Modeling a Fully Scalable Reed-Solomon Encoder/Decoder over GF(p^m) in SystemC

André Sülflow Rolf Drechsler
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
Email: {suelflow,drechsle}@informatik.uni-bremen.de

Abstract

In this paper we describe how to model arithmetic circuits over $GF(p^m)$ in SystemC*. An extension of a $GF(2^m)$ multiplier is presented to support $GF(p^m)$ arithmetic as well.*

A full integration in the simulation environment is discussed and the proposed solution can be fully synthesized down to hardware. This finds application in e.g. cryptographic systems.

As a case study a Reed-Solomon encoder/decoder system was developed with full $GF(p^m)$ encoding/decoding capability. It is shown that the modeling of a HW/SW co-design system in SystemC *can improve the speed of simulation by a factor of up to 17.*

1. Introduction

From the beginning of the computer, error detection and correction of transmitted data is an important factor for the operativeness. In the 1960's the *Bose-Chaudhuri-Hocquenghem* (BCH) [2, 10] codes were developed for mostly *binary* purposes. An extension to this is the *non-binary Reed-Solomon-Code* (RS) subclass, which has the advantage of transmitting symbols of multiple bit width and the possibility of error detection and correction. The *RS-codes* are widely used and have recently found application in e.g. communication systems or for DVD players. Thus, there is a need for developing efficient hardware simulation tools for *RS-codes*.

RS-codes are based on *Galois Fields* (GF) [2, 10], also known as *Finite fields*, typically over $GF(2^m)$. In this paper the more general approach over $GF(p^m)$ is proposed. The $GF(p^m)$ arithmetic is implemented in *SystemC* [14], where we use the language to model hardware and software. The *SystemC* feature of replacing hardware modules with its software equivalents can help to find suitable bottlenecks in the complete system and to improve the time needed for simulation.

As a case study an implementation of a scalable RS encoder and RS decoder system that is suitable for HW/SW co-design refinement steps is proposed. The granularity of our design is near to *Register-Transfer Level* (RTL). This generates extra overhead for the simulation kernel of *SystemC*, but simplifies the synthesis step.

The paper is structured as follows: Section 2 starts with a short description of the arithmetic in $GF(p^m)$ and provides an overview of the encoding/decoding steps of *RS-codes*. The modeling of our system follows in Section 3 with additionally experimental evaluation in Section 4. Finally, the results are summarized.

2. Preliminaries

2.1. Galois Fields

Galois Fields (GF) [10] are a mathematic construct with a finite number of elements. The number of elements in a field is given by a prime number p and contains the p elements: $0,1,..,p-1$. *Galois Fields* built from a prime number p are denoted as $GF(p)$.

In $GF(p)$ the following inverse element properties hold [10]:

- Inverse element of addition: $\forall a \in GF(p) : \exists a_{add}^{-1} : a + a_{add}^{-1} = 0$
- Inverse element of multiplication: $\forall a \in GF(p), a \neq 0 : \exists a_{mul}^{-1} : a \cdot a_{mul}^{-1} = 1; a = 0 : a_{mul}^{-1} = 0$

And additionally the following basic operations hold [10]:

- Addition: $add(a,b) = (a+b)\ mod\ p$
- Multiplication: $mul(a,b) = (a \cdot b)\ mod\ p$
- Subtraction: $sub(a,b) = add(a, mul(b, p-1))$
- Division: $div(a,b) = mul(a, b_{mul}^{-1})$

An extension of $GF(p)$ is $GF(p^m)$ with $m \in \mathbb{N}$. $GF(p^m)$ forms a polynomial of degree $m-1$, with coefficients in $GF(p)$. E.g. $1x^2 + 1x + 0$ is an element of $GF(2^3)$.

In $GF(p^m)$ a *primitive irreducible polynomial* F of degree m is defined for operations to ensure the finiteness. Irreducible means that the *Greatest Common Divisor* (GCD) of F and all polynomials of $GF(p^m)$ is 1. Additionally for each F there exists a *generator element* α that creates the field by building the powers of α. The p^m elements of $GF(p^m)$ are $0, \alpha^0, \alpha^1, .., \alpha^{p^m-2}$.

