Appl. Math. and Comp. Sci., 1997, Vol.7, No.3, 677–688

COMPUTING IN $GF(2^m)$ USING GAP

CZESŁAW KOŚCIELNY*

The GAP system supports, in principle, finite fields of size at most 2^{16}. Therefore, the author discusses in the paper how to compute in larger Galois fields of characteristic 2, using the GAP interpreter. The proposed method of computing is based on the software implementation of operations in $GF(2^m)$ according to a technique typical of parallel finite field arithmetic logic circuits. The GAP functions for determining in $GF(2^m)$ the sum and the product of two arbitrary elements, the k-th power, the square and the multiplicative inverse of an element, even for m equal to several hundreds, are shown. A comprehensive example, concerning the structure investigation of $GF(2^{63})$, $GF(2^{100})$ and $GF(2^{250})$, consisting in determining small subfields of these fields, is also presented.

1. Introduction

Computing in Galois fields of characteristic 2 has been recently widely used in many domains of modern technology. Thus, application research concerning $GF(2^m)$ looks for software instruments which could help to explore the structure of large and huge fields of characteristic 2. It appears that the system GAP can successfuly be used as such a tool.

As is known, the GAP interpreter was born about 9 years ago at RWTH Aachen in Germany, and it becomes an effective software package which gives access to algorithms and data structures of many algebraic systems. The aim of this paper is to demonstrate how to use GAP for computing in large Galois fields of characteristic 2, although GAP in principle supports finite fields of size at most 2^{16} (Schönert *et al.*, 1995). This limitation results from the fact that GAP generates the whole multiplicative group of $GF(q)$ and stores it to perform quickly and in a simple manner operations on elements of $GF(q)$, while the author applies the software approach to computing in $GF(q)$ of characteristic 2, based on the simulation of a technique typical of commonly used parallel finite field arithmetic circuits.

* Department of Robotics and Software Engineering, Technical University of Zielona Góra, ul. Podgórna 50, 65–246 Zielona Góra, Poland, e-mail: ckos@irio.pz.zgora.pl.

2. An Approach to Computing in $GF(2^m)$ Using GAP

In this section, a technique of computing in $GF(2^m)$, usually implemented by hardware, will be recalled.

Let m denote an arbitrary integer less than or equal to 2, and

$$p(x) = x^m + p_{m-1} \cdot x^{m-1} + \cdots + p_1 \cdot x + 1 \tag{1}$$

be an irreducible polynomial of degree m over $GF(2)$, belonging to the exponent e, with β standing for one of its roots. Then the set of elements

$$\beta^i = [a_{i,0} \;\; a_{i,1} \;\; \cdots \;\; a_{i,m-1}] \tag{2}$$

where

$$x^i \equiv \sum_{k=1}^{m} a_{i,m-k} \cdot x^{m-k} \mod(p(x)), \quad a_{i,j} \in GF(2) \tag{3}$$

and

$$i = 0,\ 1,\ \ldots,\ e-1,\ \ e \mid (2^m - 1)$$

forms a subgroup of order e of the multiplicative group of $GF(2^m)$. Moreover, it follows that

$$\left[\beta^0 \;\; \beta^1 \;\; \cdots \;\; \beta^{m-1} \right]^T \tag{4}$$

is the identity matrix and it constitutes the standard basis of $GF(2^m)$ over $GF(2)$. The elements of the vector space generated by the basis (4), which are polynomials in β over $GF(2)$ of degree at most $m-1$, form a finite field $GF(2^m)$ of size 2^m. In such a field addition and multiplication correspond respectively to addition of polynomials over $GF(2)$ and their multiplication modulo the irreducible polynomial (1) over $GF(2)$. Each element of $GF(2^m)$ is represented as a vector whose components are equal to the coefficients of the related polynomial.

