# Factorization Architecture by Direct Root Computation for Algebraic Soft-Decision Decoding of Reed-Solomon Codes

Jun Ma and Alexander Vardy
Department of Electrical Engineering
University of California, San Diego
La Jolla, CA 92093, USA
Email: jun.jma@gmail.com, vardy@kilimanjaro.ucsd.edu

Zhongfeng Wang and Qinqin Chen
School of EECS
Oregon Sate University
Corvallis, OR 97331-3211, USA
Email: zwang@eecs.orst.edu

***Abstract*— Algebraic soft-decision decoding [4] is a recent breakthrough in decoding of Reed-Solomon codes and significant decoding gain can be achieved over conventional hard-decision decoding. Bivariate polynomial factorization is an important step of the new decoding algorithm and contributes to a significant portion of the overall decoding latency. In this paper, a novel architecture based on direct root computation is proposed to greatly reduce the factorization latency. Direct root computation is feasible because in most practical applications of algebraic soft-decision decoding of RS codes, sufficient decoding gain can be achieved with a relatively low interpolation cost, which results in bivariate polynomial of small Y-degree. Compared with existing works, not only does our new architecture have a significantly smaller worst-case decoding latency, but it is also more area efficient.**

**Keywords:** Reed-Solomon codes, soft-decision decoder, factorization.

## I. Introduction

Factorization is a key step in algebraic soft-decision decoding [4] of Reed-Solomon codes. One major step of the factorization procedure is root computation for polynomial equations. The factorization architecture of [1] uses Chien search to find roots of a polynomial at the beginning of each iteration. This approach is very time consuming, especially for RS codes defined over a large finite field. In [7], a root-order prediction based method is proposed by Zhang and Parhi, who observed that the orders of roots seldom change between factorization iterations. The VLSI architecture based on this observation can improve the average factorization latency. However, the worst case latency of [7] is not any better than that of [1], because the root-order prediction has a non-zero failure rate and one has to resort to Chien search after detecting a root-order prediction failure. Thus the root-order prediction based architecture can not be used in applications with a stringent latency requirement.

In this paper, we present a fast factorization architecture based on direct computation of polynomial equation roots. Direct root computation is practically feasible only for low-degree polynomials. Fortunately this is not a problem for most practical applications of algebraic soft-decision decoding, where significant decoding gain can be achieved with relatively low interpolation cost. Low interpolation cost results in bivariate polynomials with Y-degree lower than 5. This is especially true when the iterative interpolation and factorization method [2] is applied. As shown in Figure 1 for algebraic soft-decision decoding of an $(458, 410)$ RS code, if factorization is performed on judiciously selected intermediate interpolation results, about $0.4$dB decoding gain can be achieved over hard-decision decoding at an error rate or $10^{-6}$ with an interpolation cost equal to $4580$. This interpolation cost maps to a maximum Y-degree $r$ equal to $4$ for the candidate bivariate polynomials. Actually, interpolation cost up to 6139 can be supported with this choice of $r$.

Now the applicability of the direct root computation method to bivariate polynomial factorization in practical soft-decision decoding of RS code is established. Let us assume, throughout the rest of this paper, that we work on a finite field of $2^p$ elements, i.e., $\mathbb{F}_{2^p}$. Apparently, Chien search based root computation has a latency in the order of $2^p$, which is very inefficient for solving low-degree polynomial equations in a large finite field. As will be shown later, the direct root computation can be implemented in hardware with a latency of only $2p$. Another advantage of using the direct root finding method over the conventional exhaustive search method is that the order (multiplicity) of each root can be precisely determined. This leads to a factorization architecture where only roots of polynomial equations need to be routed to desired hardware resources. Compared to the architecture of [7], where large number of MUXes are used to route both roots and polynomial coefficients, our new architecture is more area efficient.

