A FAST FLOATING-POINT SQUARE-ROOTING ROUTINE FOR THE 8080/8085 MICROPROCESSORS

VALENTIN CHAMRÁD

A speed-oriented implementation of the Newton-Raphson algorithm is described, reducing the worst-case execution time to the level of standard floating-point multiplication and thus supporting a wider use of square-root filters in microprocessor-based self-tuning controllers.

1. INTRODUCTION

Square-root is a function for which numerous numerical methods have been developed. In most math packages for microprocessors, simple iterative methods have been used, as no special demands for speed — nor even for accuracy in some cases — are expected: e.g. in [1] and [2] the execution time of square-rooting is approx. 2.5 times longer than that of multiplication, and in [3] a quintuple error limit compared with other operations is accepted. Some floating-point packages do not support this function at all leaving its evaluation to user defined programs (e.g. [4]).

In the floating-point subroutine package for the Intel 8080/8085 microprocessors [6] developed in the Institute of Information Theory and Automation in Prague, the speed of operation has been strongly emphasized; and as a frequent use in software for self-tuning controllers with square-root filters has been expected, there was a special demand for a fast square-rooting subroutine with the same accuracy ($\pm 1$ LSB) as with all the other operations. It took a considerable effort to match this condition, and every promising method of accelerating the calculation was tested — even empirical or intuitive; special testing programs were developed for this purpose, checking the real deviation of the square-root returned by the subroutine under test for all the 32k significantly different input values and printing those values yielding results with an error exceeding a preset limit of 0.5, 0.75 or 1 LSB only.

The abbreviations MS and LS will be used for most significant and least significant respectively in this paper; in connection with them, B will be reserved for bit, while byte will not be abbreviated.
2. NUMBER FORMAT

The first important step to improve the performance of a floating-point subroutine is to realize some special properties of the number format used; in our case, this is defined for every representable number as

\[ x = a \cdot 2^b, \quad 0.5 \leq |a| < 1 \]  

where \( a \) (mantissa) is a 16 bit FRACTION number in two's complement form, and \( b \) (exponent) is a 7 bit INTEGER in the "excess-64" code, i.e. with an added offset of 64 to avoid negative values, so that the real value stored in the exponent byte is

\[ b' = b + 64, \quad 0 \leq b' \leq 127 \]

which enables us to use the MSB for overflow/underflow detection. As explained in [6], the precision of \( 0.0035\% \) and the range of representable numbers approx. \( \pm 3 \cdot 10^{-19} \) to \( \pm 10^{+19} \) proved to be quite sufficient for most engineering applications; on the other hand, the achievable execution speed of arithmetic subroutines is much higher compared to longer formats due to the possibility of using register instructions only for most operations.

With respect to square-rooting, the first important thing to realize is that we deal with a product of two numbers, the second of them being a power of two; the operation thus can be simplified by a conversion — may be fictive only — to an unnormalized format with the next higher even exponent, which can be square-rooted by an integer division by 2. If we denote the input operand \( x \) and the result \( y \), then

\[ b_y = \text{INT} \left[ \frac{1}{3}(b_x + 1) \right] \]

where \( \text{INT} \) denotes integer part of the expression in square brackets. In the format used, the result exponent will be computed as

\[ b'_y = \text{INT} \left[ \frac{1}{4}(b'_x + 65) \right] \]

Division by the shift right (RAR) instruction yields the Carry bit (LSB of the sum in parentheses) representing the directive for denormalizing the mantissa; we shall see later that a different treatment of mantissa instead of real denormalization will be more useful. A simple analysis of the limit values of \( b'_y \) shows that neither overflow nor underflow can occur; no final testing of exponent will therefore be needed.

3. THE ALGORITHM AND ITS CODING

The square-rooting algorithm proper will then operate on numbers in the range

\[ 0.25 \leq a_x < 1 \]
only; the results shall lie within the range of

\[ 0.5 \leq a_y < 1 \]

and that means that they will be automatically normalized; consequently, no final normalization will be needed.