Let

$$a = [a_0 \;\; a_1 \;\; \cdots \;\; a_{m-1}] \tag{5}$$

$$b = [b_0 \;\; b_1 \;\; \cdots \;\; b_{m-1}] \tag{6}$$

be two arbitrary elements of $GF(2^m)$, associated with polynomials

$$a(x) = a_0 + a_1 \cdot x + \cdots + a_{m-1} \cdot x^{m-1} \tag{7}$$

$$b(x) = b_0 + b_1 \cdot x + \cdots + b_{m-1} \cdot x^{m-1} \tag{8}$$

Then the sum of elements (5) and (6) will be

$$s = a + b = [(a_0 + b_0) \;\; (a_1 + b_1) \;\; \cdots \;\; (a_{m-1} + b_{m-1})] \tag{9}$$

where addition of components is taken modulo 2. This operation can be implemented by means of a very simple software.

Although the implementation of multiplicative operations is more difficult, the following two theorems will concisely explain it.

Theorem 1. *The multiplication of two elements (5) and (6) of $GF(2^m)$ modulo the irreducible polynomial (1) is described by means of the equation*

$$\begin{bmatrix} pr_0 \\ pr_1 \\ \vdots \\ pr_{m-1} \end{bmatrix} = MM \cdot \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{2m-2} \end{bmatrix} \tag{10}$$

where

$$MM = \begin{bmatrix} a_{0,0} & a_{1,0} & \cdots & a_{2m-2,0} \\ a_{0,1} & a_{1,1} & \cdots & a_{2m-2,1} \\ \cdots & \cdots & \cdots & \cdots \\ a_{0,m-1} & a_{1,m-1} & \cdots & a_{2m-2,m-1} \end{bmatrix} \tag{11}$$

$a_{i,j}$ as in (2), and

$$p_k = \sum_{i+j=k} a_i \cdot b_j, \quad k = 0,\ 1, \ldots,\ 2m-2 \tag{12}$$

a_i, b_j being the components of (5) and (6), respectively. It is obvious that addition in (12) is performed modulo 2 and that the vector

$$[p_0\ p_1\ \cdots\ p_{2m-2}]$$

is associated with the product over $GF(2)$ of the polynomials (7) and (8).

Proof. It suffices to observe that column vectors of the $m \times (2m-1)$ matrix (11), are succesive powers of the element β

$$\beta^i, \quad i = 0, 1, \ldots, 2m-2$$

in transposed form. Then eqn. (10) describes exactly the rules of multiplying over $GF(2)$ two arbitrary polynomials of degree $\leq m-1$, associated with vectors (5) and (6), modulo the irreducible polynomial (1). Therefore the vector

$$[pr_0\ pr_1\ \cdots\ pr_{m-1}]$$

is equal to the product of elements (5) and (6) from $GF(2^m)$. ∎

Theorem 2. *The square of the vector a in $GF(2^m)$ is described by means of the equation*

$$\begin{bmatrix} sq_0 \\ sq_1 \\ \vdots \\ sq_{m-1} \end{bmatrix} = SM \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{m-1} \end{bmatrix} \tag{13}$$

where

$$SM = \begin{bmatrix} a_{0,0} & a_{2,0} & \cdots & a_{2m-2,0} \\ a_{0,1} & a_{2,1} & \cdots & a_{2m-2,1} \\ \cdots & \cdots & \cdots & \cdots \\ a_{0,m-1} & a_{2,m-1} & \cdots & a_{2m-2,m-1} \end{bmatrix} \tag{14}$$

Proof. The $m \times m$ matrix (14) on the right-hand side of (13) represents a linear transformation of $GF(2^m)$ over $GF(2)$. Since the columns of this matrix are equal to

$$(\beta^{2i})^T, \quad i = 0, 1, \ldots, m-1$$

the transformation determines squaring in $GF(2^m)$. Thus the vector

$$[sq_0 \;\; sq_1 \;\; \cdots \;\; sq_{m-1}]$$

is equal to the square of vector (5) in $GF(2^m)$. ■

It should be noted that the k-th power of an arbitrary element a from $GF(2^m)$ can easily be obtained by using the well-known repeated square-and-multiply algorithm and that the multiplicative inverse of a is equal to a^{2^m-2}.