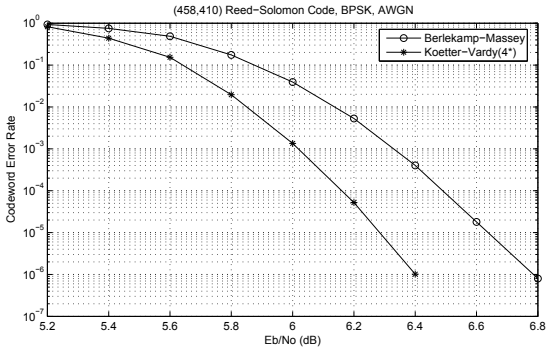


Fig. 1. Decoding Performance for (458, 410) RS code over AWGN Channel with BPSK Modulation

The rest of the paper is organized as follows. Section II presents direct root computation method for quadratic, cubic and quartic polynomial equations defined over $\mathbb{F}_{2^p}$. Efficient VLSI architecture for direct root computation is also introduced there. Overall factorization architecture is given in Section III. Section IV gives an example of factorization architecture for decoding a $(458, 410)$ RS codes. Conclusions are drawn in Section V.

Throughout the rest of the paper, we assume that the readers are familiar with the factorization algorithm presented in [6].

## II. Direct Root Computation for Polynomial Equation of Degree Lower than 5

A method for directly computing roots of affine polynomials over $\mathbb{F}_{2^p}$ is given in [3]. To find roots of a non-affine polynomial, one can either apply transformation to convert the non-affine polynomial to an affine polynomial or derive the minimum affine multiple of the polynomial. The second approach is usually very complicated and may not have any advantage in complexity over exhaustive search. Fortunately, for low degree($< 5$) polynomials, only transformation is required. In this section, method and apparatus for solving polynomial equation of degree lower than 5 are described.

As shown in [3], finding roots of an affine polynomial with coefficients in $\mathbb{F}_{2^p}$ is equivalent to solving the following linear equation array $\underline{y}\ \underline{\underline{M}} = \underline{z}$, where $\underline{y}$ and $\underline{z}$ are binary p-tuple row vectors and $\underline{\underline{M}}$ is a binary p-by-p matrix. The key step in solving such linear equation array is to transform the matrix $\underline{\underline{M}}$ to a reduced triangular idempotent (RTI) form. An estimate of the number and type of hardware units required to implement the matrix reduction is

given below in Table I. The critical path of the entire matrix reduction circuitry consists of 1 2:1 MUX, $\lceil log_2 p \rceil + 2$ AND gates, $\lceil log_2 p \rceil + 1$ OR gates, and 1 inverter.

### A. The Linear, Quadratic, Cubic and Quartic Polynomials

For linear polynomial, finding its root only takes a division operation. The quadratic polynomial $f(Y) = aY^2 + bY + c$, where $a, b, c \in \mathbb{F}_{2^p}$, is affine in nature. We will show later that cubic polynomial can be handled with the same hardware resource that transforms a general quartic polynomial to an affine polynomial.

In the case of a quartic polynomial $f(Y) = f_4Y^4 + f_3Y^3 + f_2Y^2 + f_1Y + f_0$, if $f_3 \neq 0$, $f(Y) = 0$ can be re-written as

$$Y^4 + \frac{f_3}{f_4}(Y^3 + \frac{f_2}{f_3}Y^2 + \frac{f_1}{f_3}Y + \frac{f_0}{f_3}) = 0.$$

The following "shift" transformation can be applied:

$$\begin{aligned} & Y^4 + \frac{f_3}{f_4}(Y^3 + \frac{f_2}{f_3}Y^2 + \frac{f_1}{f_3}Y + \frac{f_0}{f_3}) \\ = \quad & (Y + \sqrt{\frac{f_1}{f_3}})^4 + \frac{f_3}{f_4}(Y + \sqrt{\frac{f_1}{f_3}})^3 \\ + \quad & \frac{f_3}{f_4}(\sqrt{\frac{f_1}{f_3}} + \frac{f_2}{f_3})(Y + \sqrt{\frac{f_1}{f_3}})^2 + \frac{f_3}{f_4}\frac{f_2 f_1}{f_3^2} + \frac{f_0}{f_4} + (\frac{f_1}{f_3})^2 \end{aligned}$$

