Design Space Exploration of a Hardware-Software Co-designed GF(2^m) Galois Field Processor for Forward Error Correction and Cryptography

Wei-Ming Lim[1], M. Benaissa[2]
University of Sheffield
Department of Electronic and Electrical Engineering,
Mappin Street, Sheffield, S1 3JD, United Kingdom
[1]elp00wml@sheffield.ac.uk [2]m.benaissa@sheffield.ac.uk

ABSTRACT

This paper describes a hardware-software co-design approach for flexible programmable Galois Field Processing for applications which require operations over GF(2^m), such as RS and BCH codes, Elliptic Curve Cryptography and the AES. Complexities of flexible implementations of different applications on a same computation architecture can be migrated to software during design time. However, the underlying GF(2^m) arithmetic architecture needs to be designed with software programmability (or reconfigurability) in mind. We describe novel reconfigurable subword parallel GF(2^m) arithmetic architectures designed with an associated instruction set architecture for different applications over GF(2^m) and same applications with differing parameters. Design space exploration is carried out with two simple parameters P and Q which can be changed at design time and will affect the performance of different applications and flexibility of the final implementation. We show implementation results given for an FPGA prototype of the processor and programmed for RS and BCH coding, AES and elliptic curve cryptography with differing parameters. Complexity figures and configuration overheads for subword parallel GF(2^m) arithmetic architectures are also estimated and discussed.

Categories and Subject Descriptors

C.3 [Special-Purpose And Application-Based Systems]: Real-time and embedded systems; B.2 [Arithmetic And Logic Structures]: Design Styles---Parallel

General Terms:

Design, Algorithm, Performance

Keywords

Galois Field Processor, GF(2^m) Arithmetic, Forward Error Control Coding, Reed-Solomon Code, BCH Code, Cryptography, Elliptic Curve Cryptography, Advanced Encryption Standard, Hardware-Software Co-design, Design Space Exploration

CODES+ISSS'03, October 1–3, 2003, Newport Beach, California, USA.

1. INTRODUCTION

GF(2^m) arithmetic has been used extensively in the domains of Forward Error Correction (FEC) Codes and Cryptography. Well known examples include Reed Solomon and BCH Codes for FEC [1], Elliptic Curve Cryptography (ECC) [2] for Public Key Cryptography and lately, the Advanced Encryption Standard [3] (AES) using the Rijndael Algorithm for Private Key Cryptography.

As systems get increasingly complex, more and more effort has being channelled into re-usable implementations, particularly software controlled architectures, by allowing design re-use simply through re-programming. Some examples are described in [4-7]. Here, we define two parameters P and Q: P is the number of parallel arithmetic computation units (multiplication, division and Addition) and Q is the bit size of each unit. This paper will concentrate on the design space exploration of a software driven hardware architecture for applications over GF(2^m) with P and Q as the central design parameters.

A hardware-software co-design approach is described for hardware architectures over GF(2^m) whereby software allows the same hardware to be re-used for different applications. This entails a design space exploration where the requirements of different applications over GF(2^m) are systematically explored, and also the formulation of a hardware architecture to facilitate these applications. This design space can be broadly define into three levels of abstraction as shown in Table 1. The top level determines the global requirements of the specific applications. For example, application area (cryptography or FEC), code rate and error correction capability of a RS or BCH code (N,K), key size of the AES and the curve parameters of ECC.. etc.

The bottom level consists of the basic arithmetic circuits that form the basis of the applications. Choices here include the size and type of the arithmetic for example in terms of field size, irreducible polynomial and basis representation. These are of course influenced by the global requirements. The middle level provides the "bridge" between the top and bottom levels. Usually, this middle level determines the overall structure of a derived architecture and is determined by the primitive operations of an application. For each application, we identify these primitive operations, and this is an important first step towards designing efficient hardware/software architectures. Table 2 shows the requirements of various different operations.

There has been a trend towards using parallel arithmetic computation units driven by software for FEC algorithms as evident in [5, 6], since there is substantial inherent parallelism in these algorithms. The same principle can be applied directly to the

AES. Due to the large field size, parallel arithmetic computations are not considered for ECC as these need multiple arithmetic units. Obviously a simplistic implementation of ECC would be achieved by setting: P = 1 and Q equals the field size over which the ECC is defined. To bridge the gap of an architecture flexible enough for RS codes, BCH codes and the AES is relatively simple, however, the same cannot be said if ECC is to be included with other applications. Although a $GF(2^{163})$ processor designed for ECC can be used directly for $GF(2^8)$ computations for the AES (or RS/BCH codes), it will be highly inefficient as only 8 out of a possible 163 bits are used at any one time. (Assuming the underlying arithmetic units can support variable field sizes and irreducible polynomials). This is one main obstacle towards developing efficient processor architectures for the domain of $GF(2^m)$.

