Lecture 7: Finite Fields (PART 4)

PART 4: Finite Fields of the Form $GF(2^n)$

Theoretical Underpinnings of Modern Cryptography

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 9, 2021
4:58pm

Goals:

- To review finite fields of the form $GF(2^n)$
- To show how arithmetic operations can be carried out by directly operating on the bit patterns for the elements of $GF(2^n)$
- **Perl and Python implementations for arithmetic in a Galois Field using my BitVector modules**

CONTENTS

Back to TOC

7.1 CONSIDER AGAIN THE POLYNOMIALS OVER $GF(2)$

- Recall from Lecture 6 that $GF(2)$ is a finite field consisting of the set $\{0, 1\}$, with modulo 2 addition as the group operator and modulo 2 multiplication as the ring operator. In Section 6.7 of Lecture 6, we also talked about polynomials over $GF(2)$. Along the lines of the examples shown there, here are some more:

  $x + 1$
  $x^2 + x + 1$
  $x^2 + 1$
  $x^3 + 1$
  x
  1
  x^5
  x^{10000}
  ...
  ...

  The examples shown only use 0 and 1 for the coefficients in the polynomials. Obviously, we could also have shown polynomials with negative coefficients. However, as you'd recall from Lecture 6, -1 is the same as +1 in $GF(2)$. [Does $23 * x^5 + 1$ belong to the set of polynomials defined over $GF(2)$? How about $-3 * x^7 + 1$? The answer to both questions is yes. Can you justify the answer?]

- Obviously, the number of such polynomials is infinite.

- The polynomials can be subject to the algebraic operations of addition and multiplication in which the coefficients are added and multiplied according to the rules that apply to $GF(2)$.

- As stated in the previous lecture, the set of such polynomials forms a **ring**, called the **polynomial ring**.

Back to TOC

7.2 MODULAR POLYNOMIAL ARITHMETIC

Let's now add one more twist to the algebraic operations we carry out on all the polynomials over $GF(2)$:

- In Section 6.11 of Lecture 6, I defined an **irreducible polynomial** as a polynomial that cannot be factorized into lower-degree polynomials. From the set of **all** polynomials that can be defined over $GF(2)$, let's now consider the following *irreducible polynomial*:

  $$x^3 + x + 1$$

  By the way there exist **only two** irreducible polynomials of degree 3 over $GF(2)$. The other is $x^3 + x^2 + 1$.

- For the set of **all** polynomials over $GF(2)$, let's now consider polynomial arithmetic modulo the irreducible polynomial $x^3 + x + 1$.

- To explain what I mean by polynomial arithmetic modulo the irreducible polynomial, when an algebraic operation — *we are*

obviously talking about polynomial multiplication — results in a polynomial whose degree **equals or exceeds** that of the irreducible polynomial, we will take for our result the **remainder modulo the irreducible polynomial**.

- For example,

$$
\begin{aligned}
&(x^2 + x + 1) \times (x^2 + 1) \; mod \; (x^3 + x + 1) \\
&= \quad (x^4 + x^3 + x^2) + (x^2 + x + 1) \; mod \; (x^3 + x + 1) \\
&= \quad (x^4 + x^3 + x + 1) \; mod \; (x^3 + x + 1) \\
&= \quad -x^2 - x \\
&= \quad x^2 + x
\end{aligned}
$$

Recall that $1 + 1 = 0$ in $GF(2)$. That's what caused the x^2 term to disappear in the second expression on the right hand side of the equality sign.

- For the division by the modulus in the above example, we used the result

$$
\frac{(x^4 + x^3 + x + 1)}{(x^3 + x + 1)} \quad = \quad x + 1 + \frac{-x^2 - x}{x^3 + x + 1}
$$

Obviously, for the division on the left hand side, our first quotient term is x. Multiplying the divisor by x yields $x^4 + x^2 + x$ that when subtracted from the dividend gives us

$x^3 - x^2 + 1$. This dictates that the next term of the quotient be 1, and so on.

