A Radix-2 Digit-by-Digit Architecture for Cube Root

José-Alejandro Piñeiro, Javier D. Bruguera Member, IEEE,
Fabrizio Lamberti Member, IEEE, Paolo Montuschi Senior Member, IEEE

Abstract—A radix-2 digit-recurrence algorithm and architecture for the computation of the cube root are presented in this paper. The original recurrence based on the concept of completing the cube is modified to allow an efficient implementation of the algorithm and the cycle time and area cost of the resulting architecture are estimated as 7.5 times the delay of a full adder and around 9000 \( \text{nanod}2 \) cells, respectively, for double-precision computations.

Index Terms—cube root, digit-by-digit algorithms, computer arithmetic.

I. Introduction

Eлементary functions such as square root, cube root, inverse square root, logarithm and trigonometric functions are important operations in scientific computing, digital signal processing, multimedia and 3D-graphics applications [6], [7], [15], [16].

The computation of one of these operations, cube root, has not received much attention. An algorithm for computing the cube root of an integer operand was proposed in [10], and the cube root of a low-precision floating-point operand can be computed as a particular case of powering algorithms in [13], [17]. Several software routines for higher-precision computation of \( \sqrt[3]{x} \) have been developed [7], [19] resulting in long execution times. The increasing importance of cube root in computer graphics, scientific computing (thermodynamics, geometry, image processing, computer graphics, ...) and other potential fields of application [7], [19], [2], [15] has led to the investigation of algorithms for the hardware computation of this operation [18], [11].

In particular, in [18] a digit-recurrence algorithm for cube rooting has been reported, together with an example of application to the case of radix-2. However, in [18] the main details of the implementation have not been outlined, nor are they easily derived from the general and high level scheme shown in the paper. In fact, radix-2 is a very special case, which allows particular optimizations usually not possible in the general case. On the other hand, [11] presents a general higher radix algorithm/architecture for the computation of the powering function \( X^Y \), of which the cube root is a special case. This method belongs to the family of cascaded chain logarithm-multiplication/division-exponential (L-M/D-E) algorithms where, above all, the computations are speeded-up by using redundancy and online arithmetic. Powering function computation has been also addressed in [14], where an iteration-based architecture relying on a table-driven approach and on the minimax approximation is presented. The general theory of digit-by-digit extraction of the \( m \)-th root is described in [9].

In this paper we present an application of the cube root algorithm to the design of a special purpose unit for radix-2 cube rooting with arbitrary precision \( n \) and redundant digit set \( s_i \in \{-1, 0, +1\} \). The original recurrence is modified to allow an efficient hardware implementation, and a sequential architecture is proposed. Redundancy is employed in order to obtain a reduced cycle time. A further reduction of 2 cycles in the overall latency is achieved with no extra cost through a simple initialization. The cycle time and area cost are estimated for double precision computations, based on a technology-independent model for the delay and area of the main components. The obtained results are finally compared with those of other implementations.

This paper is organized as follows: in Section II the theory of cube root algorithm is briefly reviewed. Section III outlines the design of a special purpose unit for radix-2 cubic root based on a modified recurrence. In Section IV the estimates of cycle time and area requested are given and a comparison with other implementations is provided.

II. RADIX-2 CUBE ROOT

It is well known [5], [8] that the algorithm for square rooting in radix-2 is based on the concept of completing the square. In this Section, key steps for extending this algorithm to radix-2 cube root will be summarized. Details on the recurrence equation, on the rules for digit selection and on the initialization conditions will be provided using the notation of Table 1.

A. Recurrence of radix-2 cube root

The algorithm for cube rooting is based on the concept of completing the 3rd-power. As digit-by-digit methods for
square root and division [5], cube rooting is regulated by a recurrence equation that, according to [18], is
\[
w[i] = 2w[i-1] - [3S[i-1]^2 + 3S[i-1]s_i2^{-i} + s_i^22^{-2i}]s_i
\]
where
\[
S[i] = S[i-1] + s_i2^{-i}
\]
In order to produce a result \( S \) which is normalized to \( 1/2 \leq S < 1 \), it is necessary to consider \( 2^{-3} \leq x < 1 \). We assume to have \( x \) available at the beginning of the operations in non redundant representation, while the residual \( w[i-1] \) is stored in carry save representation. Finally, since the determination of the digit \( s_i \) is carried out by inspecting the value of the residual \( w[i-1] \), in order to avoid full length comparisons, the digit "—1" (i.e. a redundancy) is introduced in the possible digit selections, i.e. \( s_i \in \{-1, 0, +1\} \). This allows the estimate \( w[i-1] \) to be considered in place of the full precision residual \( w[i-1] \).