We now define $a \stackrel{\text{def}}{=} f_3$, $b \stackrel{\text{def}}{=} (\sqrt{f_1 f_3} + f_2)$, $c \stackrel{\text{def}}{=} \frac{f_2 f_1}{f_3} + f_0 + (\frac{f_1}{f_3})^2 f_4$, and define the variable substitution $Z \stackrel{\text{def}}{=} \frac{1}{Y + \sqrt{\frac{f_1}{f_3}}}$. Note that the variable substitution through shifting and reciprocation is valid only if $y = \sqrt{\frac{f_1}{f_3}}$ is not a root of polynomial $f(Y)$, or equivalently $c \neq 0$. In this case, finding roots $y$ of $f(Y) = 0$ is equivalent to finding roots $z$ of polynomial $g(Z) = cZ^4 + bZ^2 + aZ + f_4 = 0$ followed by reciprocation and shifting operations. On the other hand, if $c = 0$, then $y = \sqrt{\frac{f_1}{f_3}}$ is a root of the original polynomial, and this root can not be found by solving $g(Z) = 0$. In this case, root $y = \sqrt{\frac{f_1}{f_3}}$ has a multiplicity of at least 2. It may have a multiplicity of 3 if $b = 0$ and $a \neq 0$, and it may have a multiplicity of 4 if $b = 0$ and $a = 0$. In this case, roots other than $y = \sqrt{\frac{f_1}{f_3}}$, if exist, can still be derived from solutions of $g(Z) = 0$. Polynomial $g(Z) = 0$ is an affine polynomial equation and once its root $z$ is found, it can be inverted and shifted to obtain root $y$ for the original degree-4 polynomial. Note that the above transform is necessary only if $f_3 \neq 0$. Otherwise, the polynomial $f(Y)$ is an affine polynomial in the first place.

In a brief summary, there are 2 steps involved in solving a general degree-4 polynomial equation. In the first step, we apply shift and reciprocation to transform the degree-4 polynomial to an affine polynomial of the same degree. In the second step, we form the $p \times p$ binary matrix $\underline{M}$ according to [3] and convert it to the RTI form. Step 1 can be implemented with the architectures shown in Figure 2. To reduce hardware consumption, only 1 multiplier and 1 inverter is used in Figure 2 and is time shared among various operations. The inputs to the multiplier is controlled by the 2 4:1 MUXes and the corresponding timing diagram is shown in Figure 3.

To optimize resource utilization, the quartic polynomial equation solver can be configured to solve polynomial equations of lower degrees, too. The MUXes at the input and output of Figure 2 serves this purpose. As will be shown in Section III, there is no need for a seperate cubic polynomial equation solver. The control signal $s_{in}$ and $s_{out}$ are defined as follows:

$$s_{in} = \begin{cases} 1, & \text{if } \deg Q(0, Y) = 3; \\ 0, & \text{otherwise.} \end{cases} \qquad s_{out} = \begin{cases} 1, & \text{if } f_3 = 0; \\ 0, & \text{otherwise.} \end{cases}$$

TABLE I
HARDWARE UNIT COUNTS FOR IMPLEMENTATION OF THE MATRIX REDUCTION ALGORITHM

| Unit Type | Inverter | AND | OR |
|---|---|---|---|
| Count | $2p$ | $3.5p^2 - 1.5p$ | $p^2 + 2p - 2$ |
| Unit Type | XOR | MUX | Register |
| Count | $p^2 - p$ | $2p^2 + p - 1$ | $p^2 + p$ |

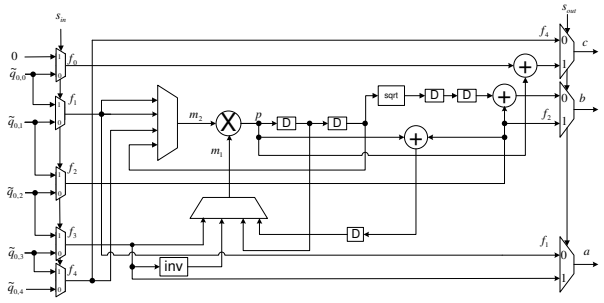


Fig. 2. Architecture for transforming a general quartic polynomial to an affine quartic polynomial