Back to TOC

7.3 HOW LARGE IS THE SET OF POLYNOMIALS WHEN MULTIPLICATIONS ARE CARRIED OUT MODULO $x^3 + x + 1$

- With multiplications modulo $x^3 + x + 1$, we have only the following **eight** polynomials in the set of polynomials over $GF(2)$:

  0
  1
  x
  x^2
  $x + 1$
  $x^2 + 1$
  $x^2 + x$
  $x^2 + x + 1$

- We will refer to this set as $GF(2^3)$ where the exponent of 2, which in this case is 3, is the degree of the **modulus polynomial**.

- Our conceptualization of $GF(2^3)$ is analogous to our conceptualization of the set Z_8. The **eight** elements of Z_8 are

to be thought of as integers modulo 8. So, basically, Z_8 maps **all** integers to the eight numbers in the set Z_8. Similarly, $GF(2^3)$ maps all of the polynomials over $GF(2)$ to the eight polynomials shown above.

- But note the crucial difference between $GF(2^3)$ and Z_8: $GF(2^3)$ is a field, whereas Z_8 is NOT.

Back to TOC

7.4 HOW DO WE KNOW THAT $GF(2^3)$ IS A FINITE FIELD?

- We do know that $GF(2^3)$ is an abelian group because of the operation of polynomial addition satisfies all of the requirements on a group operator and because polynomial addition is commutative. [Every polynomial in $GF(2^3)$ is its own additive inverse because of how the two numbers in $GF(2)$ behave with respect to modulo 2 addition.]

- $GF(2^3)$ is also a commutative ring because polynomial multiplication distributes over polynomial addition (and because polynomial multiplication meets all the other stipulations on the ring operator: closedness, associativity, commutativity).

- $GF(2^3)$ is an integral domain because of the fact that the set contains the multiplicative identity element 1 and because if for $a \in GF(2^3)$ and $b \in GF(2^3)$ we have

$$a \times b \;=\; 0 \;\; mod \;\; (x^3 \;+\; x \;+\; 1)$$

then either $a \;=\; 0$ or $b \;=\; 0$. This can be proved easily as follows:

  – Assume that **neither** a **nor** b is zero when

$a \times b \;=\; 0 \; mod \; (x^3 \;+\; x \;+\; 1)$. In that case, the following equality must also hold

$$a \times b \;=\; (x^3 \;+\; x \;+\; 1)$$

since

$$0 \;\equiv\; (x^3 \;+\; x \;+\; 1) \;\; mod \;\; (x^3 \;+\; x \;+\; 1)$$

– But the above implies that the **irreducible** polynomial $x^3 \;+\; x \;+\; 1$ can be factorized, which by definition cannot be done.

- We now argue that $GF(2^3)$ is a finite field because it is a finite set and because it contains a unique multiplicative inverse for every non-zero element.

- $GF(2^3)$ contains a unique multiplicative inverse for every non-zero element for the same reason that Z_7 contains a unique multiplicative inverse for every non-zero integer in the set. (For a counterexample, recall that Z_8 does not possess multiplicative inverses for 2, 4, and 6.) Stated formally, we say that for every non-zero element $a \in GF(2^3)$ there is always a unique element $b \in GF(2^3)$ such that $a \times b \;=\; 1$.

- The above conclusion follows from the fact if you multiply a non-zero element a with each of the eight elements of $GF(2^3)$,

the result will the **eight distinct** elements of $GF(2^3)$. Obviously, the results of such multiplications **must** equal 1 for exactly one of the non-zero element of $GF(2^3)$. So if $a \times b = 1$, then b must be the multiplicative inverse for a.

- The same thing happens in Z_7. If you multiply a non-zero element a of this set with each of the seven elements of Z_7, you will get **seven distinct** answers. The answer **must** therefore equal 1 for at least one such multiplication. When the answer is 1, you have your multiplicative inverse for a.

- For a counterexample, this is not what happens in Z_8. When you multiply 2 with every element of Z_8, you do not get **eight distinct** answers. (Multiplying 2 with every element of Z_8 yields $\{0, 2, 4, 6, 0, 2, 4, 6\}$ that has only **four distinct** elements).

