Zuse's Z3 Square Root Algorithm
Talk given at Fall meeting of the Ohio Section of the MAA
October 1999 - College of Wooster
(Revised 02/02/2009)

Abstract

Brian J. Shelburne
Dept of Math and Comp Sci
Wittenberg University

In 1941 Konrad Zuse completed the Z3 sequence controlled calculator. The Z3 was a binary machine (numbers were stored internally in floating point binary) with a 64 by 22 bit memory which could perform the four standard arithmetic operations as well as take square roots. This talk examinations the algorithm Zuse used for his square root instruction and how it was implemented on the Z3.

Zuse's Z3 Square Root Algorithm- The Talk

Konrad Zuse (1910 - 1995)
born 1910 in Berlin, Germany
studied civil engineering at Technische Hochschule Berlin-Charlottenburg
worked as design engineer in aircraft industry
needed calculations for design and analysis of structures
his first attempt at automating calculation process was to design a form to track intermediate results of calculations
not familiar with design of mechanical calculators

His Early Machines: The Z1 - Z3

Z1: begun in 1936
floating-point binary representation
24 bit word length - 1 sign bit, 7 bit exponent, 16 bit mantissa (fraction)
  4 decimal digit precision
completely mechanical (memory, arithmetic unit and control)
16 word prototype memory completed 1937
controlled by tape (used discarded 35 mm film) - hand punched!
four functions : + - × ÷
mechanical arithmetic unit unreliable

Z2: 1938 - 1940 (smaller hybrid machine)
relay based ALU and control; mechanical memory
"proof of concept" for relay based calculator
demonstration led to funding for Z3 by Deutsche Versuchsanstalt fur Luftfahrt

Z3: 1939 - 1941 (relay based machine)
same design as Z1 except included square root operation
Memory: 64 by 22 bits - normalized floating point binary

± exponent fraction
a₆ ... a₀ b₋₁ ... b₋₁₄

Numeric Keyboard input: 4 columns of 10 keys for digits
1 row of exponent keys labeled: -8, -7 ... 7, 8

Display Output - row of 5 lights for fraction plus exponent lights labeled: -8, -7 ... 7, 8

Tape Control: nine 8 bit instructions

01 110000 - read keyboard 01 100000 - add
01 111000 - display result 01 101000 - subtract
01 001000 - multiply 01 010000 - divide
11 z₁z₂z₃z₄z₅z₆ - load address 01 011000 - square root
10 z₁z₂z₃z₄z₅z₆ - store address

Arithmetic Unit: separate ALU’s for exponents and fractions

Control Unit: instructions were implemented as a sequence of micro-operations which controlled movement of data through machine.
How Do You Take A Square Root?

1. Use a calculator
2. Use a slide rule
3. Use a table of logarithms
4. Use Newton's Iteration Formula to solve \( f(x) = x^2 - a \) by using the iteration formula
   \[ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{1}{2} \left( x_n - \frac{a}{x_n} \right) \]
   where \( x_0 \) equals some initial "guess" (\( x_0 = a/2 \) works well for \( a > 1 \)). Note this requires one division per iteration (expensive!).
5. Use the binomial expansion of \( \sqrt{1 - x} \)
   \[
   \sqrt{1 - x} = 1 - \frac{1}{2} x^1 - \sum_{k=2}^{\infty} \frac{1 \times 3 \times 5 \times \ldots \times (2k - 3)}{2^k \times k!} x^k = 1 - \frac{1}{2} x^1 - \frac{1}{2^2 \times 2!} x^2 - \frac{1}{2^3 \times 3!} x^3 - \frac{1}{2^4 \times 4!} x^4 - \ldots 
   
   We can calculate the \( \sqrt{a} \) by finding some value \( b^2 \) close to \( a \) and using the above series expansion
   for \( x = 1 - \frac{a}{b^2} \) since
   \[
   \sqrt{a} = \sqrt{b^2 + a - b^2} = b \sqrt{1 - \left(1 - \frac{a}{b^2}\right)}
   
   6. Complete the Square: This technique is based on the simple idea that if \( x \) is a "good" approximation to \( \sqrt{a} \) where \( a - x^2 = e > 0 \) is the error, then find a small value \( b \) such that
   \( (x + b) \) is a better approximation; that is \( a - x^2 > a - (x + b)^2 = (a - x^2) - 2bx - b^2 \geq 0 \) or to put another way, find a value \( b \) such that \( e - b(2x + b) \geq 0 \). The inequality is not linear in \( b \) but is it
   not difficult to questimate a value. Generally this is done digit by digit.