The operations *add*, *mul*, *sub*, *div* and the inverse elements of $GF(p)$ are defined for $GF(p^m)$, too: Addition in $GF(p^m)$ is performed by splitting an element into the m components followed by a component-wise addition in $GF(p)$ [13, 12]. Multiplication of two elements $a,b \in GF(p^m)$ is a multiplication of a and b modulo F:

$(a \cdot b)$ *mod* F. If a and b are given in the form $a = \alpha^k$ and $b = \alpha^l$ multiplication is an addition of the powers followed by a modulo operation: $\alpha^k \cdot \alpha^l = \alpha^{(k+l) \, mod \, (p^m-1)}$ [13, 12].

> **Example:** Consider $GF(2^3)$ and $F = x^3 + x^1 + 1$ with *generator element* $\alpha = 2$. By taking the successive powers of α all elements, except the zero element, of $GF(2^3)$ can be created: $2^0 = 1, 2^1 = x, 2^2 = x^2, ..., 2^6 = x^2 + 1$.
>
> Adding the elements $a = x^2 + x + 1$ and $b = x$ results in $a + b = (1+0)x^2 + (1+1)x + (1+0) = x^2 + 1$.
>
> Multiplication of a and b is computed as: $(a \cdot b)$ *mod* $F = (x^3 + x^2 + x)$ *mod* $(x^3 + x + 1) = x^2 - 1 = x^2 + 1$. Regarding the *generator element* with $a = \alpha^5$ and $b = \alpha^1$, the operation is defined as: $\alpha^5 \cdot \alpha^1 = \alpha^{5+1} = \alpha^6 = x^2 + 1$.

2.2. Reed-Solomon-Codes

Reed-Solomon-Codes (RS-Codes) are a *non-binary* subclass of the *Bose-Chadhuri-Hocquenghem (BCH)* block coding codes [2, 10]. *BCH* codes are multiple-error-correcting codes and are widely used for data-transmission control and repair. The *RS* code symbols are elements of $GF(p^m)$ with $p \geq 2$ and $m \geq 1$. This makes them *non-binary*. Typically codes over $GF(2^m)$ are used, but in the following the discussion is not restricted to $p = 2$.

RS-Codes are specified by two parameters N and K which determine the maximum number of correctable symbol errors $t = \frac{N-K}{2}$. $N = p^m - 1$ is the total number of symbols in a transmission block, where $K = p^m - 1 - 2t$ symbols are the information symbols to be transmitted and the remaining $N - K$ symbols are used for the parity. The *RS-Code* is a *maximum distance code* with a minimum distance of $d_{min} = N - K + 1$. Thus, it is optimal for data-transmission [2, 10].

2.2.1 Encoder

An encoder has to transmit a block of K symbols of $GF(p^m)$ with error correction. This information block can be described as an information polynomial $i(x)$ with the K symbols as coefficients [13, 12, 11]:

$$i(x) = i_{K-1}x^{K-1} + i_{K-2}x^{K-2} + ... + i_0 \tag{1}$$

For encoding $i(x)$ the equation

$$c(x) = i(x) \cdot x^{2t} - (i(x) \cdot x^{2t}) \; mod \; g(x) \tag{2}$$

is used. The degree of $c(x)$ (the codeword polynomial) is $N-1$; $i(x)$ has degree $K-1$ and $g(x)$ (the *generator polynomial*) has degree $2t$.

The first part $(i(x) \cdot x^{2t})$ is a simple up-shift operation of the information polynomial $i(x)$. The second part $((i(x) \cdot x^{2t}) \; mod \; g(x))$ describes the generation of the $2t = N - K$ parity symbols and is the *remainder* of the operation. Thus, by subtracting the *remainder* from $i(x) \cdot x^{2t}$ the result $c(x)$ holds $c(x) \; mod \; g(x) = 0$.