- For a more formal proof (by contradiction) of the fact that if you multiply a non-zero element a of $GF(2^3)$ with every element of the same set, no two answers will be the same, let's assume that this assertion is false. That is, we assume the existence of two distinct b and c in the set such that

$$a \times b \equiv a \times c \; mod \; (x^3 + x + 1)$$

That implies

$$a \times (b - c) \equiv 0 \; mod \; (x^3 + x + 1)$$

That implies that either a is 0 or that b equals c. In either case, we have a contradiction.

- **The upshot is that $GF(2^3)$ is a finite field.**

Back to TOC

7.5 $GF(2^n)$ IS A FINITE FIELD FOR EVERY n

- None of the arguments on the previous three pages is limited by the value 3 for the power of 2. That means that $GF(2^n)$ is a finite field for every n.

- To find all the polynomials in $GF(2^n)$, we obviously need an irreducible polynomial of degree n.

- AES arithmetic, presented in the next lecture, is based on $GF(2^8)$. It uses the following irreducible polynomial

$$x^8 + x^4 + x^3 + x + 1$$

- The finite field $GF(2^8)$ used by AES obviously contains 256 distinct polynomials over $GF(2)$.

- In general, $GF(p^n)$ is a finite field for any **prime** p. The elements of $GF(p^n)$ are polynomials over $GF(p)$ (which is the same as the set of residues Z_p).

Back to TOC

7.6 REPRESENTING THE INDIVIDUAL POLYNOMIALS IN $GF(2^n)$ BY BINARY CODE WORDS

- Let's revisit the **eight polynomials** in $GF(2^3)$ (when the modulus polynomial is $x^3 + x + 1$:

$$
\begin{aligned}
&0\\
&1\\
&x\\
&x + 1\\
&x^2\\
&x^2 + 1\\
&x^2 + x\\
&x^2 + x + 1
\end{aligned}
$$

- We now claim that there is nothing sacred about the variable x. in such polynomials.

- We can think of x^i as being merely a place-holder for a bit.

- That is, we can think of the polynomials as bit strings corresponding to the coefficients that can only be 0 or 1, **each**

power of x representing a specific position in a bit string.

- So the 2^3 polynomials of $GF(2^3)$ can therefore be represented by the bit strings:

$$\begin{aligned} 0 &\Rightarrow 000 \\ 1 &\Rightarrow 001 \\ x &\Rightarrow 010 \\ x^2 &\Rightarrow 100 \\ x + 1 &\Rightarrow 011 \\ x^2 + 1 &\Rightarrow 101 \\ x^2 + x &\Rightarrow 110 \\ x^2 + x + 1 &\Rightarrow 111 \end{aligned}$$

- If we wish, we can give a decimal representation to each of the above bit patterns. The decimal values between 0 and 7, both limits inclusive, would have to obey the addition and multiplication rules corresponding to the underlying finite field.

- Given any n at all, exactly the same approach can be used to come up with 2^n bit patterns, each pattern consisting of n bits, for a set of integers that would constitute a finite field, provided

we have available to us an irreducible polynomial of degree n.

Back to TOC

7.7 SOME OBSERVATIONS ON BIT-PATTERN ADDITIONS IN $GF(2^n)$

- We know that the polynomial coefficients in $GF(2^n)$ must obey the arithmetic rules that apply to $GF(2)$ (which is the same as Z_2, the set of remainders modulo 2). These rules were reviewed in Section 6.6 of Lecture 6.

- As stated in Section 6.6 of Lecture 6, we know that the operation of addition in $GF(2)$ is like the logical XOR operation.

- Therefore, adding the bit patterns in $GF(2^n)$ simply amounts to taking the **bitwise XOR of the bit patterns**. For example, the following must hold in $GF(2^8)$:

$$
\begin{array}{rcrcrcrcrcl}
5 & + & 13 & = & 0000\,0101 & + & 0000\,1101 & = & 0000\,1000 & = & 8 \\
76 & + & 22 & = & 0100\,1100 & + & 0001\,0110 & = & 0101\,1010 & = & 90 \\
7 & - & 3 & = & 0000\,0111 & - & 0000\,0011 & = & 0000\,0100 & = & 4 \\
7 & + & 3 & = & 0000\,0111 & + & 0000\,0011 & = & 0000\,0100 & = & 4
\end{array}
$$

- The last two examples above illustrate that **subtracting is the same as adding** in $GF(2^8)$. That is because each "number" is its own additive inverse in $GF(2^8)$. In other words, for every $x \in GF(2^8)$, we have $-x \;=\; x$. Yet another way of saying the same thing is that for every $x \in GF(2^8)$, we have $x \;+\; x \;=\; 0$.