To adapt, as simply as possible, the presented method of computing in $GF(2^m)$ to the GAP system, it is assumed that the polynomial $p(x)$, the elements of $GF(2^m)$ and the matrices MM and SM are represented by the GAP vectors and matrices over the field of rationals, respectively. Next, assuming that m and $p(x)$ are known, one must write down, using the GAP language, functions which return matrices MM and SM, the product of (7) and (8) over $GF(2)$ and the right-hand sides of eqns. (10) and (13). Thereby one can construct functions returning k-th powers and multiplicative inverses.

3. GAP Functions for Computing in $GF(2^m)$

The implementation of functions determining matrices MM and SM is crucial, because these matrices must remain in the memory all the time while computing in $GF(2^m)$. The simplest way of construction of a function which returns the matrix MM is to simulate the action of a linear m-stage shift register with feedback, calculating the remainder resulting from dividing x^i by the polynomial $p(x)$ (Blahut, 1968, p.136). Knowing the matrix MM, one immediately obtains the matrix SM. Taking the above into account, the author has implemented in the GAP language the functions named `MMat` and `SMat`, which return the equivalents of matrices MM and SM, respectively, in the following way:

```
MMat := function ( )
    local  mm, tp, tm, i, k;
    tp := Copy( one );
    mm := [ ];
    mm[1] := Copy( tp );
    k := 1;
    repeat
        k := k + 1;
        i := m + 1;
        tm := tp[m];
        repeat
            i := i - 1;
            tp[i] := (tp[i - 1] + IP[i] * tm) mod 2;
        until i = 2;
        tp[1] := IP[1] * tm;
        mm[k] := Copy( tp );
    until k = 2 * m - 1;
    return mm;
end;

SMat := function ( )
    local  sm, k;
    sm := [ ];
    for k  in [ 1 .. m ]  do
        sm[k] := Copy( MM[2 * k - 1] );
    od;
    return sm;
end;
```

The function `MMat` which uses global variables `m` and `IP` denoting the extension of $GF(2)$ and the vector containing coefficients of the irreducible polynomial $p(x)$, correspondingly, returns the matrix `mm` which symbolizes the transpose of matrix (11). This function also uses a global variable `one` which is equal to the unity under multiplication in $GF(2^m)$. Similarly, the function `SMat` returns the matrix `sm` as a transposed equivalent of matrix (14). In order to call the function `SMat`, one evidently must first initialize the global variable `MM` by means of the assignment `MM := MMat( );`.

Functions GFSum, GFPrd, GFSqr, GFPwr and GFInv, which can be used to determine in $GF(2^m)$ the sum of two elements, the square of an element, the product of two elements, the k-th power of an element and the inverse of an element, respectively, are listed below:

```
GFSum := function ( a, b )
    local  i, sv;
    sv := [ ];
    for i  in [ 1 .. m ]  do
        sv[i] := (a[i] + b[i]) mod 2;
    od;
    return sv;
end;

GFPrd := function ( a, b )
    local  i, k, ab, pr, pm;
    ab := function ( a, b )
          local  i, j, ij, k, w;
          w := [ ];
          for k  in [ 1 .. 2 * m ]  do
              w[k] := 0;
          od;
          for i  in [ 1 .. m ]  do
              for j  in [ 1 .. m ]  do
                  ij := i + j - 1;
                  w[ij] := (w[ij] + a[i] * b[j]) mod 2;
              od;
          od;
          return w;
      end;
    pr := Copy( zero );
    pm := ab( a, b );
    for i  in [ 1 .. m ]  do
        for k  in [ 1 .. 2 * m - 1 ]  do
            pr[i] := (pr[i] + pm[k] * MM[k][i]) mod 2;
        od;
    od;
    return pr;
end;

GFSqr := function ( a )
    local  i, k, sq;
    sq := Copy( zero );
    for i  in [ 1 .. m ]  do
        for k  in [ 1 .. m ]  do
            sq[i] := (sq[i] + a[k] * SM[k][i]) mod 2;
        od;
    od;
    return sq;
end;
```

```
GFPwr := function ( a, k )
    local  aa, pw;
    if k < 0  then
        k := 2 ^ m - 1 + k;
    fi;
    aa := a;
    pw := one;
    if k - Int( k / 2 ) * 2 <> 0  then
        pw := a;
    else
        pw := one;
    fi;
    k := Int( k / 2 );
    while k > 0  do
        aa := GFSqr( aa );
        if k - Int( k / 2 ) * 2 <> 0  then
            pw := GFPrd( aa, pw );
        fi;
        k := Int( k / 2 );
    od;
    return pw;
end;

GFInv := function ( a )
    local  mm, i;
    mm := GFPwr( a, 2 ^ m - 2 );
    return mm;
end;
```