The *generator polynomial* $g(x) = g_{N-K}x^{N-K} + ... + g_0$ is generated by the equation:

$$g(x) = (x - \alpha^{j_0})(x - \alpha^{j_0+1})...(x - \alpha^{j_0+2t-1}) \tag{3}$$

For j_0 any integer value with $j_0 \geq 1$ can be chosen [2, 12], but typically $j_0 = 1$ is used. In the following $j_0 = 1$ is assumed. One important property of Equation (3) for the decoding procedure is the evaluation to zero for $x = \alpha^1, ..., \alpha^{2t}$. Thus, Equation (2) evaluates to zero for all $x = \alpha^1, ..., \alpha^{2t}$ with $a \; mod \; 0 = a$, too.

2.2.2 Decoder

Assume a transmitted polynomial:

$$r(x) = r_{N-1}x^{N-1} + r_{N-2}x^{N-2} + ... + r_0 \tag{4}$$

The error between $c(x)$ and $r(x)$ can then be measured as:

$$e(x) = c(x) - r(x) \tag{5}$$

If there are no errors, $e(x)$ is zero and $r(x) = c(x)$ holds. Otherwise there is at least one data-transmission error somewhere in $r(x)$ [13, 12, 11].

To test $r(x)$ for errors the so called $2t$ *syndromes* have to be calculated:

$$S_i = r(\alpha^i) \; \forall i = 1, .., 2t \tag{6}$$

If all S_i *syndromes* are zero, it can be assumed that $r(x)$ has no errors and the data-transmission was correct. This uses the evaluation to zero property of Equation (2) described above. The first K $(N, .., N-K)$ coefficients are then the coefficients of the information polynomial $i(x)$.

If there is at least one *syndrome* unequal to zero, then a data-transmission error occurred. To restore $c(x)$ from $r(x)$ the following steps have to be performed:

1. Find error-location(s)
2. Find error-magnitude for each error-location
3. Perform error-correction

Suppose there are $v, 1 \leq v \leq t$ errors in $r(x)$. Then the error polynomial $e(x)$ can be written as

$$e(x) = \sum_{l=1}^{v} e_{j_l}x^{j_l} = e_{j_1}x^{j_1} + ... + e_{j_v}x^{j_v}, \tag{7}$$

where j_l specifies the error-location searched and e_{j_l} is the magnitude to be added to r_{j_l} to get c_{j_l}.

Find error-location(s):

To find the v error-location(s), the smallest length vector Λ that satisfies

$$S_j + \sum_{k=1}^{t} S_{j-k}\Lambda_k = 0 \qquad j = t+1, ..., 2t \tag{8}$$

has to be found [2].

The coefficients of the vector Λ are then used to form the *error-locator polynomial* [2] with a maximum degree of t:

$$\Lambda(x) = \prod_{l=1}^{v}(1 - x\alpha^{j_l}) = 1 + \Lambda_1 x^1 + ... + \Lambda_v x^v \qquad (9)$$

In the literature often the *Berlekamp-Massey Algorithm* [2, 1] is used to find $\Lambda(x)$, but the *Extended Euclidean Algorithm* [2] can also be applied. The term $1 - x\alpha^{j_l}$ in Equation (9) evaluates to zero for $x = \alpha^{j_l{}^{-1}}$, due to the inverse element property of *Galois Fields*. The roots x of $\Lambda(x)$ are then the inverses of the *error-locators* a^{j_l}.

To find the v roots the *Chien search* algorithm [3] is applied. This brute-force algorithm evaluates $\Lambda(x)$ for all $x = 0, .., p^{m-1}$ and returns as the result the v roots which are evaluating to zero.

Find error-magnitude for each error-location:

To find the *error-magnitude* e_{j_l} the *Forney Algorithm* [5] is used. Therefore, the *syndrome polynomial* $s(x)$ is formed with the syndrome coefficients calculated above[12] :

$$s(x) = S_{2t}x^{2t} + S_{2t-1}x^{2t-1} + ... + S_1 x + 0 \qquad (10)$$

The *error- magnitude polynomial* $\Omega(x)$ is then defined as:

$$\Omega(x) = (\Lambda(x) \cdot (s(x) + 1)) \; mod \; x^{2t+1} \qquad (11)$$

The *error-magnitudes* e_{j_l} are calculated by:

$$e_{j_l} = \frac{-\alpha^{j_l} \cdot \Omega(\alpha^{j_l{}^{-1}})}{\Lambda'(\alpha^{j_l{}^{-1}})} \qquad (12)$$

where α^{j_l} and $\alpha^{j_l{}^{-1}}$ are computed by the *Chien search* algorithm. $\Lambda'(x)$ is the *derivative* of $\Lambda(x)$, computed by:

$$\Lambda'(x) = \sum_{j=1}^{v}(j\Lambda_j)x^{j-1} \qquad (13)$$

Perform error-correction: After all the steps above, there are:

1. v $\Lambda(x)$-roots: $\alpha^{j_1{}^{-1}}, \alpha^{j_2{}^{-1}}, .., \alpha^{j_v{}^{-1}}$
2. and the error-magnitudes $e_{j_1}, e_{j_2}, .., e_{j_v}$

The inverse element of multiplication of the root $\alpha^{j_l{}^{-1}}$ is than the searched *error-locator* α^{j_l}. The determination of the error-location j_l of α^{j_l} is known as *discrete logarithm* problem. But because the number of elements in $GF(p^m)$ is small, it is possible to create a logarithm lookup-table by building the successive powers of α.

Now $e(x)$ is computed by setting the error-magnitude e_{j_l} for location x^{j_l}. The error correction can be applied by adding $e(x)$ to $r(x)$ to get the original $c(x)$.

$$c(x) = e(x) + r(x) \qquad (14)$$

```
void compute ()
{
  //(i_opa * i_opb) % RADIX_p
 int result = i_opa.read()*i_opb.read();
 for (int i = 0; i < (RADIX_p - 2); i++)
 {
   if (result >= RADIX_p)
   {
     result = result - RADIX_p;
   }
 }
 o_result.write(result);
}
```

Figure 1. Multiplication in $GF(p)$

2.3. SystemC

SystemC is a freely available C++ class library [14]. This includes the source-code, what makes it platform independent. With a standard C++-compiler an executable is created for efficient simulation run times.

In *SystemC* the system can be modeled at different levels of abstraction - from transaction, down to RTL. Thus, a step-by-step refinement from software to hardware is possible. *SystemC* defines a set of classes for implementing modules and their interconnections. Hence, it is possible to use and compare different implementations of the same module. A first step to model multi-value circuits in *SystemC* has been proposed in [6].

3. Hardware Modeling

In the following a description of the modeling of $GF(p^m)$ and a *Reed-Solomon encoder/decoder* in *SystemC* is given.

A "pure" hardware and a software system are developed, so that parts of both systems are fully replaceable in *SystemC*. The hardware structure is scalable over $GF(p^m)$. This also holds for the *Reed-Solomon encoder/decoder* hardware.

3.1. Galois Fields

In software addition and multiplication in *Galois Fields* can be implemented by using the C++ standard operators $+, \cdot, \%$ for the `int` basic type or to use addition and multiplication tables. But especially the last approach is often not sufficient for scalable hardware simulations. In the literature different implementations for adders and multiplier over $GF(2^m)$ have been proposed e.g. [2, 10, 4, 7]. In the following an extension of the one in [7] is presented for modeling a $GF(p^m)$ multiplier.

3.1.1 $GF(p)$

The result of addition in $GF(p)$ $((a+b)\; mod\; p)$ is a simple integer addition plus an optional required modulo operation. The modulo operation is required, if the result exceeds $p-1$. This is done by subtracting p from the result. The basic type `int` is used for implementing the behavior.

Multiplication in $GF(p)$ $((a \cdot b)\; mod\; p)$ is similar to addition. Because a and b have a max value of $p-1$, the multiplication result has a maximum of $(p-1)^2$. Thus, p has to be subtracted from the result as long as the result is larger