Table 1 : Abstraction Levels

Abstraction Levels	Application	
Algorithm Level	Cryptography	FEC
	AES Diff Key Lengths ECC Diff Curve sizes	RS(N, K) codec BCH (N,K) codec
Primitive Operations	Point Additions/ Doubling over Projective or Affine Co-ordinates	Key Equation Solving Using BMA or Euclidean Algorithm, Systematic Non Systematic Encoding
		Different Polynomial Operations
Arithmetic Level	Variable Field Size, Variety of Bases, Variable Irreducible Polynomials	

Table 2 : Requirements Analysis of Applications over $GF(2^m)$

Application	RS Codes	BCH	AES	ECC
Ranges	3-8 Bits	3-16 Bits	8 Bits	>100 Bits
Primitive Operations	Polynomial		Matrix and Polynomial like	Quadratic Equations

Subword parallelism (SWP) is a concept from computer architecture first introduced by Lee in [8]. Multiple subwords are packed into a word and processed with a single instruction, which can be seen as a form of SIMD (Single instruction Multiple Data). This concept can be extended to the domain of $GF(2^m)$ as well. It can provide the flexibility without loss of efficiency required for a $GF(2^m)$ processor and solves the problem of large data size mismatch for different applications. A SWP $GF(2^m)$ processor is defined as an entity with a data path of m = PxQ bits and can operate in two modes : Single Instruction Single Data (SISD) and Single Instruction Multiple Data (SIMD) mode. A SWP $GF(2^m)$ arithmetic circuit allows the processor to compute P $GF(2^n)$ arithmetic operations ($n \leq Q$) or one $GF(2^n)$ arithmetic operation ($Q < n \leq PxQ$) per instruction. This means by suitable selection of P and Q, it is possible to define a structure that can be utilized efficiently for both large and small field size operations. For e.g, if P = 21 and Q = 8, m = 168. The processor can be used for one large $GF(2^n)$ computation ($n \leq 168$) or 21 parallel smaller $GF(2^n)$ computations ($n \leq 8$), the former useful for ECC and the latter useful for AES, RS and BCH codecs.

The process of design space exploration is very much top down/bottom up. In the very first pass, the top down approach will attempt to identify the requirements of each application at each level as given in table 1. The bottom up approach will attempt to formulate suitable architectures to suit these requirements at each level, and this may take several iterations before a compromise is reached for a given cost function in terms of speed versus area versus power for example. To reach a compromise on all of these requirements plus considerations for flexibility is not a trivial task and further work are needed here. In this paper, we will be concentrating primarily on the requirements issues for the domain of $GF(2^m)$ which is not a common subject in the published literature.

The SWP type architectural structure of the arithmetic circuits of the $GF(2^m)$ processor forms the middle abstraction level in Table 1 in this paper. An Instruction Set Architecture can then be defined over this architecture which allows the processor to compute the primitive operations for different applications.

2. SUBWORD PARALLEL ARCHITECTURES

This section describes the architecture of a Subword Parallel $GF(2^m)$ processor which consists of the SWP $GF(2^m)$ arithmetic circuits.

2.1. SWP $GF(2^m)$ Arithmetic Circuits

We briefly describe how a subword parallel $GF(2^m)$ multiplier can be designed using a simple example for m = 4. The GF multiplication algorithm used here is based on the well known MSB first algorithm operating over a polynomial basis which is outlined in [9] and reproduced briefly below. We need to compute C(y) = A(y).B(y) mod G(y).