Completing the Square (details)

Without loss of generality assume \( 1 \leq a < 100 \) and suppose \( x = Q_{k-1} \) is an approximation to \( \sqrt{a} \) accurate to \( 10^{-k+1} \) (i.e. the \( k - 1^{\text{st}} \) digit below the decimal point). That is

\[
Q_{k-1} = d_0 + \sum_{i=1}^{k-1} d_i \times 10^{-i} \quad \text{where} \quad d_i \text{ is a digit between 0 and 9}
\]
Letting $b = d_k 10^{-k}$, find the largest digit $d_k$ such that

$$a - (x + b)^2 = a - (Q_{k-1} + d_k 10^{-k})^2 = \left(a - Q_{k-1}^2\right) - 2Q_{k-1} (d_k 10^{-k}) - (d_k 10^{-k})^2 \geq 0$$

To find the largest such digit $d_k$ factor out $d_k 10^{-k}$ to obtain

$$\left(a - Q_{k-1}^2\right) - d_k 10^{-k} \left(2Q_k + d_k 10^{-k}\right) \geq 0$$

Scaling (multiply) by $10^{2k}$ does not change the sign of the expression but it makes all values integers.

$$10^{2k} \left(a - Q_{k-1}^2\right) - d_k \times \left(2Q_k 10^k + d_k\right) \geq 0$$

To find the largest digit $d_k$ use the following guess and check method.

1. Guess the largest digit $d_k$ such that $d_k \times 2Q_{k-1} \times 10^k$ is less than $10^{2k} e = 10^{2k} \left(a - Q_{k-1}^2\right)$

2. Check that $d_k \times \left(2Q_{k-1} \times 10^k + d_k\right)$ is still less than the “scaled error”. If not reduce by 1 and try again.

If successful then $Q_k = d_0 + \sum_{i=1}^{k} d_i 10^{-i}$ is a better approximation with a smaller “scaled error” now equal to

$$10^{2k} \left(a - Q_{k-1}^2\right) - d_k \times \left(2Q_{k-1} \times 10^k + d_k\right) = 10^{2k} \left(a - Q_k^2\right) < 10^{2k} \left(a - Q_{k-1}^2\right)$$
Finding the largest $d_k$ which maintains this inequality is the theory behind the following well-known algorithm (well-known before the advent of cheap calculators) used to compute a square root by hand. To demonstrate we take the square root of 20.375.

\[
\begin{array}{cccc}
d_0 & d_1 & d_2 & d_3 \\
4. & 5 & 1 & 3 \\
\sqrt{20.375} 00 \leftarrow a \\
16 & 00 \leftarrow 10^2 \times Q_0^2 \\
\hline
4 \times 20 + d_1 \rightarrow 85 & 4 & 37 & \leftarrow 10^2 \times (a - Q_0^2) \\
4 & 25 & \leftarrow d_1 \times (2Q_0 \times 10 + d_1) \\
\hline
45 \times 20 + d_2 \rightarrow 901 & 12 & 50 & \leftarrow 10^4 \times (a - Q_1^2) \\
9 & 01 & \leftarrow d_2 \times (2Q_1 \times 10^2 + d_2) \\
\hline
3 & 49 & 00 & \leftarrow 10^6 \times (a - Q_2^2) \\
\hline
451 \times 20 + d_3 \rightarrow 9023 & 2 & 70 & 69 & \leftarrow d_3 \times (2Q_2 \times 10^3 + d_3) \\
\hline
78 & 31 & 00
\end{array}
\]

First - find the largest integer $d_0 = Q_0$ whose square is less than or equal to the integer part of the number $a$. Square $Q_0$ and subtract it from the integer part. Bring down the next two digits.