B. Initialization and selection intervals

Unlike other algorithms, we observe that our radix-2 cube root algorithm does not require any initialization. In fact, if \( i = 1 \) and \( S[0] = 0 \) we get the expression of the first region of convergence \(-1 \leq w[0] \leq 1 \). Since \( w[0] = x \) with \( 2^{-3} \leq x < 1 \), then certainly \( w[0] \) belongs to the first region of convergence, and hence the algorithm converges without the need for any initialization. However, we observe that an initialization could be useful if, at the cost of no additional delay, it allows the saving of a few iterations, because of the bits of the result provided by the initialization step. According to [18], a first possible initialization is \( S[1] = 0.1_2 = 1/2 \). A more interesting initialization [12], which is also trivial, is to set \( i = 3 \). \( S[2] = 2^{-3} + 2^{-2} \) and \( w[2] = (x - S[2]^3)2^2 \). A first advantage of this 2-bit initialization is that, without any additional cost, the number of iterations decreases by two. One further advantage is that, given this initialization, selection intervals are independent of the iteration index \( i \), and can be written as
\[
\begin{align*}
s_i = +1 & \quad if \quad 0 \leq w[i-1] < 3 \\
s_i = 0 & \quad if \quad -3/8 \leq w[i-1] \leq 3/8 \\
s_i = -1 & \quad if \quad -3 < w[i-1] \leq 0
\end{align*}
\]
Note that in (3) there is an overlapping in the selection rules. This is a direct consequence of the adoption of the redundant digit set \( \{-1, 0, +1\} \), which, we will see in the next paragraph, becomes very handy.

C. Number of bits and digit selection rules

The digit selection intervals (3) refer to the value of the residual \( w[i-1] \), expressed in its full precision. In order to avoid full length comparisons it is known [3], [8], that while selecting the digits of the result, the overlapping of the digit selection intervals (3) is used to transform into digit selection rules of the form \( s_i = select(w[i-1], \hat{S}) \), where select is the digit selection function which, in general, depends on the estimates \( w[i-1] \) and \( \hat{S} \) of the residual and of the result, respectively. The estimate \( w[i-1] \) can be obtained by considering \( c \) integer bits of the residual \( w[i-1] \), and truncating it up to the \( t \)-th fractional position. According to [12], in order to obtain a valid digit selection function, \( c = 3 \) integer and \( t = 2 \) fractional bits are needed. The digit selection rules, in terms of discretized values \( w[i-1] \) with step \( 2^{-t} = 1/4 \), then become
\[
\begin{align*}
s_i = +1 & \quad if \quad 0 \leq w[i-1] < 11/4 \\
s_i = 0 & \quad if \quad w[i-1] = -1/4 \\
s_i = -1 & \quad if \quad -13/4 \leq w[i-1] \leq -1/2
\end{align*}
\]
This corresponds to obtaining \( w[i-1] \) by taking the 5 most significant bit weights only of the carry-save representation of \( w[i-1] \). The logic assimilation into non redundant form of these 5+5 sum & carry bits of \( w[i-1] \), as seen later, can be incorporated as a part of the digit selection hardware.

III. IMPLEMENTATION OF RADIX-2 CUBE ROOT

Based on the algorithm and digit-selection rules detailed in the previous Section, a sequential architecture for the computation of the radix-2 cube root is now proposed. We focus our interest in the development of a modified recurrence which allows a more efficient hardware implementation than the one resulting from the original recurrence. We then briefly provide an overview of alternative implementations.