Halving of the result range, together with the same resolution of 15 bits for both the input and result values and with the nonlinearity of the function, causes that we shall get the same results for two or even three adjacent input values, which should not be considered erroneous.

For square-rooting of mantissa, we have adopted the Newton-Raphson iteration formula, used in [2] and [3] as well, which in the \( i \)th iteration computes the new approximation \( y_{i+1} \) as

\[ y_{i+1} = \frac{1}{2} \left( y_i + \frac{x}{y_i} \right) \]

i.e. as the mean value of the old approximation \( y_i \) and the quotient of the input value and the old approximation. For the known deviation of the \( i \)th approximation

\[ \Delta y_i = y - y_i \]

where \( y \) is the correct value of the square root, the deviation of the next step can be estimated as

\[ \Delta y_{i+1} = \frac{\Delta y_i^2}{2(y - \Delta y_i)} \]

As a rule, in conventional computers the iteration cycle starts for simplicity with \( y_0 = x \), and the iteration process is stopped when the difference between two successive values of \( y_i \) is lower than the accuracy required. A similar method — used in our testing programs — determines the accuracy of the estimate using the difference between \( y_i \) and the quotient computed when evaluating formula (5); using (6), this difference \( d_i \) equals

\[ d_i = y_i - \frac{x}{y_i} = y + \Delta y_i - \left( \frac{x}{y + \Delta y_i} + \Delta q_i \right) \]

where \( \Delta q_i \) is the truncation error of the quotient. For \( \Delta y_i \ll y_i \) we can approximate

\[ d_i = \frac{\Delta y_i(2y + \Delta y_i) - \Delta q_i(y + \Delta y_i)}{y + \Delta y_i} \approx 2 \Delta y_i - \Delta q_i \]

and if we assume \( \Delta q_i \) to be small enough (which is satisfied by extended precision in our test programs), we can take

\[ \Delta y_i = \frac{1}{2} d_i \]
Note that the last but one approximation is tested here, so that an accuracy exceeding the precision of the format used can theoretically be achieved with the final result.

The possibilities of reducing the overall execution time of this iterative process comprise both reducing the execution time of a single iteration cycle and reducing the total number of iterations. For the latter way, the only means of reduction is a better original estimate, which could be constructed simply enough. Practically the choice is limited to a linear function, as any more complicated function (e.g. polynomial) would consume more time than another iteration cycle. Let us remind that this estimate should be constructed using a still normalized mantissa and the Carry bit representing an even or odd exponent; in other words, the Carry bit tells us whether the mantissa belongs to the “lower octave” of operands described by

\[ Carry = 0 \], \quad 0.25 \leq x_L < 0.5, \quad x_L = 0.5a_L \]

or to the “higher octave” with

\[ Carry = 1 \], \quad 0.5 \leq x_H < 1, \quad x_H = a_H \]

We found advantageous to choose different estimates for each octave: in fact, we used the results of the first iteration, taking the known correct values in the end points of interval (3) as primitive estimates, but we used a direct method to construct them.

For the upper octave, we used \( y_{0H} = x_H \) (correct for \( x = 1 \)), and from (5) and (10) we obtained

\[ y_{1H} = 0.5 \left( x_H + \frac{x_H}{x_H} \right) = 0.5a_H + 0.5 \]

Similarly, we used \( y_{0L} = 2x_L \) (correct for \( x = 0.25 \)) for the lower octave and obtained from (5) and (9)

\[ y_{1L} = 0.5 \left( 2x_L + \frac{x_L}{2x_L} \right) = 0.5a_L + 0.25 \]

Equations (11) and (12) can be interpreted geometrically as equations of tangents touching the square root curve in the end points of the interval (3). Thus we get an approximation by a broken line (Fig. 1) with a maximum deviation in the breaking point between both octaves

\[ \Delta_{\text{max}} = 0.75 - \sqrt{0.5} = 0.0428 \]

i.e. approx. 6.1% of the correct value.