The functions GFSqr and GFPwr use the global variable zero denoting the null vector with m components, which is equal to the identity element under addition in $GF(2^m)$ while the function GFPwr uses the previously-mentioned global variable one which is the m-component vector $[1\,0\,0\,\cdots\,0]$. It can be observed that the function GFPrd uses the function ab returning the product of two polynomials (7) and (8) over $GF(2)$. It is implemented in a conventional manner, but assuming that the coefficients of the polynomial product are reduced modulo 2. It is also obvious that, in order to perform multiplication, the global variable MM is found in the body of the function GFPrd. Since the function GFSqr uses the global variable SM, the instruction `SM := SMat( );` must be executed first if one wants to determine squares in $GF(2^m)$. The function GFPwr utilizes both GFSqr and GFPwr, and therefore the global variables MM and SM must be initialized while powering in $GF(2^m)$.

To find inverses in $GF(2^m)$, the function GFInv is defined. It uses powering since the function GFPwr is very fast. It might be interesting, however, to check if the extended Euclidean remainder algorithm would not be better in this case.

Although the overloading of infix operators is one of the main advantages of the GAP system, all the arithmetic funtions mentioned above are called with parameters, in order to achieve a higher speed of operation. For the same practical reason, instead

of the $GF(2)$-arithmetic provided in GAP, the integer arithmetic modulo 2 is used here.

4. Example

A self-explanatory example of the GAP program, applying functions `MMat`, `SMat`, `GFSqr`, `GFPrd` and `GFPwr` for computing in $GF(2^{63})$, $GF(2^{100})$ and $GF(2^{250})$, constructed using the primitive polynomials $1+x+x^{63}$, $1+x^{37}+x^{100}$ and $1+x^{103}+x^{250}$, respectively, is given in Appendix A. The functions for computing in $GF(2^m)$ are used by the function `compute` having in the header a list of three formal variables `mx`, `kx` and `sx`. The meaning of these variables can be clarified by stating that the function `compute` calculates the multiplicative group of the subfield $GF(2^{sx})$ of the field $GF(2^{mx})$, formed by means of the primitive trinomial $1 + x^{kx} + x^{mx}$ $(sx \mid mx,\ 1 < sx < mx,\quad 0 < kx < mx)$.

The function `compute` uses the following auxiliary functions:

- `info`, printing on the screen what the program will perform;
- `bthc`, converting the m-component binary vector which represents an element of $GF(2^m)$, into the corresponding hexadecimal vector. In this manner, an element of $GF(2^m)$ is represented more compactly by a vector having as components hexadecimal digits according to the mapping:

0	$\leftrightarrow$	0000	4	$\leftrightarrow$	0010	8	$\leftrightarrow$	0001	C	$\leftrightarrow$	0011
1	$\leftrightarrow$	1000	5	$\leftrightarrow$	1010	9	$\leftrightarrow$	1001	D	$\leftrightarrow$	1011
2	$\leftrightarrow$	0100	6	$\leftrightarrow$	0110	A	$\leftrightarrow$	0101	E	$\leftrightarrow$	0111
3	$\leftrightarrow$	1100	7	$\leftrightarrow$	1110	B	$\leftrightarrow$	1101	F	$\leftrightarrow$	1111.