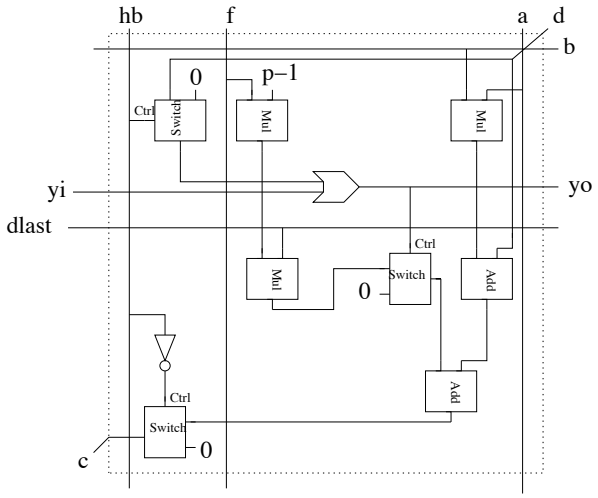

Figure 2. Multiplier cell for $GF(p^m)$

than $p-1$. This needs a maximum of $p-2$ subtractions and is shown in Figure 1.

Note, for $GF(2)$ addition is a simple `XOR`-operation, while multiplication is an `AND`-operation. For $p > 2$ an implementation that uses multiple bit-encoding for the same element of $GF(p)$ can be used. For example in $GF(3)$ [9] uses a two-bit representation for each element (0=00,01; 1=10; 2=11). For this, a multiplication in $GF(3)$ is a simple `AND`-operation of the first bit in combination with an `XOR`-operation of the second bit.

3.1.2 $GF(p^m)$

Each element of $GF(p^m)$ is a vector of m elements of $GF(p)$. These are the coefficients of polynomials. Therefore, addition of two elements is a component wise addition in $GF(p)$ and involves m additions over $GF(p)$ in parallel.

For multiplication the $GF(2^m)$ multiplier proposed in [7] as an extension of [4] is used. This array-type multiplier is used e.g. for *Digital Signal Processing (DSP)* and takes one clock-cycle for the multiplication. It is build of $(m+1) \cdot m$ multiplier cells. Each of these cells performs one calculation step in parallel. The structure of the original multiplier cell was changed by adding two multipliers and one extra input and replacing some $GF(2^m)$ specific gates by its $GF(p^m)$ counterpart. The modified structure for $GF(p^m)$ is shown in Figure 2. The adders and multipliers in the cell perform computation over $GF(p)$. The additional input `dlast` is connected with the input `d` of the left most cell in the current row. This satisfies the modulo F operation, if required.

For the *Forney Algorithm* [5] and the *error-correction* step an inverter- and a log-module in $GF(p^m)$ are needed. Both of them are implemented as software in our system. So this software version simulates the behavior of a static lookup-table in hardware and has a response time of one clock-cycle. An alternative implementation in hardware for the inverter-module could be a one multiplier implementation and the use of brute-force to get the inverse element. But it takes in worst case p^m clock-cycles ($p^m - 1$ for brute-force plus 1 for the result). Computing in parallel for all p^m elements takes p^m multipliers. Because of the high cost of a multiplier the scaling is very expensive.