The construction of the estimate is extremely simple, as the additive constant only depends on the value of the Carry bit after the calculation of result exponent; a simple logic function has been adopted for the realisation (see the listing at the end, lines 27 through 35).
An advantageous side effect of this estimate is that it always holds

\[ y_{1L} \leq a_L \quad \text{and} \quad y_{1H} > a_H \]

so that the information on the real exponent (or "octave affiliation") of the input operand need not be stored for the calculation of the iteration formula (see later).

For this approximation, we can estimate the maximum error after first iteration cycle using (7) as

\[ \Delta_{2\text{max}} = 0.0014 \approx 0.2\% \]

and after the second cycle

\[ \Delta_{3\text{max}} = 0.000013 \approx 0.0002\% \]

which exceeds already the precision of our floating-point format; a constant number of two iteration cycles is fully acceptable and helps to reduce the execution time of each cycle by omitting the test for the accuracy of the result. From this point of view, the choice of a better first estimate can be considered useless, as even the best linear approximation — by an intersecting line with symmetrical deviations — might reduce the max. error \( \Delta_1 \) to appr. 1.5% only, which would yield an accuracy of 0.012% after the first iteration cycle, and consequently would not enable any further reduction of the number of iterations; the construction of any nonlinear approximation would evidently consume more time than one iteration cycle saved.

By this important modification we entered the second group of methods, i.e. reducing the execution time of a single iteration cycle, with our next attention concentrated on the division in (5) as the most time-consuming operation. However queer it
may sound, the most important decision was not to handle the iterations as a loop and
not to use standard division subroutines, and to reduce instead the precision of com­
puting according to the expected accuracy in the respective iteration cycle. In the first
cycle, 10 bits are sufficient for the accuracy calculated above, and — on conditions given
later — even 8 bits (with the precision of 0.4%) will maintain the accuracy of 0.016% in
the next iteration, which still exceeds almost twice the accuracy needed for the
final result. A special division routine was therefore adopted for the first division:
the MS bytes only enter the operation, the LS byte of dividend being replaced by
its MSB followed by the mean of all possible values of the remainder (i.e. 07FH);
in the program, this is realized by shifting in trailing ones into the remainder instead
of zeros except of the first cycle. Full 8 bits are calculated and shifted right before
addition of the first estimate.

For both divisions, due to (13) the end-of-loop is tested for the normalized format
of result only, i.e. one left shift of dividend is added if the starting value of divisor
is greater; as explained before, this can — and always will — occur with the upper
octave operands only. This simplification requires an added precaution for the first
iteration: as the difference may not appear in the MS bytes, a test for zero result
of the first subtraction is unavoidable, causing a skip of the whole first iteration if
true. In this case, the argument lies very close to 0.25 or 1 (within 2^-16), and the

For the second iteration, the kernel of the standard division subroutine only was
adopted, thus enabling to omit all the unnecessary parts (such as testing of signs and
zero values of operands, exponent operations etc.); two calls to the internal division
loop FTDSR of the FTAR.LIB package (appended to the program listing for
reference) reduce the extra memory requirement to an acceptable extent. This enabled
us a different testing of operands to be incorporated; with the second division, the
equivalence of operands means that the estimate is correct, and even the second
iteration is skipped.

The final result of the second iteration is rounded using the shifted-out bit of the
final division by two. This is important not only to maintain the accuracy (the limits
of ± 1 LSB would be met even with mere truncation), but to ensure the stability
of a test loop invoking square root and square in turn: due to the not unique assign­
ment of values mentioned above, the loop will reach a stable pair of values not later
than in the second repetition, while with the final truncation it would produce a se­
quence of continuously decreasing values. An analysis of this type of numerical
instability exceeds the scope of this paper.