A. Modified recurrence

The recurrence for the radix-2 cube root given by (1) and, by replacing \( S[i] = S[i-1] + s_i2^{-i} \), can be rewritten as \( w[i] = 2w[i-1] - [3S[i-1]^2 + 3S[i-1]s_i2^{-i} + s_i^22^{-2i}]s_i \). If we define \( D[i+1] = 3S[i+1] \cdot S[i] \), the final recurrence is obtained, \( w[i] = 2w[i-1] - (D[i+1] + s_i^22^{-2i})s_i \). After some simple passages we get the recurrence for \( D[i] \):
\[
D[i] = D[i-1] + 3S[i](s_i+1)2^{-i} + s_i2^{-i}
\]
In each iteration, the computation of the recurrences for \( w[i] \) and \( D[i] \), the selection of the next digit of the result \( s_i \) and the update of the partial result \( S[i] \) are needed. Note that in (5), two digits of the result, \( s_i \) and \( s_{i+1} \), are used. Therefore, we propose to compute in iteration \( i \) (with \( i \geq 3 \)) the recurrences for \( w[i] \) and \( D[i-1] \) using the digit \( s_{i-1} \), obtained in iteration \( i-1 \) from \( w[i-2] \), and the digit \( s_i \), obtained from \( w[i-1] \) as the first step of iteration \( i \). In such a case, with the initialization of Section II-B, we have \( S[0] = 0, S[1] = 1/2, S[2] = 3/4 \) and \( D[1] = 3S[1]S[2] = 9/8 \). A timing diagram of our algorithm is shown in Figure 1, where it is highlighted that the initialization reduces the overall latency by two cycles, i.e., the iterations start at \( i = 3 \). Summarizing, each iteration has to compute the recurrences

\[
\begin{align*}
    s_i &= \text{select}(w[i-1]) \\
    w[i] &= 2w[i-1] - (D[i-1] + s_i2^{-2i})s_i \\
    S[i] &= S[i-1] + s_i2^{-i} \\
    D[i-1] &= D[i-2] + 3S[i-1](s_i2^{-i} + s_{i-1}2^{-(i-1)})
\end{align*}
\]

### B. Architecture

In this subsection we present a sequential architecture for the proposed algorithm. A block diagram is shown in Figure 2, where the block "select" performs in a single step the logic "assimilation" into non redundant form of the residual \( w[i-1] \) (i.e., the most significant 5 bit weights only of the residual \( w[i-1] \) in carry save form), and the digit selection based on the resulting value. Since \( s_i \in \{-1, 0, +1\} \), we observe that \( s_i = s_i \), and the recurrences for \( w[i] \) and \( D[i-1] \) become

\[
\begin{align*}
    w[i] &= 2w[i-1] - D[i-1]s_i - s_i2^{-2i} \\
    D[i-1] &= D[i-2] + 3S[i-1]s_i2^{-i} + 3S[i-1]s_{i-1}2^{-(i-1)}
\end{align*}
\]

In order to speed up the operation, we use redundant representations: carry-save (CS) for variables \( w[i] \) and \( D[i] \) and signed-digit (SD) for \( S[i] \). The high number of operations involved in the computation of \( D[i-1] \) could lead to a long critical path, slowing down the overall performance of the architecture. Therefore, apart from storing \( w[i-1] \) and \( D[i-2] \), the intermediate values \( 3S[i-1]2^{-i} \) and \( S[i-1]2^{-i} \) are also stored to minimize the delay of the critical path. The digit \( s_{i-1} \) is stored in a two-bit register and combined in the next iteration with the stored value \( 3S[i-1]2^{-i} \) and a wired left shift to obtain \( 3S[i-1]s_{i-1}2^{-(i-1)} \). The main features of the proposed architecture are:

- All values in the \( S-path \) are in SD radix-2 (SD-2) representation, in order to easily accomodate the concatenation of \( S[i-1]2^{-i} \) and \( s_{i-1}2^{-2i} \) to form \( S[i]2^{-i} \), and are recoded to CS representation before being used in the \( DW-path \).
- An SD radix-2 adder performs the computation of

\[
3S[i]2^{-(i+1)} = S[i]2^{-(i+1)} + 2S[i]2^{-(i+1)} \text{, with } 2S[i]2^{-(i+1)} \text{ obtained by performing a wired left shift of } S[i]2^{-(i+1)} \text{. Since two bit-vectors } (P-N) \text{ are necessary to represent a SD word, four operands must be accumulated by a } 4:2 \text{ adder (SDA).}
\]

- The recoding unit performs a recoding from SD to CS representation, by computing \( P-N \rightarrow P + N + 1 \), with \( P \) and \( N \) as main inputs and \( N \) as the third input.
- The product of the digit \( s_i \) by an input operand \( \delta \) has three possible results: \( \delta \), when \( s_i = +1 \), \( 0 \) when \( s_i = 0 \), and \( -\delta \) when \( s_i = -1 \). This operation can be performed, as explained in [4], [12], by using a 2:1 multiplexer composed of two levels of \text{\textit{nand}} gates which generates \( \delta \), \( 0 \), or \( \bar{\delta} \). The control signal \( b_0 \) of the multiplexer selects between \( +\delta \) and \( -\delta \), while \( b_1 \) selects between any of these values and \( 0 \). The extra 1 necessary to complete the two’s complement operation of \( \delta \) can be later injected using the empty slot in the least-significant bit of the carry word in the CSA adder where the result is consumed.
- Whenever the product must be generated with opposite sign (multiplication by \( -s_i \) instead of \( s_i \)), a bit inversion in the control signal \( b_0 \) suffices to guarantee a correct operation. Buffering is used for the control signals (the \( s_i \) digits) of long multiplexers.
- The value \( -s_i2^{-2i} \), in the \( DW-path \), is computed by using a shifting register which stores a constant vector \( 1, 0, \ldots, 0 \) (mask), and performs a right shift of two positions each iteration, and employing the 2:1 mux for computing the multiplication by the digit \( s_i \) with opposite sign. This mask is also employed as a control signal in the concatenation.