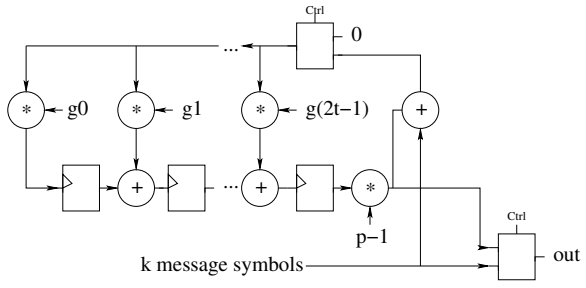

Figure 3. LFSR Reed-Solomon Encoder for $GF(p^m)$

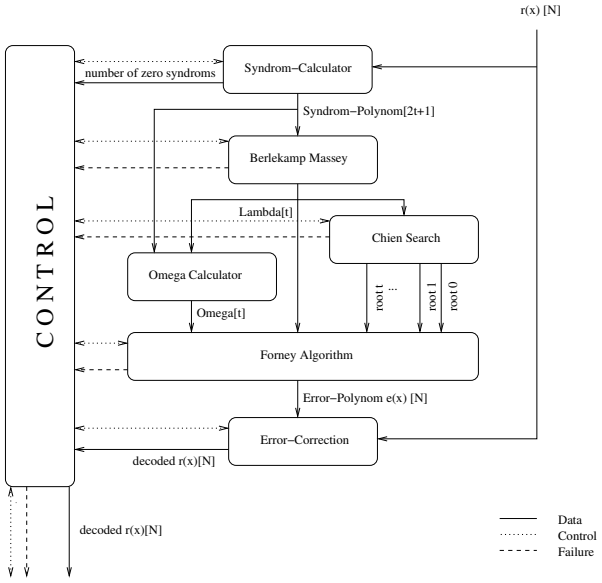

Figure 4. A Reed-Solomon Decoder over $GF(p^m)$

3.2. Reed-Solomon Encoder

Encoding an information polynomial $i(x)$ is described in Equation (2) and can be efficiently implemented with a *Linear Feedback Shift Register* (LFSR). An LFSR implementation is shown in Figure 3. The $p-1$ multiplier is an extension for $GF(p^m)$ encoding and is not necessary for $GF(2^m)$ symbols. The $g0,..,g(2t-1)$ inputs are the coefficients of the generator polynomial.

During the first K clock-cycles the K message symbols of $i(x)$, starting at the highest degree, are shifted into the encoder and can be read at the out-port. After K clock cycles the registers contain the remainder polynomial and have to be shifted out of them. For this, the two switches are changed from true to false and the remainder is sent clockwise to the out-port. Altogether, N symbols are sent to the out-port and form the codeword polynomial $c(x)$ with highest degree $N-1$ first: $i(x)$ (K symbols) and the *remainder* ($N-K$ symbols).

3.3. Reed-Solomon Decoder

Our decoder consists of six submodules which are connected as shown in Figure 4. Each of these modules is

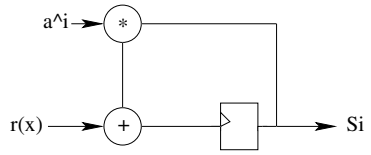

Figure 5. Calculates S_i by evaluating $r(x)$ with α^i

implemented as a state-machine, but all arithmetics over $GF(p^m)$ are computed by internally connected modules like adders, multipliers and inverters.

There are three different types of connection-signals:

1. `Data [z]`: z coefficients of type $GF(p^m)$, equivalent to a polynomial of degree $z-1$. Note: In a complete hardware environment this class is used for data-transmission over *SystemC*-signals only. And in a mixed hardware/software environment it is possible to make a C++-method call.
2. `Control`: Two Boolean signals. One for starting a module and the other for sending/receiving a computation finished.
3. `Failure`: Boolean failure signal, true if a failure occurred

Table 1. Used number of arithmetic modules over $GF(p^m)$

Module	Mul	Add	Reg	Inv	Log
Syndrom Calculator.	2	1	1	0	0
Berlekamp-Massey	3t+2	2t	2t	1	0
Omega Calculator.	1	1	1	0	0
Chien Search	1	1	1	0	0
Forney	4	2	2	2	1
Error-Correction	0	N	0	0	0

In Table 1 the type and the corresponding number of gates used for the submodules of the decoder are shown. Only the modules *Berlekamp-Massey* and *Error-Correction* are dependent on the choice of the parameters t or N. All the other modules have a constant numbers of gates. This is an important factor for the scalability.

In the following the most important modeling parts of the decoder module are briefly described.

The syndrome calculator uses the hardware of Figure 5 to calculate $S_i = r(\alpha^i)$. This implementation takes N clock-cycles for evaluation and follows the Horner equation [8]:

$$r(x) = (((r_{N-1}x) + r_{N-2})x + ...) + r_0 \qquad (15)$$

Altogether it takes $2t \cdot N$ clock-cycles for calculating the $2t$ syndromes. This form of polynomial evaluation is also used in the *Forney Algorithm* module, where polynomials of degree t are evaluated.

The Figures 6 and 7 show the modeling of the discrepancy calculator and the temporary needed polynomial $T(x)$ calculation as part of the *Berlekamp-Massey* module. It uses a state-machine for the computation as described in [2].

The modules *Omega Calculator*, *Chien Search*, *Forney Algorithm* and *Error Correction* are straight-forward implemented (see Section (2.2.2)). The details are left out due to page limitation.