  Since the numbers 63 and 250 are not divisible by 4, the last hexadecimal component of the vector representing the $GF(2^m)$ element for $m = 63$ and $m = 250$ must be treated as a 3-digit and a 2-digit binary number, respectively;
- `printh`, displays a hexadecimal vector on the screen.

The output of the program (slightly retouched) is presented in Appendix B.

5. Conclusion

The author has demonstrated that GAP can support finite fields of characteristic 2 of size much more greater than 2^{16}. It is also not difficult to observe that the presented approach to computing in finite fields can be easily generalized for $GF(p^m)$ with $p > 2$. It proves that meaningful work has really been done in Aachen: the GAP system is a powerful tool for application researchers, easily adaptable to various tasks concerning the solution to many difficult algebraic problems.

Acknowledgement

The author wishes to thank the anonymous referee for very constructive criticism.

Appendices

A. An Example of GAP Program for Computing in $GF(2^{63})$, $GF(2^{100})$ and $GF(2^{250})$

```
m := 0;;
zero := [ ];;
one := [ ];;
IP := [ ];;
SM := [ ];;
MM := [ ];;

Read("GF(2^m).fun"); #GF(2^m).fun is the name of the file, containing
                     #functions for computing in GF(2^m), used in the
                     #body of function compute

compute := function ( mx, kx, sx )
    local  i, x, b0, info, bthc, printh, alpha, beta, order;
    m := mx;
    zero := List( [ 1 .. m ], function ( x )
            return 0;
        end );
    one := Copy( zero );
    one[1] := 1;
    order := (2 ^ m - 1) / (2 ^ sx - 1);
    IP := [  ];
    for i  in [ 1 .. m + 1 ]  do
        IP[i] := 0;
    od;
    IP[1] := 1;
    IP[kx + 1] := 1;
    IP[m + 1] := 1;
    info := function ( x )
          local  i;
          Print( "computing in GF(2^", m, ") constructed using " );
          Print( " primitive polynomial " );
          Print( "\n", "p(x) = 1+" );
          for i  in [ 2 .. m ]  do
              if IP[i] <> 0  then
                  Print( "x^", i - 1, "+" );
              fi;
          od;
          Print( "x^", m, " over GF(2).", "\n", "beta = alpha^", order );
          Print( ", alpha - primitive element ", "\n", "of GF(2^", m, ")." );
```

```
        Print( "\n", "Multiplicative group of GF(", 2 ^ x, ")", "\n" );
        Print( "generated by the element beta" );
        Print( " belonging to GF(2^", m, "):", "\n" );
    end;
bthc := function ( n, l )
        local  i, k, nhd, s, h;
        k := -1;
        nhd := Int( l / 4 );
        h := [  ];
        if nhd <> 0  then
            repeat
                s := 0;
                k := k + 1;
                for i  in [ 1 .. 4 ]  do
                    s := s + n[(4 * k + i)] * 2 ^ (i - 1);
                od;
                h[k + 1] := s;
            until k = nhd - 1;
        fi;
        if l - 4 * nhd <> 0  then
            s := 0;
            for i  in [ 4 * nhd + 1 .. l ]  do
                s := s + n[i] * 2 ^ (i - 4 * nhd - 1);
            od;
            h[nhd + 1] := s;
        fi;
        return h;
    end;
printh := function ( h )
        local  i, nc;
        nc := [  ];
        Print( "[" );
        nc[1] := "A";
        nc[2] := "B";
        nc[3] := "C";
        nc[4] := "D";
        nc[5] := "E";
        nc[6] := "F";
        for i  in [ 1 .. Length( h ) ]  do
            if h[i] in [ 0 .. 9 ]  then
                Print( h[i] );
            fi;
            if h[i] in [ 10 .. 15 ]  then
                Print( nc[h[i] - 9] );
            fi;
        od;
        Print( "]", "\n" );
    end;
info( sx );
MM := MMat(  );
```

```
    SM := SMat(  );
    alpha := Copy( zero );
    alpha[2] := 1;
    b0 := Copy( one );
    Print( "beta^0 = " );
    printh( bthc( b0, m ) );
    beta := GFPwr( alpha, order );
    for i  in [ 1 .. 2 ^ sx - 2 ]  do
        Print( "beta^", i, " = " );
        x := GFPrd( b0, beta );
        b0 := x;
        printh( bthc( x, m ) );
    od;
end;

compute( 63, 1, 3 );
compute( 100, 37, 4 );
compute( 250, 103, 2 );
```