### C. Alternative architectures

The proposed architecture is recommended when the delay of the selection function and the shifter used in the \( S-path \) are high and let it become the critical path. If this is not the case, an alternative architecture can be adopted based on a different way of computing the term \( 3S[i-1]s_{i-1}2^{-(i-1)} \). In this case, the wired left shift and the multiplication of \( 3S[i-1]2^{-i} \) by the digit \( s_{i-1} \) are performed in iteration \( i-1 \) and the result is stored in a register. Also other alternative implementations are possible, such as moving or reallocating the different blocks, which (marginally) can trade area for execution time. However, they basically do not introduce relevant changes. For example, a slightly different implementation can be defined where the SDA in the \( S-path \) is eliminated and placed in the \( DW-path \). Now, \( 3S[i-1]2^{-i} \) is not stored in the register and is computed, in the \( DW-path \), using the stored value of \( S[i-1]2^{-i} \). The advantage is the use of 4 registers instead of 5 and 6, as in the two previous architectures. Based on the analysis carried out in [12], where we evaluate in detail several alternative architectures, the unit of Figure 2 emerges as the one with the best overall area \times delay metric.

\footnote{The bit inversion of \( N \) can thus be performed in parallel with the \text{\textit{xor}} operation between \( P \) and \( 1 \).}
**IV. EVALUATION OF THE PROPOSED ARCHITECTURE**

A. Area and delay estimates

In this section we present estimates of the cycle time and the area cost of the proposed architecture, for a precision of $n = 53$ bits. These estimates are based on a model for the cost and delay of the logic blocks used. The actual delays and area costs depend on the technology used and on the actual implementation. However, this model provides a good first-order approximation to the actual execution time and area values, and allows a technology-independent comparison with similar architectures. Such a model has been used for the analysis of similar digit-recurrence architectures, with fairly accurate results [1], [4]. The assumptions made, the delay equations (which are composed of a base component and a factor which depends on the load $L$ or fan-out) and a detailed explanation of the model can be found in [12]. The units employed are the area of a nand2 standard cell ($\text{nand2}$), and the delay $\tau_{\text{inv}}$ of an inverter with fan out of four (FO4, $L = 4$).

This unit can be easily converted to $\tau_{\text{fa}}$, the delay of a full-adder, since in this model $\tau_{\text{fa}} = 4.0 + 0.2 \times 4 = 4.9 \tau_{\text{inv}}$ [12]. Table II shows the delay estimates for the logic blocks employed in our architecture, according to the

<table>
<thead>
<tr>
<th>Path Description</th>
<th>Delay ($\tau_{\text{inv}}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>select (9.6) + buffer (3.2) + mux (1.9) + 3:2 CSA (c) (2.5) + 4:2 CSA (6.8) + regW (4.0)</td>
<td>28.0</td>
</tr>
<tr>
<td>select (9.6) + buffer (3.6) + mux (2.0) + 4:2 CSA (6.8) + regW (4.0)</td>
<td>26.0</td>
</tr>
<tr>
<td>select (9.6) + buffer (3.6) +mux (2.0) + 4:2 CSA (6.8) + regW (4.0)</td>
<td>15.8</td>
</tr>
<tr>
<td>rec. (4.6) + mux (2.0) + 4:2 CSA (7.1) +mux (2.0) + 4:2 CSA (6.8) + regW (4.0)</td>
<td>35.1</td>
</tr>
<tr>
<td>rec. (4.6) + mux (2.0) + 4:2 CSA (7.0) + 4:2 CSA (7.1) +mux (2.0)</td>
<td>33.5</td>
</tr>
<tr>
<td>select (9.6) + concat. (2.0) + SD adder (6.8) + regS[i-1]/2^{-1} (3.2)</td>
<td>21.6</td>
</tr>
</tbody>
</table>