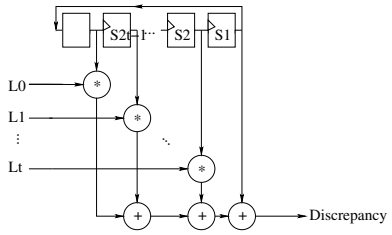

Figure 6. Discrepancy calculator

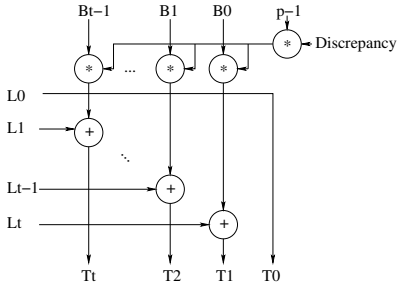

Figure 7. Hardware for computing $T(x) = \Lambda(x) - discrepancy \cdot X \cdot B(x)$

4. Experimental Evaluation

All techniques described above have been implemented in *C++/SystemC*. The experiments have been carried out on a AMD Athlon 64 3700+ (2.2 GHz) with 1 GByte running Linux. The focus is on the following criteria:

1. Simulation run time
2. Scalability of the design
3. HW/SW co-designs of the Reed-Solomon Encoder/Decoder

The results of regarding these criteria can e.g. be used for refinement and optimization steps. Especially the run time can show limits for the size or granularity of the *SystemC* design.

The first section focuses on the performance of $GF(p^m)$ arithmetic in hardware in comparison to a software model in *SystemC*. Then the evaluation of these modules as part of the Reed-Solomon encoder/decoder architecture is described.

4.1. Galois Fields

In this section the focus is on the influence of the chosen values of p and m on the run time. For the experiments the adder and multiplier implementations over $GF(p^m)$ in hardware and software are used, respectively. Each of the four hardware- and software-modules is tested for 1 million clock-cycles with random values for the two operators and the run time is measured in CPU seconds.

The results in Table 2 show for the adders in hardware differences with a maximum of two seconds in comparison to the software implementation. The choice of p shows no significant influence on the run time. Thus, the only important factor for scalability is the choice of m.

The run time of multiplication in hardware over $GF(2^m)$ grows non-linear, while in software a linear growth was

Table 2. Simulation time in $GF(p^m)$

	Hardware		Software	
$GF(p^m)$	Add	Mul	Add	Mul
2^3	3.58	12.16	3.30	5.03
2^4	4.12	24.59	3.76	6.82
2^5	4.81	47.75	4.28	8.66
2^6	5.23	73.37	4.68	10.10
2^7	5.81	119.62	5.01	11.74
2^8	6.31	226.29	4.49	13.88
3^6	5.51	125.67	3.75	12.46
5^4	4.32	47.94	3.06	8.24
7^3	3.83	24.82	2.72	5.78
17^2	3.08	9.23	2.23	4.20

measured. Multiplication in $GF(p^m)$ for $p > 2$ needs a similar run time as for $GF(2^{m+1})$.

4.2. Reed-Solomon Codes

For the Reed-Solomon encoder/decoder the simulation times of a HW/SW co-design in comparison to a "pure" hardware implementation are studied. The later one is denoted as HW in the following. Here, especially the arithmetic over $GF(p^m)$ with $2t$ adders and $2t+1$ multipliers for the encoder and $5+N+2t$ adders and $10+2t$ multipliers for the decoder (see Table 1) is very complex. In the HW/SW co-design setting the modules for $GF(p^m)$ arithmetic were replaced with a software solution. The hardware specific parts of the *SystemC* modules were substituted by a software method call, where the underlying addition and multiplication algorithm is equivalent. From the run time (t) and the number of encoded messages (n), the time (t_m) for encoding one message is computed with the equation: $t_m = t/n$. Considering an evaluation in $GF(p^m)$, then $N = p^m - 1$ and $K = N - 2 \cdot t$ are chosen. $t = 2$ and a clock-cycle rate of 100000 was chose for all settings.