The flow diagram of the subroutine described here is given in Fig. 2 and the
complete listing of the program in Fig. 3.
4. COMPARISON WITH OTHER METHODS

The effect of matching the algorithm, its coding and the instruction set of the given microprocessor can be demonstrated by comparison of the worst-case execution times and memory requirements of four types of programs:

a) A user program created by mechanical coding of the Newton-Raphson formula, using standard floating-point arithmetic subroutines, the primitive initial estimate \( y_0 = x \) and the condition \( y_{i+1} = y_i \) for end of iteration, would need 42 bytes of memory and execute in 1-8 ms per iteration\(^2\), i.e. in 6 to 650 ms approx. depending on the size of input numbers.

![Flow diagram for square root](image_url)

Fig. 2. Flow diagram for square root.

\(^2\) 2 MHz clock frequency assumed in all compared cases.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>47</td>
<td>17</td>
<td><strong>FTSR1</strong>: MOV B, A ! ENTRY FOR INPUT OP IN H-L-A REGS</td>
</tr>
<tr>
<td>0001</td>
<td>EB</td>
<td>18</td>
<td>XCHG</td>
</tr>
<tr>
<td>0002</td>
<td>EF</td>
<td>19</td>
<td><strong>FTSRX</strong>: LDA A ! ENTRY FOR INPUT OP IN D-E-B REGS</td>
</tr>
<tr>
<td>0003</td>
<td>B2</td>
<td>20</td>
<td>ORA D ! SET FLAGS ACCORDING TO INPUT OP</td>
</tr>
<tr>
<td>0004</td>
<td>FA7300</td>
<td>C</td>
<td>21</td>
</tr>
<tr>
<td>0007</td>
<td>CH</td>
<td>22</td>
<td>RZ ! EXIT FOR ZERO OPERAND</td>
</tr>
<tr>
<td>0008</td>
<td>7B</td>
<td>23</td>
<td>MOV A, B ! ADD RAG TO EXPONENT</td>
</tr>
<tr>
<td>0009</td>
<td>C611</td>
<td>24</td>
<td>ADD 4H ! DIVIDE BY 2, LSB TO CARRY</td>
</tr>
<tr>
<td>000A</td>
<td>IF</td>
<td>25</td>
<td>RAR</td>
</tr>
<tr>
<td>000B</td>
<td>F5</td>
<td>26</td>
<td>PSH PSW ! STORE RESULT EXPONENT ON STACK</td>
</tr>
<tr>
<td>000C</td>
<td>9F</td>
<td>27</td>
<td>SBF A ! REPEAT CARRY IN ALL ACCU BITS</td>
</tr>
<tr>
<td>000D</td>
<td>EE40</td>
<td>28</td>
<td>XRI 40H ! INVERT A-TH BIT</td>
</tr>
<tr>
<td>0010</td>
<td>E600</td>
<td>29</td>
<td>ANI OCWM ! CLEAR ALL LOWER RITS</td>
</tr>
<tr>
<td>0012</td>
<td>82</td>
<td>30</td>
<td>ADD W ! ADD HI BYTE OF INPUT OPERAND</td>
</tr>
<tr>
<td>0013</td>
<td>IF</td>
<td>31</td>
<td>RAR ! DIVIDE BY 2 AND</td>
</tr>
<tr>
<td>0014</td>
<td>47</td>
<td>32</td>
<td>MOV A, B ! MOV FIRST ESTIMATE TO D-C REGS</td>
</tr>
<tr>
<td>0015</td>
<td>7B</td>
<td>33</td>
<td>MOV A, E</td>
</tr>
<tr>
<td>0016</td>
<td>IF</td>
<td>34</td>
<td>RAR</td>
</tr>
<tr>
<td>0017</td>
<td>6F</td>
<td>35</td>
<td>MOV A, D</td>
</tr>
<tr>
<td>0018</td>
<td>7A</td>
<td>36</td>
<td>MOV A, D ! SUBTRACT NI BYTES OF</td>
</tr>
<tr>
<td>0019</td>
<td>90</td>
<td>37</td>
<td>SUB B ! INPUT OPERAND AND ESTIMATE</td>
</tr>
<tr>
<td>001A</td>
<td>FA2300</td>
<td>C</td>
<td>38</td>
</tr>
<tr>
<td>001B</td>
<td>C300</td>
<td>C</td>
<td>39</td>
</tr>
<tr>
<td>0020</td>
<td>C32800</td>
<td>C</td>
<td>40</td>
</tr>
<tr>
<td>0023</td>
<td>7B</td>
<td>41</td>
<td>MOV A, E ! SHIFT INPUT OPERAND LEFT</td>
</tr>
<tr>
<td>0024</td>
<td>17</td>
<td>42</td>
<td>RAL</td>
</tr>
<tr>
<td>0025</td>
<td>7A</td>
<td>43</td>
<td>MOV A, B</td>
</tr>
<tr>
<td>0026</td>
<td>17</td>
<td>44</td>
<td>RAL</td>
</tr>
<tr>
<td>0027</td>
<td>90</td>
<td>45</td>
<td>SUB B ! AND REPEAT SUBTRACTION OF HI BYTES</td>
</tr>
<tr>
<td>0028</td>
<td>21025B</td>
<td>46</td>
<td>LDI H, <strong>0F0800</strong>: INITIALIZE RESULT REGISTERS</td>
</tr>
<tr>
<td>0029</td>
<td>17</td>
<td>47</td>
<td><strong>FTSR1</strong>: HAL ! FIRST DIVISION LOOP:SHIFT REMAINDER</td>
</tr>
<tr>
<td>002C</td>
<td>8B</td>
<td>48</td>
<td>CMP B ! TRY IF SUBTRACTION POSSIBLE</td>
</tr>
<tr>
<td>002D</td>
<td>FA2000</td>
<td>C</td>
<td>49</td>
</tr>
<tr>
<td>0030</td>
<td>73</td>
<td>50</td>
<td>INX H ! SET RESULT BIT</td>
</tr>
<tr>
<td>0031</td>
<td>90</td>
<td>51</td>
<td>SHR B ! SUBTRACT ESTIMATE</td>
</tr>
<tr>
<td>0032</td>
<td>79</td>
<td>52</td>
<td>INB H ! SHIFT RESULT, TEST BIT TO CARRY</td>
</tr>
<tr>
<td>0033</td>
<td>54</td>
<td>53</td>
<td>JC <strong>FTSR1</strong>: TEST FOR END OF LOOP</td>
</tr>
<tr>
<td>0034</td>
<td>7D</td>
<td>54</td>
<td>MOV A, L ! RESULT TO A</td>
</tr>
<tr>
<td>0035</td>
<td>6F</td>
<td>55</td>
<td>RCU ! SHIFT BACK</td>
</tr>
<tr>
<td>0038</td>
<td>80</td>
<td>56</td>
<td>ADD B ! ADD FIRST ESTIMATE</td>
</tr>
<tr>
<td>0039</td>
<td>1F</td>
<td>57</td>
<td>RAR ! SHIFT TO DIVIDE BY AND</td>
</tr>
<tr>
<td>003A</td>
<td>47</td>
<td>58</td>
<td>MOV A, E ! REPLACE FIRST ESTIMATE BY THE</td>
</tr>
<tr>
<td>003B</td>
<td>79</td>
<td>59</td>
<td>MOV A, L ! RESULT OF FIRST ITERATION</td>
</tr>
<tr>
<td>003C</td>
<td>1F</td>
<td>60</td>
<td>RAR</td>
</tr>
</tbody>
</table>