B. Results Obtained After Running the **GAP** Program Given in Appendix A

```
Computing in GF(2^63) constructed using  primitive polynomial
p(x) = 1+x+x^63 over GF(2).

beta = alpha^1317624576693539401, alpha - primitive element of GF(2^63).

Multiplicative group of GF(8), generated by the element beta belonging
to GF(2^63):

beta^0 = [1000000000000000]
beta^1 = [31B4B361C0216000]
beta^2 = [5E00D561D3504110]
beta^3 = [4E00D561D3504110]
beta^4 = [7FB4660013712110]
beta^5 = [21B4B361C0216000]
beta^6 = [6FB4660013712110]

#Run time for GF(2^63): 12529 msec

Computing in GF(2^100) constructed using  primitive polynomial
p(x) = 1+x^37+x^100 over GF(2).

beta = alpha^845100400152152934331135470 25, alpha - primitive
element of GF(2^100).

Multiplicative group of GF(16), generated by the element beta belonging
to GF(2^100):
```

```
beta^0  = [10000000000000000000000000]
beta^1  = [5F25DF2220BECADF000602441]
beta^2  = [4D1435631FDC6AFE040044D60]
beta^3  = [2572EC71568DAFCF004641154]
beta^4  = [3572EC71568DAFCF004641154]
beta^5  = [6A573353763365100040043515]
beta^6  = [2743063069EF0FEE044007875]
beta^7  = [0231EA413F62A021040646921]
beta^8  = [3743063069EF0FEE044007875]
beta^9  = [5D1435631FDC6AFE040044D60]
beta^10 = [7A573353763365100040043515]
beta^11 = [7866D9124951C531044605C34]
beta^12 = [4F25DF2220BECADF000602441]
beta^13 = [1231EA413F62A021040646921]
beta^14 = [6866D9124951C531044605C34]

#Run time for GF(2^100): 41749 msec.

Computing in GF(2^250) constructed using  primitive polynomial
p(x) = 1+x^103+x^250 over GF(2).

beta = alpha^e, e =

603083798111021851164432213586916186735781170133544604372174916707880883541,

alpha - primitive element of GF(2^250).

Multiplicative group of GF(4), generated by the element beta belonging
to GF(2^250):

beta^0 = [10000000000000000000000000000000000000000000000000000000000000]
beta^1 = [B93ACFFA537AA4113CC325A5D9B24ACF1B379FE9B6DADD32B8D6F7AE184B811]
beta^2 = [A93ACFFA537AA4113CC325A5D9B24ACF1B379FE9B6DADD32B8D6F7AE184B811]

#Run time for GF(2^250): 650369 msec.
#Running on PC 486 DX with 100 MHz clock.
```

References

Blahut R.E. (1968): *Theory and Practice of Error Control Codes.* — Reading Mass: Addison-Wesley.

Schönert M. *et al.* (1995): GAP - *Groups, Algorithms and Programming, Version 3, Release 4, Patchlevel 3 (manual).* — RWTH Aachen, Germany.

Received: 14 January 1997
Revised: 14 May 1997