape. All necessary auxiliary constants are also loaded.

It is clear that any Turing machine can be simulated using the following master loop:

- read tape symbol,
- look-up new state, output symbol, and direction of movement

- modify tape symbol,
- update state and position of read/write head.

The simulation can be done using table look-up. For example, reading the tape symbol amounts to the operation

$$s = \mathsf{memory}(tape, pos),$$

where "tape" is the initial address of the simulated tape and "pos" the current position. Only basic arithmetic is needed to compute the position of entries in a table: A table starting at address T can be accessed at position k by computing $T + k$ and using the result as an address. Thus, the only thing which is still lacking for the simulation of the Turing machine using the Z3 is indirect addressing, that is we want to use the results of arithmetic operations as addresses.

Assume that we want to implement the indirect addressing operation

$$a = \mathsf{memory}(x)$$

where x is the result of an arithmetic operation with integers and $n_a < x < n_b$. The integer constants n_a and n_b are the limits of the memory segment that we want to address indirectly.

We can implement the above operation using a CASE statement with one section for each integer between n_a and n_b. In each section i of the CASE statement we load address a with the contents of address i. Assume for example that $n_a = 10$ and $n_b = 20$. The code, before transforming it to work as a CASE statement, would be:

```
P_10:  LOAD 10
       STORE a
P_11:  LOAD 11
       STORE a
...
P_20:  LOAD 20
       STORE a
```

Now we apply a transformation similar to the one discussed above, using x as the CASE variable. The transformed program will select the contents of address x and will store it in address a, since only section P_x will modify the contents of a.

Note that the whole CASE statement contains one load statement for each consecutive memory address. We read all memory addresses between n_a and n_b, but we only keep the one we are interested in in a, namely address x. In the extreme case, when the indirect addressing operation refers to the whole memory, we would need to read all addresses in order to implement a single indirect addressing. But since the number of indirect memory references during one simulation cycle of the Turing machine is constant, the size of the program that we need is also constant (for a given memory size).

Using an entirely analogous approach we can store a number to the address represented by an arithmetical result x (indirect addressing in STORE operations). We can, for example, update the simulated tape of the TM using this approach.

It is clear that we have been helped here by the fact that the program is stored in a punched tape independent of the memory. The punched tape is allowed to be as large as necessary to read the sections of memory that we need to address indirectly (the state tables and the tape of the TM). The TM tables, of size N, are read once, the Turing Tape, of size M, is read once and updated once. We therefore need three transformed sections of code. If the CASE transformation expands each LOAD and STORE segment of code by a factor c, then we need at least a tape with $c(N + 2M)$ instructions in order to simulate the Turing machine. Since we choose to simulate a Universal Turing Machine, the size of the Turing tables is fixed once and for all. Given that the maximum size of the Turing tape is M, the size of the punched tape needed for our simulation program can be enormous, but is bounded.

This proves that with the computing model of the Z3 we can, in principle, do any computation that any other computer with a bounded memory can perform.

The Halting Problem

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

Whenever an undefined operation is performed, the Z3 stops and a lamp is set on the console. This is the case, for example, for the operation 0/0. Thus we define state $Q_0 = 0$ of the simulation as the "halting state" (for all other states Q_i is a positive integer) and the computation $0/Q$ is performed at the beginning of the master loop (Q is the current state). If the simulation reaches state Q_0 the machine stops.

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.

Conclusions

The main result shown in this paper 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 computers can compute. It could be argued that whenever we expand the memory (to accommodate more tape positions for a 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. This is undesirable. However, in real computers, there is also a limit for 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 paper 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 in making the hardware branchings visible to the programmer.

A possible criticism of the approach discussed in this paper 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 we 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. Also, the large loop of punched tape needed for the TM simulation program would pose extraordinary and most likely unsolvable mechanical difficulties.

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 for me from the study of Zuse's unpublished manuscripts (held in the archives of the Heinz-Nixdorf Museum in Paderborn) that after completing the Z3 he realized (between 1943 and 1945) that he could "lift" the decisions taken in hardware to the software level, so as 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, will be discussed elsewhere.

References

[1] D. Harel, "On Folk Theorems", *Communications of the ACM*, Vol. 23, N. 7, 1980, pp. 379–389.

[2] O. Ibarra, S. Moran, L.E. Rosier, "On the Control Power of Integer Division", *Theoretical Computer Science*, Vol. 24, 1983, pp. 35–52.

[3] R. Péter, *Recursive Functions*, Academic Press, New York, 1967.

[4] R. Rojas,"Conditional Branching is not Necessary for Universal Computation in von Neumann Computers", *Journal of Universal Computer Science*, Vol. 2, N. 11, 1996, pp. 756–767.

[5] R. Rojas, "Konrad Zuse's Legacy: the Architecture of the Z1 and Z3", *Annals of the History of Computing*, Vol. 19, N. 2, 1997, pp. 5–16.

[6] R. Rojas, *Die Rechenmaschinen von Konrad Zuse*, Springer-Verlag, Berlin, 1998.