Fig. 3a. Program listing for square root.
003D 4F 003E EB 003F AF 0040 91 0041 5F 0042 9F 0043 90 0044 57 0045 19 0046 D25300 0049 7C 004A B5 004B C25600 004E 09 004F EB 0050 C36C00 0053 09 0054 29 0055 19 0056 3E05 0058 CB7E00 005B F5 005C 3E01 005E CD7E00 0061 E1 0062 6F 0063 09 0064 7C 0065 1F 0066 57 0067 7D 0068 1F 0069 CEOO 006B 5F 006C 3E00 006E 8A 006F 57 0070 (J1 0071 C9 0072 3E42
0046 C2500 C 70 JNC FT3R3: BRANCH IF RESULT NEGATIVE 0049 7C 71 MOV A+H 004A R5 72 ORA L 004B C25600 C 73 JMP FT3R4: AND G0 TO FINAL FLAG SETTING 004C 0F 7F 74 DAD B: RESTORE ESTIMATE IF CORRECT. 004D EA 75 XCHD B+H: MOVE TO D-E 004E 09 76 JMP FT3R5: GO TO SECOND DIVISION IF POSITIVE 0050 C36CO0 C 77 SUB A+H: PREPARE DE3 := -CBCI 0053 09 78 DAD B: MOVE INPUT OP TO H-L 0054 29 79 DAD D: SUBTRACT 2-ND ESTIMATE FROM IN.OP 0055 19 7A DAD H: BRANCH IF RESULT NEGATIVE 0056 3E05 7B DAD H: GO TO SECOND DIVISION 0057 09 7C DAD H: MOVE TO \[i-E 0058 C37E00 C 7D CALL FTDSR: INITIALIZE RESULT REGISTER 0058 F5 80 CALL FTDSR: FIRST PART OF SECOND DIVISION 005B F5 81 PUSH PSW: PUSH FIRST BYTE OF RESULT ON STACK 005C 3E01 82 CALL FTDSR: INITIALIZE RESULT REGISTER 005E C37E00 C 83 CALL FTDSR: SECOND PART OF SECOND DIVISION 0061 E1 85 POP H: POP FIRST BYTE OF RESULT 0062 AF 86 MOV L+A: APPEND SECOND BYTE FROM ACCUMULATOR 0063 09 90 DAD B: ADD SECOND ESTIMATE 0064 7C 91 MOV A+L: INITIALIZE RESULT REGISTER 0065 4F 92 MOV A+L: MOVE TO \[i-E 0066 5F 93 MOV A+L: AND G0 TO FINAL FLAG SETTING 0067 C2500 C 94 MOV A+L: SHIFT RESULT LEFT ADDING ZERO LSD 0069 EA 95 MOV A+L: TEST FOR END OF DIVISION LOOP 0064 C2500 C 96 MOV A+L: ERROR CODE FOR NEG.INPUT OPERANDS 0064 C2500 C 97 MOV A+L: INITIALIZE RESULT REGISTER 0067 AF 98 MOV A+L: SET DEFAULT RESULT ZERO AND 0067 57 99 POP B: POP RESULT EXPONENT FROM STACK 0070 13 100 POP B: POP RESULT EXPONENT FROM STACK