Table 3. Time for encoding/decoding in $GF(p^m)$

	Encoder			Decoder		
$GF(p^m)$	n	t_m HW	t_m HW/SW	n	t_m HW	t_m HW/SW
2^3	10000	< 0.01	< 0.01	746	0.03	0.01
2^4	5555	< 0.01	< 0.01	450	0.14	0.02
2^5	2941	0.01	< 0.01	251	0.65	0.08
2^6	1515	0.04	< 0.01	133	3.14	0.27
2^7	769	0.17	< 0.01	68	15.07	1.35
2^8	387	0.74	0.25	34	106.65	6.56
3^6	136	0.72	0.07	12	426.00	43.83
5^4	159	0.28	0.04	14	129.57	20.07
7^3	289	0.08	0.01	26	15.15	4.69
17^2	344	0.02	< 0.01	31	4.29	1.90

Table 3 shows the results for HW vs. a co-design modeling. By increasing m and constant p the number of encoded/decoded messages per 100000 clock-cycles decreases. This is the same for an increasing of parameter p with constant m. This gap is a result of the different choices for N and K and the resultant extra overhead necessary for e.g. *Chien-Search* algorithm. The use of a HW-SW model results in speed-ups ranging from 4 to 17.

The experiments were carried out for different values for N and K. Since it is possible to build *shortened* Reed-Solomon encoder/decoder [2, 10], the values given are upper bounds on the run time. Thus, the decoding of the same number of symbols in $GF(2^8) = GF(256)$ is more than 25 times slower than in $GF(17^2) = GF(289)$. Of course a higher bit width for data-transmission is needed, but for e.g. mobile systems this could bring significant speed-up.

5. Conclusions and Future Work

In this paper we presented an approach to model arithmetic circuits over $GF(p^m)$ in SystemC. As a case study, a Reed-Solomon encoder/decoder system has been investigated. Alternative implementations have been discussed resulting in significant speed-ups in simulation time, if adders and multipliers are modeled in software.

The usage of $GF(17^2)$ multiplication arithmetic instead of $GF(2^8)$ can increase the run time of Reed-Solomon encoding and decoding by a factor of 25. Of course, this needs a higher bit width for data-transmission, but e.g. for mobile systems that would bring a significantly improvements.

For future work the successfully used multiple bit-encoding idea in $GF(3^m)$ [9] could be investigated for $p > 3$. But therefore efficient implementations for multiplication and addition are needed.

References

[1] E. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, 1968.

[2] R. Blahut. *Theory and Practice of Error Control Codes*. Addision-Wesley, 1984.

[3] R. Chien. Cyclic decoding procedure for the Bose-Chaudhuri-Hocquenghem codes. *IEEE Transactions on information theory*, 10:357–363, 1964.

[4] W. Drescher and G. Fettweis. VLSI architectures for multiplication in GF(2^m) for application tailored signal processors. In *Workshop on VLSI Signal Processing IX*, pages 55–64, 1996.

[5] G. Forney. On decoding BCH codes. *IEEE Transactions on information theory*, 11:549–557, 1965.

[6] D. Große, G. Fey, and R. Drechsler. Modeling multi-valued circuits in SystemC. In *Int'l Symp. on Multi-Valued Logic*.

[7] D. Hoffmann and T. Kropf. Verification of a GF(2^m) multiplier-circuit for digital signal processing. Technical Report 22, University of Karlsruhe, 1998.

[8] W. Horner. A new method for solving equations of all borders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, pages 308–335, 1816.

[9] T. Kerins, E. Popovici, and W. Marnane. Fully paramaterisable galois field arithmetic processor over GF(3^m) suitable for elliptic curve cryptography. In *24th International Conference on Microelectronics*, volume 2, pages 739–742, 2004.

[10] S. Lin and D. Costello. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, 1983.

[11] A. Matache. Encoding/decoding reed solomon codes. *http://www.ee.ucla.edu/˜matache/rsc /slide.html*, 1996.

[12] S. S. Shah, S. Yaqub, and F. Suleman. Self-correctin codes conquer noise part2: Reed-solomon codecs. *EDN*, March 15, 2001.

[13] B. Sklar. *Digital Communications: Fundamentals and Applications*. Prentice-Hall, 2001.

[14] Synopsys Inc., CoWare Inc., and Frontier Design Inc., http://www.systemc.org. *Functional Specification for SystemC 2.1*.