Multiplication Algorithm [9]

```
1  C'(y) = all zeros
2  for i = 0 to m-1
3       C'(y) = C'(y) + G(y).c'_m + A(y).b_{m-1}
4       B(y) = y.B(y) mod (y^m)
5       C'(y) = y.C'(y) mod (y^{m+1})
6 end loop;
```

We can break line 3 of the algorithm into bit slices, ($0 \leq k \leq m$):

$$c'_k = c'_k \oplus (g_k \bullet c'_m) \oplus (a_k \bullet b_{m-1})$$

Figure 1 shows a 4-bit bit slices cascaded together. This performs one iteration of the multiplication algorithm. By feeding the intermediate result back into the same structure m times, a m-bit $GF(2^m)$ multiplication can be performed. Notice that as long as the data are MSB justified, the same m-bit structure can be used for $GF(2^n)$ multiplications ($n \leq m$) by varying the number of iterations n. From the algorithm, we can see that at every iteration, b_{m-1} and c'_m are used to determine whether C'(y) is added with A(y) and G(y). We call these global signals.

B(y) is loaded into a shift register and is MSB shifted *n* times during the course of the algorithm. At the i^{th} iteration, the most significant bit of the shift register will be the i^{th} bit of B(y), b_i. Line 4 and 5 of the algorithm is simply the mathematical representation of logical shift with the MSB discarded. Shifting of Line 5 can be hardwired as shown in Figure 1. Suppose the structure in Figure 1 is cut into two parts of 2-bits each, and

additional configuration circuitry (basically multiplexers and basic gates) are added to each 2-bits parts called Logic Units (LU). The configuration circuitry of each LU is controlled by a set of control signals MSBlock and LSBlock. Setting both MSBlock and LSBlock of a LU to '1' "isolates" that particular LU such that all global signals are derived from the same LU, and shifting signals do not cross into other LUs. In fact, both LUs in this situation can be seen as independent 2-bit multipliers. Similarly, setting LSBlock[2] and MSBlock[1] to '0', and setting LSBlock[1] and MSBlock[2] to '1' configures the structure in Figure 2 to behave as if it is a 4-bit multiplier, as the global signals are now multiplexed from the MSB LU (LU 2) and shifting signals are allowed to pass between LUs. In general, we can thus break-up a large M-bits GF(2^m) multiplier into P smaller Q-bits LUs, which essentially are Q-bits GF(2^Q) multipliers. In practice, it will be easier to design a LU of a specific size (Q-bits) and couple as many of them together to achieve the required large field size.

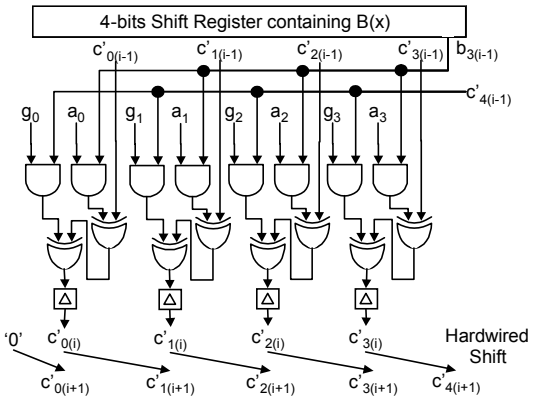

Figure 1: 4-bit GF(2^m) Multiplier

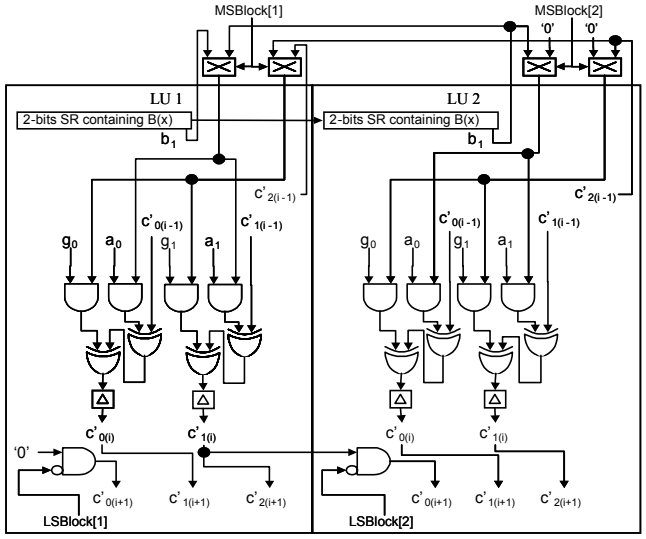

Figure 2: Modified GF(2^m) multiplier with 2 LUs.