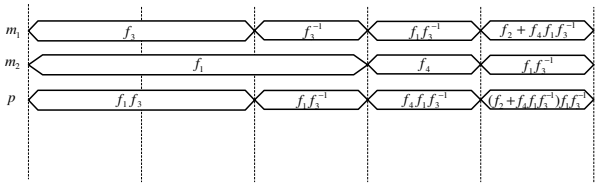

Fig. 3. Timing diagram for transforming a general quartic polynomial to an affine quartic polynomial

### B. Root Order Determination

The orders of the roots found at a certain iteration level of the factorization algorithm are key to resource allocation and scheduling for the next iteration level. In [7], the order of each root found at certain iteration is predicated based on the statistics collected from simulations. Polynomial and root scheduling for succeeding iterations are made accordingly. The predication, though accurate most of the time, has a non-zero failure rate. Thus a mechanism to check the correctness of the predication has to be applied. Once a prediction fails, an exhaustive root search has to be carried out and hardware resources need to be reallocated, which leads to longer latency for the factorization procedure. In addition, a large number of MUXes are needed to route polynomial coefficients to different polynomial update engines. This issue can be completely circumvented by using the direct root computation since the order of the roots found by direct computation can be precisely determined. To illustrate this for a quartic polynomial, we give the following theorem.

*Theorem 1:* The roots of a quartic affine polynomial equation of the following form $X^4 + \mu_2 X^2 + \mu_1 X + \mu_0 = 0$, where $\mu_0, \mu_1, \mu_2 \in \mathbb{F}_{2^p}$, have the following properties:

- The polynomial equation can have no root, 1 single root, 2 distinct roots, or 4 distinct roots in $\mathbb{F}_{2^p}$.
- If the polynomial equation only has 1 root, the root can only be of multiplicity 1 or 4.
- If the polynomial equation has 2 distinct roots, either both roots are of multiplicity 1 or both roots have a multiplicity of 2.

Due to space limitation, a proof of the theorem is omitted. For a general (non-affine) polynomial $f(Y) = f_4Y^4 + f_3Y^3 + f_2Y^2 + f_1Y + f_0$, a combination of the above theorem and the solutions of the binary linear equation array can be used to infer the root orders.

## III. OVERALL FACTORIZATION ARCHITECTURE

A parallel factorization architecture, where root computation and polynomial update for all $Q(X, Y)$ in the same iteration level are carried out simultaneously, is given in Figure 5. The polynomial update can be implemented by FST (fast shift transform) proposed

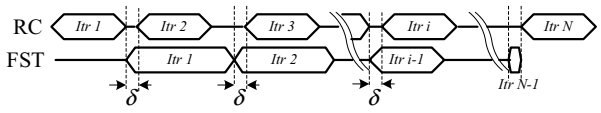


Fig. 4. Timing Diagram of Concurrent Root Computation and Fast Shift Transfrom

in [1]. Since we only deal with bivariate polynomial of Y-degree 4, at most 4 copies of bivariate polynomial coefficient buffer and corresponding FST engines are needed. In Figure 5, the superscript $(i)$ in $Q_j^{(i)}(X,Y)$, $Q_j^{(i)}(0,Y)$ and $\gamma_{j,j'}^{(i)}$ indicates the iteration level, the subscript $j$ identifies the bivariate polynomial at the iteration level, and the 2nd subscript $j'$ in $\gamma_{j,j'}^{(i)}$ is used as root index. For example $\gamma_{1,1}^{(i)}$ refers to the 2nd root found from solving equation $Q_1^{(i)}(0,Y)$ at iteration level $i$. There are 3 types of equation solvers in our architecture, namely, linear, quadratic and quartic equation solvers. As mentioned in Subsection II-A, the quartic equation solver unit can be configured to compute roots for lower degree polynomial equations. Though it is more area efficient to solely use the quartic equation solver to handle all polynomial equations of degree lower than 5, applying the quartic polynomil equation solver to linear polynomial is certainly an "overkill" and causes unnecessary delay. In addition, our simulations indicate that, with a high probability, only linear equations arise in subsequent iteration levels. Thus a linear equation solver is used as a "slave" engine to the quartic equation solver. By doing so, the worst-case factorization latency is not improved, but the average factorization delay is greatly reduced. The same argument applies to the linear equation solver bundled with the quadratic equation solver in Figure 5. In summary, a total of 4 linear equation solver, 1 quadratic equation solver and 1 quartic equation solver is used in our architecture. For linear equation solver, the RC1 architecture of [7] can be used. As one can see, the 2 extra linear equation solvers only cost 2 $\mathbb{F}_{2^p}$ inverters and multipliers, 2 2:1 MUXes and 2 p-bit registers. The root condition check block associated with the quartic equation solver takes the polynomial coefficients ($a$, $b$ and $c$), solutions to the binary linear equation array, etc., as inputs and generates control signal to route the output of the quartic equation solver to appropriate places. A similar root condition check block is used for the quadratic equation solver as well. In addition, the root MUX controller block is used to generate controlling signals for the MUXes that select the roots at the input of the 4 FST engines. In addition, the 4 root buffers store all possible factorization output sequences.