### Table III
Delay of the Main Paths in Our Architecture
proposed model, and the area estimates obtained according to the model used in [4], for 53-bit computations. When several values for the delay are shown, these correspond to units of the same type with different loads. For the unit implementing the selection rules described in Section II, our values are based on estimates for similar units presented in [4], where a selection function with 4-bit redundant input has a delay equivalent to 1.67τfa and a unit with 10-bit redundant input has a delay equivalent to 2.67τfa. Since our select unit has 5-bit redundant input, a conservative estimate is considering 2.07τfa, hence 9.67τinv. For the registers, the estimates are taken directly from an implementation performed with Synopsys synthesis tools, using a CMOS 0.35 μm standard-cells library [1]. According to the values shown in Table II, we can estimate the delay of the critical path in our architecture as 35.1τinv, as shown in Table III. Since the delay of a full-adder in our model is 4.8τinv, the cycle time can be estimated to be around 8τinv. The total area can be estimated as 7135 nand2 cells, when the latency of the algorithm is 54 cycles, or 9035 nand2 cells when an on-the-fly conversion and rounding scheme is employed (since the area estimate for this unit is 1900 nand2 cells [4]). In this case, the latency would be 52 cycles, with a total execution time of around 1825τinv.

### Table II

<table>
<thead>
<tr>
<th>logic block</th>
<th>delay (τinv)</th>
<th>area (nand2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>9.6</td>
<td>201</td>
</tr>
<tr>
<td>2:1 muxes</td>
<td>3.2/3.6</td>
<td>35</td>
</tr>
<tr>
<td>2:1 nand</td>
<td>1.9/2.0</td>
<td>170</td>
</tr>
<tr>
<td>recording unit</td>
<td>4.6</td>
<td>360</td>
</tr>
<tr>
<td>3:2 CSA</td>
<td>5.0 (ah)/2.5(c)</td>
<td>360</td>
</tr>
<tr>
<td>4:2 CSA</td>
<td>6.8/7.0/7.1</td>
<td>2160</td>
</tr>
<tr>
<td>SDA</td>
<td>6.8</td>
<td>720</td>
</tr>
<tr>
<td>concat registers</td>
<td>2.0</td>
<td>180</td>
</tr>
<tr>
<td>TOTAL</td>
<td>35.1 (crit. path)</td>
<td>7135</td>
</tr>
</tbody>
</table>

### B. Comparisons with other implementations

In [18] a digit-recurrence algorithm for cube rooting was presented, together with a corresponding very high level block diagram. No detailed architecture implementing the algorithm was proposed, which makes it difficult to estimate it. In [11] the authors presented a high-radix \((r = 2^5)\) algorithm and architecture for the computation of the powering function \((X^Y)\) with exponents of the type \(Y = 1/m\), of which the cube root operation is a particular case \((Y = 1/3)\). Since for this algorithm the details of implementation exist, we will roughly compare them.

The algorithm proposed in [11] belongs to the class of L-M/D-E methods, and is based on the identity \(X^Y = 2^{Y \log_2(X)}\). Considering a floating-point representation of \(X\), the powering function can be computed as \(X^Y = 2^{Y \log_2(X)} + I\), with \(I\) and \(F\) the integer and fractional parts of the product \(Y \log_2(M_E)\), and \(M_E\) and \(E_2\) the significand and exponent, respectively, of the floating-point operand \(X\). With this algorithm, powering is computed as a sequence of overlapped operations: (a) digit-recurrence logarithm \((\log_2(M_E))\), (b) left-to-right carry free (LRCF) multiplication \((Y \log_2(M_E))\), and (c) on-line exponential \((2^Y)\). A redundant representation is used and the selection in (a) and (c) is done by rounding, except for the first iteration, where selection by table look-up is required. Unlike some previous algorithms [17], this method operates on two input operands \(X\) and \(Y\), and computes the powering function with much higher precision than table-based implementations, such as that presented in [6].