Second. - Find the largest integer $d_1$ such that $2 \times Q_0 \times 10 + d_1$ times $d_1$ is less than the previous remainder. Subtract it and bring down the next two digits. $Q_1 = Q_0 + d_110^{-1}$

Third - Find the largest integer $d_2$ such that $2 \times Q_1 \times 10^2 + d_2$ times $d_2$ is less than the previous remainder. Subtract it and bring down the next two integers. $Q_2 = Q_1 + d_210^{-2}$

$k^{th}$ Step - Find the largest integer $d_k$ such that $2 \times Q_{k-1} \times 10^k + d_k$ times $d_k$ is less than the previous remainder. Subtract it and bring down the next two integers. $Q_k = Q_{k-1} + d_k10^{-(k-1)}$ Repeat.
The Binary Version of the "Completing the Square" Square Root Algorithm

The same algorithm can be easily done in binary. By way of demonstration, we find the square root of the binary value 10100.011 which is 20.375 decimal. The floating point representation is 1.0100011 x 2^{10} (note the even exponent). Therefore the square root of 10100.011 is the square root of 1.0100011 times 2^{10} (or 4).

Remember that multiplication by 2 in binary is equivalent to a left shift. Furthermore scaling by powers of 10 (binary) is scaling by powers of 2.

\[
\begin{array}{cccccccccc}
1 & . & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\sqrt{1} & . & 01 & 00 & 01 & 10 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
1 \\
- \\
0 & 01 & <- & 2^2 \times (a - Q_0^2) \\
100 & 00 & <- & b_1 \times (2 \times Q_0 \times 2^1 + b_1) \\
--------- \\
1 & 00 & <- & 2^4 \times (a - Q_1^2) \\
1000 & 00 & <- & b_2 \times (2 \times Q_1 \times 2^2 + b_2) \\
----- \\
1 & 00 & 01 & <- & \text{etc...} \\
10001 & 1 & 00 & 01 \\
--------- \\
0 & 10 \\
100100 & 0 & 00 \\
----- \\
10 & 00 \\
1001000 & 00 & 00 \\
----- \\
10 & 00 & 00 \\
10010000 & 00 & 00 & 00 \\
------------- \\
10 & 00 & 00 & 00 \\
100100000 & 00 & 00 & 00 & 00 \\
-------------- \\
10 & 00 & 00 & 00 & 00 \\
1001000000 & 00 & 00 & 00 & 00 & 00 \\
------------ \\
10 & 00 & 00 & 00 & 00 & 00 \\
10010000001 & 1 & 00 & 10 & 00 & 00 & 01 \\
------------- \\
11 & 01 & 11 & 11 & 11 & 00 \\
10010000011 & 10 & 01 & 00 & 00 & 11 & \text{whew !!!} \\
\end{array}
\]

So therefore the square root of 10100.011 is approximately 1.0010000011 x 2^{10} = 100.10000011 = 4.51171875 which is "close" to 4.513867521
The Z3 Square Root Implementation - Completing the Square in Binary

The Z3 used a variant of the above algorithm to compute a square root. Given \( 1 \leq a < 100_2 \) (values are in binary) if \( Q_{k-1} \) is an approximation to \( \sqrt{a} \) accurate to \( 2^{-(k-1)} \), that is

\[
Q_{k-1} = b_0 + \sum_{i=1}^{k-1} b_i 2^{-i}
\]

where each \( b_i \) is either 0 or 1, then find the largest binary digit \( b_k \) (0 or 1) such that

\[
a - \left(Q_{k-1} + b_k 2^{-k}\right)^2 = \left(a - Q_{k-1}^2\right) - 2 \times Q_{k-1} \times (b_k 2^{-k}) - (b_k 2^{-k})^2 \geq 0
\]

Because there are only two possible choices for \( b_k \), either 0 or 1, this algorithm is easy to implement. Moreover since \( b_k^2 = b_k \) these are some “tricks” to make iteration efficient. Begin by scaling (multiplying) the above inequality by \( 2^k \) (instead of \( 2^{2k} \)). If we have

\[
2^k \times \left(a - Q_{k-1}^2\right) - (2 \times Q_{k-1} + 2^{-k}) \geq 0
\]

then \( b_k = 1 \); otherwise \( b_k = 0 \). In any case since \( b_k^2 = b_k \) it follows that the above expression \( 2^k \times \left(a - Q_{k-1}^2\right) - (2 \times Q_{k-1} + 2^{-k}) \) equals \( 2^k \left(a - \left(Q_{k-1} - b_k 2^{-k}\right)^2\right) = 2^k \left(a - Q_k^2\right) \) which can be used in the calculation for the next step. If we multiply \( 2^k \left(a - Q_k^2\right) \) by 2 we obtain

\[
2^{k+1} \left(a - Q_k^2\right)
\]

for use in the next iteration step. Multiplication by 2 is easily done on a binary machine using a left shift.