The same procedure can be applied to a GF(2^m) division algorithm, although this is not described here it due to length constraints. In particular, Brunner in [10] described a polynomial basis GF(2^m) division algorithm which can be used for modification into the SWP structure as it is regular in structure and computation cycles. This regularity is important as it makes modifications easier. GF(2^m) addition needs no modifications as it only involves bit-wise XOR operations.

2.1.1. Complexity Analysis

Table 3 shows the complexity figures of the SWP GF(2^m) arithmetic circuits in terms of P and Q. The size of the arithmetic circuits without modification are comparable to that outlined in [9, 10], which is expected since they use the same algorithm. Due to the configuration circuitry, the overall area and time complexity incurs a penalty. For reasonable choices of P and Q, this added complexity represents a small percentage overall. For example, if P = 21 and Q = 8 giving M = 168, the percentage overhead due to the configuration circuitry will be 4.01% for the multiplier and 5.81% for the divider circuit. The added propagation delay due to the configuration circuits for multiplication and division corresponds to T_{MUX2} and to $T_{MUX2} + T_{MUXQ}$ respectively. (Note: T_{MUXi} is the propagation delay through an i-input multiplexor.)

Table 3 : Complexity Figures of SISD/SIMD ALU

	Gates for P, Q-bits Logic Unit without Control		Gates for Config. Cct.		Overall Propagation Delay
Multiplier	2PQ 2PQ 2PQ PQ	XOR AND F/F MUX_2	2P 2P	MUX_2 AND	$T_{MUX2} + 2\,T_{XOR} + T_{AND}$
Divider	P(4Q + 1) P(3Q + 1) P(5Q + 2) P(Q + 1) P(16Q + 7)	XOR AND F/F NOT MUX_2	5P P 4P	MUX_2 MUX_Q AND	$T_{XOR} + T_{AND} + 2\,T_{MUX2} + T_{MUXQ}$

2.2. SWP GF(2^m) Processor Architecture

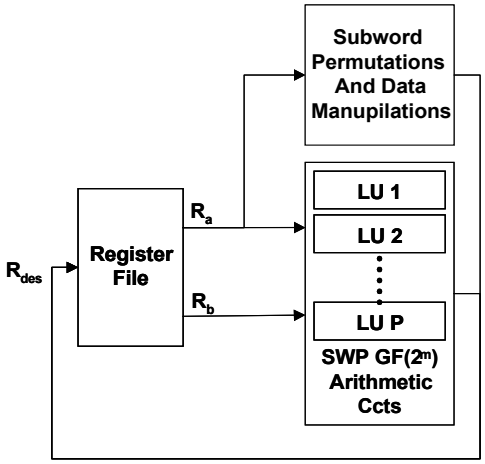

Figure 3 : Processor Datapath Block Diagram

A simplified processor datapath block diagram is shown in Figure 3. It consists of the SWP arithmetic circuits (multiplication, division and addition over GF(2^m) and subword permutation circuits which are necessary to handle subword manipulations. The register file is made up of many Register Locations each with a word size of M bits wide and each Register Location (a word) can be seen as P number of Coefficient Locations (subwords), each Q bits wide (i.e. m=PxQ). In SIMD mode, the data in each Register location can be viewed as a polynomial of degree P-1, with coefficients as elements of a Galois Field up to a size of

$GF(2^Q)$ (i.e. the name coefficient locations). A polynomial with degree larger than P can be stored in two or more Register Locations. In SISD mode, each Register Location will only have one data element in it. See Figure 4.

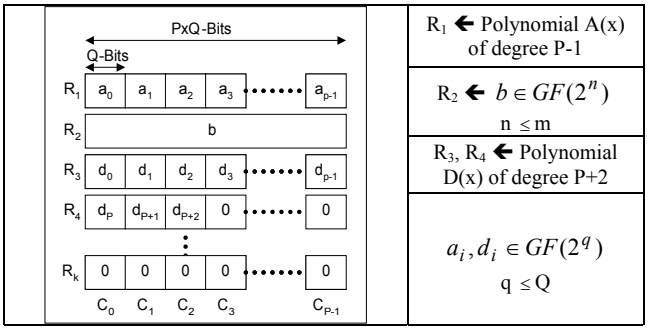

Figure 4 : Register File Structure with different contents

2.3. Core Instruction Set Architecture

For each different sets of applications, the instruction set architecture maybe different. This is because for different applications, additional specialized instructions maybe included to improve significantly the performance of the processor running these applications. However, it is possible to identify a core set of instructions that will be applicable for a majority of applications, and these are usually present regardless of the applications the processor is designed for.