Since only polynomial coefficients corresponding to the monomials with zero X-degree, i.e., coefficients of $Q(0,Y)$, are needed for root computation, root computation for the next iteration can start immediately after those coefficients are available, while the rest of the polynomial coefficients are being updated. This concurrency in the root finding and FST processes can significantly reduce the latency associated with the root-finding procedure. Actually the latency contribution from root computation step can be completely discounted, except for the very first iteration and for iterations when the polynomial update takes fewer clock cycles than the root computation process. The concurrent RC (root computation) and FST operations can be illustrated with the timing diagram of Figure 4. In the figure, $\delta$ represents the number of clock cycles the FST engine takes to produce coefficients of $Q(0,Y)$ for the next iteration. A total of $N$ factorization iterations are assumed, where initially FST takes more clock cycles than RC per iteration and the trend is reversed in later iterations. This effect will be explained in more detail in Section IV.

It should be emphasized that in our new architecture, each of the 4 FST engine is tied to a polynomial coefficient buffer. Compared to the architecture of [7], where a large number of MUXes and DeMUXes are used in the root and polynomial scheduling and de-scheduling blocks to route polynomial coefficients from polynomial buffers to FST engines, our new architecture only needs to route appropriate roots from the equation solvers to FST engines, thus significantly reduces MUX consumption. At each iteration of the factorization procedure, appropriate roots are routed to corresponding FST engines and bivariate polynomial coefficients at the output of the FST engines are stored back to the same buffer, where they have been read from. Our factorization architecture utilizes a routing scheme that has the following properties:

- At the beginning of the factorization procedure, the coefficients of the bivariate polynomial $A(X,Y)$ are "broadcast" to all coefficient buffers.
- In the ensuing iterations, roots are routed among the FST engines based on their conditions, such as number of roots found and the order of each root, etc.

The existence of such a routing algorithm is guaranteed by Corollary 6.3 of [6] that if a root of order $r$ is found at iteration $i$, the degree of corresponding $Q(0,Y)$ in the ensuing iteration can not be larger than $r$. The implementation of the routing algorithm is feasible because of the precise knowledge of the root conditions from the direct root computation method given in Section II. A switch can be designed properly such that appropriate roots appear at the 4 output ports of the quartic equation solver.

## IV. Example of an (458, 410) Reed-Solomon Code over $F_{2^{10}}$

As an illustrative example, the new factorization architecture is applied to soft-decision decoding of an $(458, 410)$ RS code defined on $\mathbb{F}_{2^{10}}$. This code is used in some magnetic recording products. Throughout this section, without specific mentioning, all logic gates are assumed to be 2-input gates.

### A. Algorithm-Level Factorization Complexity

As shown in [5], the re-encoding and coordinate transformation technique also significantly reduces factorization complexity for high-rate RS codes, since at most $2\Delta$ iterations are needed in the factorization process, where $\Delta$ is the maximum number of errors to be corrected in the received hard-decision vector. Otherwise, at least $k$, the number of information symbols in a codeword, iterations are required for the factorization algorithm. According to our simulations carried out for the $(458, 410)$ RS code in a binary AWGN channel, as many as 32 symbol errors, 8 more than a hard-decision decoder's error-correcting capability, can be corrected by the soft-decision decoder at codeword error rate of $10^{-6}$. Thus for practical applications of soft-decision decoding to this RS code, we may assume that $\Delta = 32$, i.e. a total of 64 iterations are required for the factorization procedure.