Back to TOC

7.8 SOME OBSERVATIONS ON ARITHMETIC MULTIPLICATION IN $GF(2^n)$

- As you just saw, it is obviously convenient to use simple binary arithmetic (in the form of XOR operations) for additions in $GF(2^n)$. Could we do the same for multiplications?

- We can of course multiply the bit patterns of $GF(2^n)$ by going back to the modulo polynomial arithmetic and using the multiplications operations defined in $GF(2)$ for the coefficients. [Recall from Section 6.6 of Lecture 6 that multiplication is the same as "logical AND" in $GF(2)$.]

- But it would be **nice** if we could directly multiply the bit patterns of $GF(2^n)$ without having to think about the underlying polynomials.

- It turns out that we can indeed do so, but the technique is specific to the order of the finite field being used. The **order of a finite field** refers to the number of elements in the field. So the order of $GF(2^n)$ is 2^n.

- More particularly, the bitwise operations needed for directly multiplying two bit patterns in $GF(2^n)$ are specific to the irreducible polynomial that defines a given $GF(2^n)$.

- On the next slide, we will focus specifically on the $GF(2^8)$ finite field that is used in AES, which we will take up in the next lecture, and show that multiplications can be carried out directly in this field by using bitwise operations.

Back to TOC

7.9 DIRECT BITWISE OPERATIONS FOR MULTIPLICATIONS IN $GF(2^8)$

- Let's consider the finite field $GF(2^8)$ that is used in AES. As you will see in the next lecture, this field is derived using the following irreducible polynomial of degree 8:

$$m(x) \;=\; x^8 \;+\; x^4 \;+\; x^3 \;+\; x \;+\; 1$$

- Now let's see how we can carry out multiplications with direct bitwise operations in this $GF(2^8)$.

- We first take note of the following equality in $GF(2^8)$:

$$x^8 \;\; mod \;\; m(x) \;=\; x^4 \;+\; x^3 \;+\; x \;+\; 1$$

  The result follows immediately by a long division of x^8 by $x^8 + x^4 + x^3 + x + 1$. Obviously, the first term of the quotient will be 1. Multiplying the divisor by the quotient yields $x^8 + x^4 + x^3 + x + 1$. When this is subtracted from the dividend x^8, we get $-x^4 - x^3 - x - 1$, which is the same as the result shown above.

- Now let's consider the general problem of multiplying a general polynomial $f(x)$ in $GF(2^8)$ by just x. Let's represent $f(x)$ by

  $$f(x) \;=\; b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

  Therefore, this $f(x)$ stands for the bit pattern $b_7b_6b_5b_4b_3b_2b_1b_0$.

- Obviously,

  $$f(x)\times x \;=\; b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x$$

  But now recall that we must take the modulo of this polynomial with respect to $m(x) = x^8 + x^4 + x^3 + x + 1$. What that yields depends on whether or not the bit b_7 is set.

- If the bit b_7 of $f(x)$ is equals 0, then the right hand above is already in the set of polynomials in $GF(2^8)$ and nothing further needs to be done. In this case, the output bit pattern is $b_6b_5b_4b_3b_2b_1b_00$.

- However, if b_7 equals 1, we need to divide the polynomial we have for $f(x) \times x$ by the modulus polynomial $m(x)$ and keep just the remainder. Therefore, when $b_7 = 1$, we can write

$$(f(x) \times x) \;\; mod \;\; m(x)$$

$$\begin{aligned}
&= (b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) \; mod \; m(x) \\
&= (b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) \quad + \quad (x^8 \; mod \; m(x)) \\
&= (b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) + (x^4 + x^3 + x \; + 1) \\
&= (b_6b_5b_4b_3b_2b_1b_00) \quad \otimes \quad (00011011)
\end{aligned}$$

where, in the last expression shown, we have used the fact that the addition in $GF(2^8)$ corresponds to the logical XOR operation for the bit patterns involved.