2.3.1. Core Instructions

We denote R_j (C_i) as the i^{th} Coefficient Location of the j^{th} Register Location. In SIMD Mode, MULT, DIVI and ADDP instructions operate on p parallel data pairs in a pair of Register Locations. In SISD Mode, they will operate on only one data pair per pair of Register Locations. The other instructions are required for subword re-arrangement, data alignment and movement (shifting, copying, Subword copying etc).

Table 4: Core Arithmetic Instructions

$ADDP_{SIMD}$	$R_{des}(C_i) \leftarrow [R_a(C_i) + R_b(C_i)] \bmod G(y)$	for i = 0 to P-1
$ADDP_{SISD}$	$R_{des} \leftarrow [R_a + R_b] \bmod G(y)$	
$SUMA_{SIMD}$	$R_{des}(C_a) \leftarrow \sum_{i=0}^{P-1} R_a(C_i)$	
$MULT_{SIMD}$	$R_{des}(C_i) \leftarrow [R_a(C_i) \times R_b(C_i)] \bmod G(y)$	for i = 0 to P-1
$MULT_{SISD}$	$R_{des} \leftarrow R_a \times R_b \bmod G(y)$	
$DIVI_{SIMD}$	$R_{des}(C_i) \leftarrow [R_a(C_i) / R_b(C_i)] \bmod G(y)$	for i = 0 to P-1
$DIVI_{SISD}$	$R_{des} \leftarrow [R_a / R_b] \bmod G(y)$	
$REPA_{SIMD}$	$[R_{des}(C_i)] \leftarrow R_a(C_a)$	for i = 0 to P-1
$REPO_{SIMD}$	$R_{des}(C_{des}) \leftarrow R_a(C_a)$	
SHPX	$R_{des} \leftarrow$ LSB/MSB Bit Shift R_a by x-bits. Pad with Zeros	
COPY	$R_{des} \leftarrow R_a$	
SETC	Setup Instruction for SIMD/SISD mode, Irreducible Polynomial etc	

3. PRIMITIVE OPERATIONS

In this section, we show that almost all of the primitive operations for different applications over $GF(2^m)$ can be synthesized from the core ISA.

3.1. Reed Solomon and BCH codes

All of the primitive operations required in RS and BCH codec can be broken down into polynomial operations over $GF(2^m)$ of one form or another. They are: Polynomial Multiplications (PM), Polynomial Divisions (PD) and Polynomial Evaluations (PE). Non-Systematic RS encoding is basically a PM whereas systematic RS Encoding is a PD. Syndrome Computation and Chien Search of the decoding stage are PEs. Key Equation Solving using Extended Euclidean Algorithm is a combination of PDs and PMs. Therefore, deriving efficient ways of computing these polynomial operations are crucial. This section will briefly describe some ways these polynomial operations can be synthesized simply by the core ISA described before and are by no means definitive. It will be straightforward to extend the techniques presented here to more general polynomial operations.

3.1.1. Polynomial Multiplication

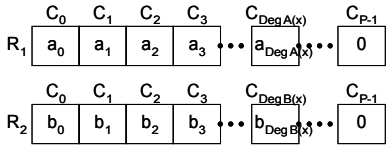

Figure 5 : Data Orientation of A(x) and B(x) for PM

Let $A(x) = \sum_{i=0}^{\deg(A(x))} a_i x^i$, $B(x) = \sum_{i=0}^{\deg(B(x))} b_i x^i$, $a_i, b_i \in GF(2^m)$.
The polynomial multiplication of $D(x) = A(x) \times B(x)$ is given by:

$$D(x) = \sum_{i=0}^{\deg(A(x))} \sum_{j=0}^{\deg(B(x))} a_i b_j x^{i+j}$$

Assuming the degrees of A(x) and B(x) is less than P-1, then each of the polynomial can be fitted into a single Register Location LSB justified as shown in Figure 5. Using the core Arithmetic Instructions, a PM can thus be synthesised:

- $R_1 \leftarrow A(x)$; $R_2 \leftarrow B(x)$; $R_{Temp1} \leftarrow$ Zero Polynomial
- For i = 0 to deg (A(x)) Loop
- R_{Temp2} ← REPA $R_1(C_i)$
- R_{Temp3} ← MULT R_{Temp2}, R_2
- R_{Temp1} ← ADDP R_{Temp3}, R_{Temp1}
- $R_{Result}(C_i)$ ← REPO $R_{Temp1}(C_i)$
- R_{Temp1} ← SHPX LSB, R_{Temp1}, Q-Bits
- End Loop;
- For j = 0 to deg(B(x)) – 1 Loop
- $R_{Result}(C_{deg(A(x))+j})$ ← REPO $R_{Temp1}(C_j)$
- End Loop;

This is basically a multiply-add-shift operation of A(x) and B(x). The result D(x) will be stored in R_{Result}. Note that if deg(A(x) + deg(B(x)) > P-1, the result D(x) maybe exceed the size of one Register Location and have to be stored accordingly. Since the degrees of A(x), B(x) and D(x) of all PMs present in RS and BCH codes can be predetermined, it is relatively easy determine the storage required.

3.1.2. Polynomial Division

Again, let A(x) and B(x) follows the same notation as before. Given A(x) and B(x), calculate find D(x) and E(x) which satisfies the expression:

$$D(x) \times B(x) + E(x) = A(x)$$

In other words, we want to calculate A(x)/B(x) and get the quotient D(x) and the remainder which is E(x). Assuming the deg(A(x)) = deg(B(x)) + 1, then deg(E(x)) = 1 and deg(D(x)) = deg(B(x)) – 1. For simplicity, here we assume that each A(x) and B(x) can be fitted into a single Register Location MSB justified. The steps involved in the polynomial division (also commonly known as Polynomial Long Division) are:

1. R_1 ← A(x); R_2 ← B(x);
2. R_{Temp1} ← REPA $R_1(C_{P-1})$
3. R_{Temp2} ← MULT R_{Temp1}, R_2
4. R_{Temp1} ← REPA $R_2(C_{P-1})$
5. R_{Temp2} ← DIVI R_{Temp2}, R_{Temp1}
6. $R_{Quotient}(C_1)$ ← REPO $R_{Temp2}(C_{P-1})$
7. R_{Temp3} ← ADDP R_{Temp2}, R_1
8. R_{Temp1} ← REPA $R_{Temp3}(C_{P-2})$
9. R_{Temp2} ← MULT R_{Temp1}, R_2
10. R_{Temp1} ← REPA $R_2(C_{P-1})$
11. R_{Temp2} ← DIVI R_{Temp2}, R_{Temp1}
12. $R_{Quotient}(C_0)$ ← REPO $R_{Temp2}(C_{P-2})$
13. $R_{Remainder}$ ← ADDP R_{Temp3}, R_{Temp2}
14. $R_{Quotient}$ = D(x), $R_{Remainder}$ = E(x)

The number of instructions needed to compute a polynomial division will be minimum when A(x) and B(x) can be fitted into a single register. Polynomial division is the fundamental primitive operation in Key Equation Solving using the Euclidean Algorithm and it can be concluded that as long as P ≥ 2t, where t is the error correcting capability of a BCH or RS code, the number of instructions needed for the Euclidean Algorithm will be at a minimum.

3.1.3. Polynomial Evaluation

	C_0	C_1	C_2	C_3	.....	C_{P-1}
R_1	a_0	a_1	a_2	a_3	.....	a_{p-1}
R_2	a_P	a_{P+1}	a_{P+2}	0	.....	0
R_3	1	α	α^2	α^3	.....	α^{p-1}
R_4	α^p	α^{p+1}	α^{p+2}	0	.....	0

Figure 6 :Data Orientation for Polynomial Evaluation

Compute $A(\alpha) = \sum_{i=0}^{\deg(A(x))} a_i \alpha^i$ where $\alpha \in GF(2^m)$. As an example, we assume that A(x) in this case spans two Register Locations as shown in Figure 6. To save on computation cycles, P-1 multiple powers of α are pre-computed and stored in multiple Register Locations as well. To evaluate $A(\alpha)$;

1. R_1, R_2← A(x); R_3, R_4 ← Pre-computed Powers of α ;
2. R_{Temp1} ← MULT R_1, R_3
3. R_{Temp2} ← MULT R_2, R_4
4. R_{Temp1} ← ADDP R_{Temp1}, R_{Temp2}
5. $R_{Result}(C_0)$ ← SUMA R_{Temp1}
6. $A(\alpha)$ = $R_{Result}(C_0)$;