The algorithm is given below. Assume \( 1 < a \leq 100_2 \) and let \( e_k = 2^k \left(a - Q_k^2\right) \) (a scaled error value)

where \( e_0 = (a-1) \).

1. let \( Q[0] = 1 \)
2. let \( e[0] = a - 1 \)
3. for \( k = 1, 2, 3, \ldots \)
   3.1 Compute \( 2 \times e[k-1] - (2 \times Q[k-1] + 2^k) \)
   3.2 If positive or zero (i.e. \( b[k] = 1 \))
      3.2.1 \( Q[k] = Q[k-1] + 2^k \)
      3.2.1 \( e[k] = 2 \times e[k-1] - (2 \times Q[k-1] + 2^k) \)
   3.3 Otherwise (i.e. \( b[k] = 0 \))
      3.3.1 \( Q[k] = Q[k-1] \)
      3.3.2 \( e[k] = 2 \times e[k-1] \)
Example - Compute the square root of 1.0100011

<table>
<thead>
<tr>
<th>k</th>
<th>2xe[k-1] - (2 × Q[k-1] + 2^k)</th>
<th>e[k]= a – Q[k]^2</th>
<th>Q[k]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0.0100011</td>
<td>1.</td>
</tr>
<tr>
<td>1</td>
<td>0.100011 - 10.1</td>
<td>negative</td>
<td>0.100011</td>
</tr>
<tr>
<td>2</td>
<td>1.00011 - 10.01</td>
<td>negative</td>
<td>1.00011</td>
</tr>
<tr>
<td>3</td>
<td>10.0011 - 10.001</td>
<td>positive</td>
<td>0.0001</td>
</tr>
<tr>
<td>4</td>
<td>0.001 - 10.0101</td>
<td>negative</td>
<td>0.001</td>
</tr>
<tr>
<td>5</td>
<td>0.01 - 10.01001</td>
<td>negative</td>
<td>0.01</td>
</tr>
<tr>
<td>6</td>
<td>0.1 - 10.010000</td>
<td>negative</td>
<td>0.1</td>
</tr>
<tr>
<td>7</td>
<td>1.0 - 10.0100001</td>
<td>negative</td>
<td>1.0</td>
</tr>
<tr>
<td>8</td>
<td>10.0 - 10.01000001</td>
<td>negative</td>
<td>10.0</td>
</tr>
<tr>
<td>9</td>
<td>100.0 - 10.010000001</td>
<td>positive</td>
<td>1.101111111</td>
</tr>
<tr>
<td>10</td>
<td>11.011111111 - 10.010000011</td>
<td>positive</td>
<td>1.001111011</td>
</tr>
</tbody>
</table>

This agrees with the example given above.
A Closer Look at the Architecture of the Z3 Arithmetic Unit

Z3 had separate ALU's to handle the exponent and fractional parts of a number.

Basic data path of the fraction ALU consisted of four registers (Bf, Ba, Bb, and Be) an addition/subtraction unit and two shifter units connected by data lines. A number of gates controlled the flow of data around the data path.

Hardware operations included addition and subtraction, pre-shifting input to the Ba and Bb registers, detecting a positive or zero result and setting individual bits in a register.

A micro-operation consisted of a timed sequence of gates openings that allowed the contents of registers to flow through the pre-shifters and addition/subtraction unit. Detecting a positive or zero result could conditionally alter the flow of data.

Instructions were implemented as sequences of micro-operations.


Micro-operation Implementation of Z3 Square Root

Note Bf[-k] refers to bit -k in register Bf

0.   Bf := a  // Load Bf with a
     Ba := Bf (or 2 × Bf if exponent is odd)  // Ba = a
     Bb[0] := 1  // Q₀ = 1

1.   if Ba - Bb ≥ 0 then  // test
     Be := Ba - Bb
     Bf[0] := 1  // set bit
     else
     Be := Ba  // set up next comparison
     Ba = 2 × Be  // 2 × r₀
     Bb = 2 × Bf;  Bb[-1] := 1  // 2 × Q₀ + 2⁻¹