The extension of such an algorithm to computations with non-integer exponents \((Y = 1/m)\) requires some minor adjustments [11]: \(X^Y = 2^{1/m \log_2(M_E) \log \psi}/mE_2\). The integer part \(1/m\) is now zero (which reduces by one cycle the overall latency of the algorithm) and therefore \(2^{1/m \log_2(M_E)} = 2^{F \log_2(M_E)}\). On the other hand, \(2^{1/m}\) must be handled in a different way, since the exponent is not integer, i.e. \(2^{1/m} = 2^{[1/m] + \psi}/mE_2\), with \(\psi = E_2 \% m\) the modulus operation. The term \(2^{[1/m]} 2^{\psi} \ln 2 \) can be obtained as the output of the exponential stage, since the exponential algorithm computes \(\exp(\psi)\) when the input argument is \(\psi\) and \(E[\psi] = 0\) is taken as initial value for the \(E[\psi]\) recurrence instead of \(E[1] = 1\). For a small \(m\), a table can be employed to store the \(m\) values \(\exp(\psi)\), which go from \(2^0, 2^1/m, \ldots\) to \(2^{m-1}/m\), and then the corresponding value can be used to load and initialize the register \(E[\psi]\) depending on \(\psi\), the result of the modulus operation. The floating-point exponent of the result is the integer value \([E_2/m]\) and thus, the output of the algorithm is the floating-point result \(X^Y = 2^{[1/m]} 2^{\psi} \ln 2 E_2/m\).

The cycle time and area estimates for the composite algorithm of [11] are shown in Table IV, for double-precision results and have been obtained according to the same model of Table II. A deep analysis of the data in Table IV shows that for higher-radix implementations (basically above 64), the Lookup Table of [11] accounts for more than half the area of the whole circuit. For lower radices the biggest contributors (in percentage) to area are the combinational logic blocks which, as well as the Lookup Table, cannot be "reused" for other purposes.

The proposed architecture also accounts entirely for dedicated areas which cannot be directly reused for other purposes. However, although having a higher execution time, the proposed architecture presents significant area savings, which allow it to become more amenable to chip implementations than the architecture in [11]. Not to be forgotten is that the final residual is computed in exact form.

### V. Concluding remarks

An implementation of the radix-2 digit-by-digit algorithm for the computation of the cube root has been presented,
based on the concept of completing the cube. Details concerning initialization, intervals for digit-selection and iteration-independent digit-selection rules for a redundant digit set $s_i \in \{-1, 0, +1\}$ have been provided. In particular, an effective initialization scheme allows a reduction of two cycles in the overall latency without any additional cost.

The original recurrence has been modified to allow an efficient hardware implementation of the algorithm, and a sequential architecture has been proposed, with redundant representation of the variables (CS for the residual $\omega[i]$ and recurrence $D[i]$, and SD for the partial result $S[i]$).

The cycle time and area cost of this architecture have been estimated based on a technology-independent model for the delay and area of the main logic blocks used. These estimates show that cube root can be computed in a dedicated unit similar to those employed for digit-recurrence division and square root.

The proposed special purpose unit has been compared with the architecture presented in [11]. With respect to the unit presented in [11] we see that the proposed implementation well fits in cases where a small area can be dedicated to computation of cube root. In addition, as for square root, it is very likely [12] that for high radices the area $\times$ delay parameter will improve with our implementation.

ACKNOWLEDGEMENTS

The authors thank E. Antelo and A. Nannarelli for their contribution in obtaining the area and delay estimates for our architecture. This work was done when J.-A. Piñeiro was with the University of Santiago de Compostela.

REFERENCES


<table>
<thead>
<tr>
<th>unit</th>
<th>radix</th>
<th>latency</th>
<th>cycle time ($\tau_{inv}$)</th>
<th>exec. time ($\tau_{inv}$)</th>
<th>area ($\text{AND}$)</th>
<th>area $\times$ delay ($\times 1M$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>prop</td>
<td>256</td>
<td>12</td>
<td>50.4</td>
<td>604.8</td>
<td>98568</td>
<td>59.6</td>
</tr>
<tr>
<td>16</td>
<td>128</td>
<td>13</td>
<td>48.0</td>
<td>624.0</td>
<td>57378</td>
<td>35.8</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>16</td>
<td>45.6</td>
<td>729.6</td>
<td>47214</td>
<td>34.4</td>
</tr>
<tr>
<td>512</td>
<td>32</td>
<td>17</td>
<td>43.2</td>
<td>734.4</td>
<td>33732</td>
<td>24.8</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>20</td>
<td>40.8</td>
<td>816.0</td>
<td>24360</td>
<td>19.9</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>15</td>
<td>35.1</td>
<td>1020.0</td>
<td>25314</td>
<td>15.3</td>
</tr>
<tr>
<td>arch. of [11]</td>
<td>8</td>
<td>25</td>
<td>35.8</td>
<td>18252</td>
<td>9015</td>
<td>10.3</td>
</tr>
</tbody>
</table>

TABLE IV

Comparison of execution time and total area with the powering function.