Back to TOC

7.10 SUMMARY OF HOW A MULTIPLICATION IS CARRIED OUT IN $GF(2^8)$

- Let's say you want to multiply two bit patterns B_1 and B_2, each 8 bits long.

- If B_2 is the bit pattern 00000001, then obviously nothing needs to be done. The result is B_1 itself.

- If B_2 is the bit pattern 00000010, then we are multiplying B_1 by x. Now the answer depends on the value of the most significant bit in B_1. If B_1's MSB is 0, the result is obtained by shifting the B_1 bit pattern to the left by one bit and inserting a 0 bit from the right. On the other hand, if B_1's MSB is 1, first we again shift the B_1 bit pattern to the left as above. Next, we take the XOR of the shifted pattern with the bit pattern 00011011 for the final answer.

- If B_2 is the bit pattern 00000100, then we are multiplying B_1 by x^2. This amounts to first multiplying B_1 by x, and then multiplying the result again by x. So it amounts to two

applications of the logic in the previous two steps.

- In general, if B_2 consists of a single bit in the j^{th} position from the right (using the 0 index for the right-most position), we need j applications of the logic laid out above for multiplying with x.

- Even more generally, when B_2 consists of an arbitrary bit pattern, we consider the bit pattern to be a sum of bit patterns each containing only single bit.

- For example, if B_2 is 10000011, we can write

$$
\begin{aligned}
& B_1 \times 10000011 \\
&= \quad B_1 \times (00000001 \;+\; 00000010 \;+\; 10000000) \\
&= \quad (B_1 \times 00000001) \;+\; (B_1 \times 00000010) \;+\; (B_1 \times 10000000) \\
&= \quad (B_1 \times 00000001) \;\otimes\; (B_1 \times 00000010) \;\otimes\; (B_1 \times 10000000)
\end{aligned}
$$

Each of the three multiplications shown in the final expression involves multiplying B_1 with a single power of x. That we can easily do with the logic already explained.

Back to TOC

7.11 FINDING MULTIPLICATIVE INVERSES IN $GF(2^n)$

- So far we have talked about efficient bitwise operations for implementing the addition, the subtraction, and the multiplication operations for the bit patterns in $GF(2^n)$.

- But what about division? Can division be carried out directly on the bit patterns? You could if you knew the multiplicative inverses of the bit patterns. Dividing a bit pattern B_1 by the bit pattern B_2 would mean multiplying B_1 by the multiplicative inverse of B_2.

- In general, you can use the Extended Euclid's Algorithm (See Section 5.7 of Lecture 5) for finding the multiplicative inverse (MI) of a bit pattern in $GF(2^n)$ provided you carry out all the arithmetic in that algorithm according to the rules appropriate for $GF(2^n)$. Toward that end, shown on the next page is my implementation of the bit array arithmetic in $GF(2^n)$. The function $gf\_MI(num, mod, n)$ returns the multiplicative inverse of a *num* bit pattern in the finite field $GF(2^n)$ when the modulus bit pattern is as specified by *mod*. As the note at the beginning of the code presented says, the OO version of this implementation is included in Versions 2.1 and higher of my