3.2. Advanced Encryption Standard

In general, the basic operations of the AES can be synthesized using the core ISA (with the exception of the Affine Transform required in the SubWord Transformation). We can easily modify the DIVI instruction so that an Forward Affine Transform (FAT) or Inverse Affine Transform (IAT) is computed after and before an inversion is computed respectively (See Table 5). There is substantial parallelism in the AES algorithm, which makes its implementation in a SWP architecture very attractive. An AES State is stored in the register file in the way as shown in Figure 7, where a 128-bit Block (or a state) is stored across 4 Registers Locations. If P > 4, multiple blocks of data can be stored in the same 4 Register Locations. In the example of Figure 7, P = 8, hence we can store 2 AES states every 4 register locations. This also means that multiple AES encryptions or decryptions can be computed at the same time. On closer examination, it is evident that the ShiftRow Operation of the AES is the bottleneck of the system if it is implemented with just the core ISA. A new instruction SHPW is created specifically for the ShiftRow Operation in AES.

Table 5 : Extended Instructions for AES

DIVI (SIMD-AES)	$R_{des}(C_i)$ ← FAT[1/$R_b(C_i)$] mod G(y) $R_{des}(C_i)$ ← 1/ IAT[$R_b(C_i)$] mod G(y)	for i = 0 to P-1
SHRW	R_{des} ← Shift-Row-Index, R_a	

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
R_1	$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$	$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
R_2	$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
R_3	$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
R_4	$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$	$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$
	1st AES State				2nd AES State			

Figure 7 : Storage of AES state in Register File

3.3. Elliptic Curve Cryptography

The primitive operations of the Elliptic Curve Cryptography can be broken down simply into Point Additions and Point Doubling. A Non-Supersingular Elliptic Curve [2] is defined over Affine Co-ordinates as:

$$y^2 + xy = x^3 + a_2x^2 + a_6 \qquad \text{where } a_2 \text{ and } a_6 \in GF(2^m)$$

Given that two points A = (x_1,y_1) and B = (x_2,y_2) lie on an Elliptic Curve, point addition D = A + B is defined as below, where D = (x_3,y_3). If A ≠ B then

$$\theta = \frac{y_2 - y_1}{x_2 - x_1},\ x_3 = \theta^2 + \theta + x_2 + x_1 + a_2,\ y_3 = \theta(x_3 + x_1) - y_1$$

If A=B then $\theta = x_1 + \frac{y_1}{x_1}$ or $\theta = x_2 + \frac{y_2}{x_2}$

and, $x_3 = \theta^2 + \theta + a_2$, $y_3 = x^2 + (\theta + 1)x_3$.

These operate over quadratic equations defined over GF(2^m). and can be synthesized using the core ISA, primarily ADDP, MULT and DIVI operating in SISD mode

4. RESULTS & CONCLUSIONS

It is evident that once the requirements of the primitive operations are determined and a processor architecture is designed to meet these requirements, a stable platform is available that can be tailored for different applications or different groups of applications by changing P and Q at design time. We show an example where P and Q are fixed and determine the range of applications this processor can be used for without re-design. This is to show the inherent flexibility of the processor architecture for different applications and does not represent a practical design flow, where applications usually determine the values of P and Q.

4.1. P = 8, Q = 8 for RS, BCH, AES and ECC

The largest small field size the processor can compute in parallel is constrained by Q. For a BCH or RS (N,K) codes, this will constrain the maximum value of N. Table 6 shows the ranges of (N,K) codecs that can be computed without re-design for Q = 8. For the AES, the only variable is the different key schedule computation and P/4 denotes the number of parallel AES blocks that can be computed at the same time for a specific value of P. The largest field size the processor can operate on in this case is $GF(2^{64})$ (m = PxQ. In practical applications, P can be chosen to be large enough for secure elliptic curve cryptography. A proof of concept prototype GF Processor has been implemented on Field Programmable Gate Arrays (Xilinx Virtex XCV-800) using the concepts outlined in this paper for the parameters of P=8 and Q=8 and Table 7 shows the throughput of a few applications from Table 6. The processor was designed using VHDL on Xilinx Foundation 3.1e and occupies 2505 Slices with an equivalent gate count of 43,251 gates. Through analysis, it can be determined that the size of the processor scales more or less linearly with m (i..e. PxQ).