Fig. 3b. Program listing for square root (cont'd)
b) The same program, but with a better initial estimate (halving the exponent) and testing the difference (8) for end of loop would need appr. 60 bytes and execute in appr. 5 ms.

c) A BASIC-oriented subroutine published in [2], using separate treatment of exponent and mantissa, with the same initial estimate and fixed number of iterations as described above, but standard arithmetic subroutines for mantissa operations, needs 68 bytes and executes in 4099 clock periods, i.e. slightly more than 2 ms.

d) The subroutine described here needs 117 bytes (FTDSR not included) and executes in 1479 clock periods, i.e. 0.74 ms, which is 8.5% only more than needed for the speed-oriented multiplication subroutine described in [6], and even 14% less than for a standard multiplication (such as that described in [2] with a maximum of 0.861 ms).

5. CONCLUSION

This subroutine seems to be attractive for use in self-tuning controllers with square-root filters, because its execution time lies near the geometric average and thus fills the gap between that of the standard software solution and of a peripheral hardware unit (such as iSBC 310 with 0.205 ms), the price of which is much higher than that of the necessary extension of memory and seems to be unaffordable for usual controller applications.

(Received September 27, 1982.)

REFERENCES


Ing. Valentin Chamrad, CSc, Ústav teorie informace a automatizace ČSAV (Institute of Information Theory and Automation — Czechoslovak Academy of Sciences), Pod vodarenskou věži 4, 182 08 Praha 8, Czechoslovakia.