### B. Hardware Complexity and Factorization Latency Estimate

In this section, we provide an area and latency estimate of our factorization architecture. For area estimate, the gate counts of all building blocks shown in Figure 5, except the controller blocks, are given. The critical path of our factorization architecture is determined by the critical path of the matrix reduction block mentioned in Section II. In this case, there are 6 AND gates, 5 OR gates, 1 inverter and 1 MUX in the critical path, which is comparable to the critical path in earlier designs [1][7]. Necessary pipelining is implemented for all blocks that have longer critical paths. A summary of gate counts and critical paths of all building blocks, except the controllers, are given in Table II.

With appropriate pipelining, transforming a general quartic equation to an affine quartic polynomial takes 6 clock cycles. Constructing the matrix $\underline{\overline{M}}$ needs 10 clock cycles. Matrix reduction takes 20 clock cycles. In addition, it takes 4 clock cycles to route appropriate roots to the output ports. Thus a total of $6+10+20+4 = 40$ clock cycles are required to solve the quartic equation and route the appropriate roots to the FST engines.

We now present a worst-case latency estimate for the new factorization architecture. Since, at any iteraton level, polynomial update only needs to be applied to the coefficients required for root computation in future iterations, the number of clock cycles required for polynomial update decreases linearly. It can be shown that up to 7 clock cycles