Table 6: Ranges of Applications for P= 8 and Q=8

Application	Variables for Q = 8	
RS, BCH codec	(255,K) (127,K) (63,K)	(31,K) (15,K) (7,K)
AES	128, 192, 256 Bits Key size. P/4 Parallel Blocks Computations	
ECC	Up to $GF(2^{64})$	

Table 7 : Throughput of $GF(2^{64})$ Processor for P = 8, Q = 8

Results for m = 64 P = 8, Q = 8	Speed 40MHz Clock
RS(255,247) Decoder	11 Mbps
RS(31,25) Decoder	6.6Mbps
BCH(31,16) Decoder*	1.33Mbps
128-Bit Key AES (10 Rounds) without key Expansion. 2 Parallel Computations	3.8 Mbps
128-but Key Schedule Computation	42.5 µs
Elliptic Curve Point Addition Affine Coordinates $GF(2^{61})$	7.125 µs
Elliptic Curve Point Doubling Affine Coordinates $GF(2^{61})$	8.425 µs
*Uses the same program as RS(31,25) Codec. i.e. not optimised for BCH Codec.	

4.2. Conclusions

Flexibility of a given architecture cannot be measured easily in quantifiable terms. Yet, as systems get increasingly complex, issues of design re-use makes a flexible architecture with programmable parameters over a large range of applications increasingly important. There has been a general trend to migrate complexity of traditionally application specific implementations towards software controlled architectures, where flexibility of software automatically allows the maximum flexibility over different applications. However, not all such migrations can be achieved easily, with problems ranging from designing flexible arithmetic circuits to supporting flexible software controlled architectures. This paper has outlined a design space exploration for domain applications based on Galois Fields. The applications are broken down into their primitive operations and a flexible software programmable processor has been designed to handle these primitive operations. This in turn allows the same architecture to be used for all applications that can be defined using these primitive operations. This paper focuses mainly on designing for maximum flexibility across different applications over the same domain, and further work is needed to address design issues at each abstraction level for other constraints like speed, area and power trade-offs with flexibility for a given application in the GF domain. A design methodology can then be defined for primitive based hardware/software co-design techniques.

5. Reference

[1] B. Wicker Stephen, *Error control systems for digital communication and storage*. Englewood Cliffs, N.J. ; London: Prentice Hall : Prentice-HallInternational, 1995.

[2] M. Rosing, *Implementing Elliptic Curve Cryptography*. Greenwich, CT.: Manning, 1999.

[3] J. Daemen and V. Rijmen, "AES Proposal : The Rijndael Block Cipher," *http://csrc.nist.gov/encryption/aes/rijndael/*, 2000.

[4] L. Song, K. K. Parhi, I. Kuroda, and T. Nishitani, "Hardware/software codesign of finite field datapath for low-energy Reed-Solomon codecs," *IEEE-Transactions-on-Very-Large-Scale-Integration-VLSI-Systems. April 2000; 8(2): 160-72*, 2000.

[5] H. M. Ji, "An optimized processor for fast Reed-Solomon encoding and decoding," *2002-IEEE-International-Conference-on-Acoustics,-Speech,-and-Signal-Processing.-Proceedings-Cat.-No.02CH37334. 2002: III-3097-100 vol.3*, 2002.

[6] W. Drescher, M. Mennenga, and G. Fettweis, " An architectural study of a digital signal processor for block codes," *Proceedings-of-the-1998-IEEE-International-Conference-on-Acoustics,-Speech-and-Signal-Processing,-ICASSP-'98-Cat.-No.98CH36181. 1998: 3129-32 vol.5*, 1998.

[7] M. Hasan and A. Wassal, "VLSI Algorithms, Architectures, and Implementation of a Versatile $GF(2^m)$ Processor," *IEEE Transactions On Computers*, vol. 49, pp. 1064-1073, 2000.

[8] R. B. Lee, "Subword parallelism with MAX-2," in *IEEE-Micro. Aug. 1996; 16(4): 51-9*, 1996.

[9] P. A. Scott, S. E. Tavares, and L. E. Peppard, "A Fast VLSI Multiplier for $GF(2^m)$," *IEEE Journal Selected Areas in Communications*, vol. SAC-4, pp. 62-66, 1986.

[10] H. Brunner, A. Curiger, and M. Hofstetter, "On Computing Multiplicative Inverses in GF (2^m)," *IEEE Transactions On Computers C*, vol. 42, pp. 1010, 1993.