2   if Ba - Bb ≥ 0 then  // test
     Be := Ba - Bb
     Bf[-1] := 1  // set bit
     else
     Be := Ba  // set up next comparison
     Ba := 2 × Be  // 2 × r₁
     Bb := 2 × Bf;  Bb[-2] := 1  // 2 × Q₁ + 2⁻²

and in general

if Ba - Bb ≥ 0 then  // if 2rk-1 - (2 × Qk-1 + 2ᵏ) ≥ 0 then
     Be := Ba - Bb  // r_k = 2 × r_k-1 - (2×Q_k-1 + 2ᵏ)
     Bf[-k] := 1  // Q_k = Q_k-1 + 2ᵏ
     else
     Be := Ba  // r_k = 2 × r_k-1
     // Q_k = Q_k-1
     // set up next comparison
     Ba := 2 × Be  // 2×r_k
     Bb := 2 × Bf;  Bb[-(k+1)] := 1  // 2 × Q_k + 2⁻(k+1)

At the end the exponent which was even was divided by 2 (a right shift) yielding the square root.

The thing to notice was the economy of operations needed to compute a square root - shifts, subtractions, comparisons and setting individual bits within a word.
Summary

1. The square root algorithm used by the Z3 was the binary analog of the "completing the square" algorithm, a widely known (?) manual technique for obtaining a square root one digit at a time.

2. For $1 \leq a < 100$ and $Q_{k-1}$ an approximation of $\sqrt{a}$ accurate to $k^{-(k-1)}$, set bit $b_k$ equal to 1 if

$$a - \left( Q_{k-1} + b_k 2^{-k} \right)^2 \geq 0$$

otherwise set bit $b_k$ equal to 0. It was shown that this is the same as checking if

$$2^k \left( a - Q_{k-1}^2 \right) - \left( 2Q_{k-1} + 2^{-k} \right) \geq 0$$

and leads to the iteration algorithm

```
Repeat until tired
Compute 2 × e[k-1] - (2 × Q[k-1] + 2⁻ᵏ )
If positive or zero
    Q[k] = Q[k-1] + 2⁻ᵏ
    e[k] = 2 × e[k-1] - (2 × Q[k-1] + 2⁻ᵏ)
Otherwise
    Q[k] = Q[k-1]
    e[k] = 2 × e[k-1]
```

where $e[k] = (a - Q[k]^2)$ and initial condition $Q_0 = 1$ and $e[0] = (a - 1)$

3. A close examination of this algorithm reveals that it requires only the operations of subtraction, left shift (multiplication by 2), set bit, and test if positive or zero. - all of which are easy to implement in binary logic.

4. The Z3 had a data path that could easily implement these operations and a micro-operation sequencer that could iterate each step of the algorithm the requisite number of times. Use of floating point binary notation helped.

5. Binary representation is used in computers because for economic reasons computer hardware is built out of bi-stable components. However, many complicated arithmetic operations like multiplication, division and square root are easy to implement in binary. Even though we humans find binary notation complicated to use (even moderately sized numbers take a lot of bits), because of the ease in implementing complicated algorithms in binary, there is something to be said for the advantages of binary notation in and of itself.

Any sufficiently advanced civilization uses a number system based on a power of two.
Epilog: A Follow Up on the Z3 - the Z4

The Z3 was really a "proof of concept" for the more ambitious Z4, a 32 bit machine begun after the Z3 was completed. Unlike the Z3, the Z4 used a mechanical memory of 1000 words (which took up a cubic meter of space!). Before it was completed it was moved to Gottingen to avoid the Berlin air raids which had destroyed the Z3.

Just before Gottingen was "liberated" at the end of WWII, the Z4 was moved to Hinterstein, Bavaria. In 1950 the refurbished Z4 with some additional features that gave it a conditional jump was moved to ETH Zurich. In 1955 it was moved to Basel where it gave "acceptable service" until 1960.

After the war Zuse founded Zuse Kommandit Gesellschaft (Zuse KG) which produced for the Leitz Optical Company a relay based, punch-tape controlled "computer" called the Z5 which was based on the Z4 design. This led to the production of a number of relay-based calculators, called the Z11. In 1956 Zuse began work on a vacuum-tube based computer called the Z22 with a transistor version following. A series of mergers and takeovers finally lead to Zuse KG becoming part of Siemens.

References