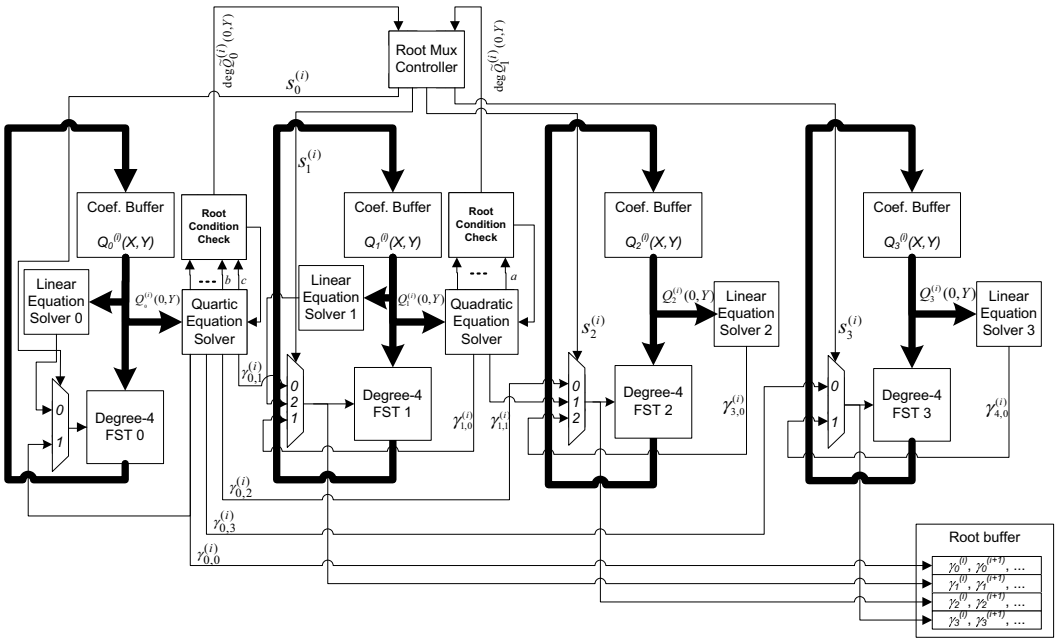


Fig. 5. Factorization architecture for bivariate polynomial of Y-Degree 4

TABLE II
GATE COUNTS AND CRITICAL PATH FOR THE BUILDING BLOCKS IN FACTORIZATION ARCHITECTURE

| Unit | Area | Critical Path |
|---|---|---|
| Converting General Quartic Polynomial to Affine Quartic Polynomial | 373XOR+260AND +36OR+5INV +90MUX+100REG | 7XOR+1AND |
| Matrix Construction | 27XOR+130REG | 3XOR |
| Matrix Reduction | 90XOR+335AND +118OR+20INV +209MUX+110REG | 6AND+5OR +1INV+1MUX |
| Linear Equation Solver | 265XOR+260AND +36OR+5INV +10MUX+10REG | 6XOR+1AND |
| Quadratic Equation Solver | 127XOR+335AND +118OR+20INV +299MUX+240REG | 6AND+5OR +1INV+1MUX |
| Quartic Equation Solver | 1265XOR+1570AND +416OR+65INV +758MUX+450REG | 6AND+5OR +1INV+1MUX |
| FST Engine | 567XOR+500AND +260REG | 6XOR |
| Equation Solver to FST MUXes | 60MUX | – |
| total | 4720XOR+4945AND +678OR+105INV +1157MUX+1770REG | |

are required for the FST engines to generate $Q(0, Y)$ for the next iteration level. Thus in the worst case, polynomial update takes $64+7 = 71$ clock cycles at iteration level $i = 0$ and requires 7 clock cycles for the last iteration. Since root computation and routing by the quartic equation solver takes 40 clock cycles, it can completely overlap with polynomial update from iteration level $i = 0$ up until iteration level $i = 24$. For the rest 38 iterations, each iteration needs $7 + 40 = 47$ clock cycles as solving quartic equation dominates the total delay. Thus the worst case clock cycle count can be estimated as $40+((7+64)+(7+40))\times 25/2+(7+40)\times 38 = 3301$. If exhaustive root search based architectures [2][7] are applied, without any overlap between root computation and polynomial update, the worst-case latency is at least $1024+((1024+7+64)+(1024+7+1))\times 63/2 = 680245$ clock cycles.

Our simulations indicate that high order roots are very rare in practice, and with a very high probability, roots of order 1 are the only roots from the initial root computation. In this case, root computation takes only 1 clock cycle with the linear equation solver, from iteration level $i = 2$ and onwards. Thus it takes $40 + ((4 + 63) + (4 + 1)) \times 63/2 = 2308$ clock cycles to finish factorization procedure most of the time.

## V. CONCLUSION

A novel architecture based on direct root computation is proposed to speed up the factorization process of algebraic soft decision decoding of Reed-Solomon codes. Even though direct root computation can only be applied to bivariate polynomials with Y degree lower than 5, it is sufficient for most practical applications of algebraic soft-decision decoding. With the new architecture, there is only a small variation in decoding latency and the worst-case latency is significantly reduced compared to prior works. Due to precise knowledge of root orders from direct root computation, there is no need to multiplex polynomial coefficients to multiple parallel polynomial update (FST) engines, which can cost a large amount of MUXes. Thus the new architecture is more area efficient as well.

## REFERENCES


[1] A. Ahmed, R. Koetter, and N. Shanbhag, "VLSI architectures for soft-decision decoding of Reed-Solomon codes," *Proc. ICC2004*, Paris, France, June 2004.

[2] A. Ahmed, R. Koetter and N. R. Shanbhag, "Reduced Complexity Interpolation for Soft-Decoding of Reed-Solomon Codes," *Proc. IEEE Symp. Inform. Theory*, Chicago, USA, Jun. 2004, pp. 385.

[3] E. R. Berlekamp, *Algebraic Coding Theory*, New York:McGraw-Hill, 1968.

[4] R. Koetter and A. Vardy, "Algebraic Soft-Decision Decoding of Reed-Solomon Codes," *IEEE Trans. Inform. Theory*, vol. 49, no. 11, pp. 2809-2825, Nov. 2003.

[5] R. Koetter, J. Ma, A. Vardy, and A. Ahmed, "Efficient interpolation and factorization in algebraic soft-decision decoding of Reed-Solomon codes," *IEEE Int. Symp. Inform. Theory*, Yokohama, Japan, July 2003.

[6] R.M. Roth and G. Ruckenstein, "Efficient decoding of Reed-Solomon codes beyond half the minimum distance," *IEEE Trans. Inform. Theory*, vol. 46, pp. 246–258, 2000.

[7] X. Zhang and K. Parhi, "Fast Factorization Architecture in Soft-Decision Reed-Solomon Decoding," *IEEE Trans. VLSI Systems*, vol. 13, No. 4, pp. 413–426, April 2005.