Python `BitVector` class.

```
#!/usr/bin/env python

##   GF_Arithmetic.py
##   Author: Avi Kak
##   Date:   February 13, 2011

##   Note: The code you see in this file has already been incorporated in
##         Version 2.1 and higher of the BitVector module.  If you like
##         the object-oriented approach to scripting, just use that module
##         directly.  The documentation in that module shows how to make
##         the function calls for doing GF(2^n) arithmetic.

from BitVector import *

def gf_divide(num, mod, n):
    '''
    Using the arithmetic of the Galois Field GF(2^n), this function divides
    the bit pattern 'num' by the modulus bit pattern 'mod'
    '''
    if mod.length() > n+1:
        raise ValueError("Modulus bit pattern too long")
    quotient = BitVector( intVal = 0, size = num.length() )
    remainder = num.deep_copy()
    i = 0
    while 1:
        i = i+1
        if (i==num.length()): break
        mod_highest_power = mod.length() - mod.next_set_bit(0) - 1
        if remainder.next_set_bit(0) == -1:
            remainder_highest_power = 0
        else:
            remainder_highest_power = remainder.length() \
                                - remainder.next_set_bit(0) - 1
        if (remainder_highest_power < mod_highest_power) \
              or int(remainder)==0:
            break
        else:
            exponent_shift = remainder_highest_power - mod_highest_power
            quotient[quotient.length() - exponent_shift - 1] = 1
            quotient_mod_product = mod.deep_copy();
            quotient_mod_product.pad_from_left(remainder.length() - \
                                        mod.length() )
            quotient_mod_product.shift_left(exponent_shift)
            remainder = remainder ^ quotient_mod_product
    if remainder.length() > n:
        remainder = remainder[remainder.length()-n:]
    return quotient, remainder

def gf_multiply(a, b):
```

```
    '''
    Using the arithmetic of the Galois Field GF(2^n), this function multiplies
    the bit pattern 'a' by the bit pattern 'b'.
    '''
    a_highest_power = a.length() - a.next_set_bit(0) - 1
    b_highest_power = b.length() - b.next_set_bit(0) - 1
    result = BitVector( size = a.length()+b.length() )
    a.pad_from_left( result.length() - a.length() )
    b.pad_from_left( result.length() - b.length() )
    for i,bit in enumerate(b):
        if bit == 1:
            power = b.length() - i - 1
            a_copy = a.deep_copy()
            a_copy.shift_left( power )
            result ^=  a_copy
    return result

def gf_multiply_modular(a, b, mod, n):
    '''
    Using the arithmetic of the Galois Field GF(2^n), this function returns 'a'
    divided by 'b' modulo the bit pattern in 'mod'.
    '''
    a_copy = a.deep_copy()
    b_copy = b.deep_copy()
    product = gf_multiply(a_copy,b_copy)
    quotient, remainder = gf_divide(product, mod, n)
    return remainder

def gf_MI(num, mod, n):
    '''
    Using the arithmetic of the Galois Field GF(2^n), this function returns the
    multiplicative inverse of the bit pattern 'num' when the modulus polynomial
    is represented by the bit pattern 'mod'.
    '''
    NUM = num.deep_copy(); MOD = mod.deep_copy()
    x = BitVector( size=mod.length() )
    x_old = BitVector( intVal=1, size=mod.length() )
    y = BitVector( intVal=1, size=mod.length() )
    y_old = BitVector( size=mod.length() )
    while int(mod):
        quotient, remainder = gf_divide(num, mod, n)
        num, mod = mod, remainder
        x, x_old = x_old ^ gf_multiply(quotient, x), x
        y, y_old = y_old ^ gf_multiply(quotient, y), y
    if int(num) != 1:
            return "NO MI. However, the GCD of ", str(NUM), " and ", \
                                        str(MOD), " is ", str(num)
    else:
        quotient, remainder = gf_divide(x_old ^ MOD, MOD, n)
        return remainder


mod = BitVector( bitstring = '100011011' )             # AES modulus

a = BitVector( bitstring = '10000000' )
```

```
result = gf_MI( a, mod, 8 )
print("\nMI of %s is: %s" % (str(a), str(result)))

a = BitVector( bitstring = '10010101' )
result = gf_MI( a, mod, 8 )
print("\nMI of %s is: %s" % (str(a), str(result)))

a = BitVector( bitstring = '00000000' )
result = gf_MI( a, mod, 8 )
print("\nMI of %s is: %s" % (str(a), str(result)))
```

- When you run the above script, it returns the following result:

```
MI of 10000000 is: 10000011

MI of 10010101 is: 10001010

MI of 00000000 is: ('NO MI. However, the GCD of ', '00000000', ' and ', '100011011', ' is ', '100011011')
```

- Shown below is a Perl version of the same code. The version uses my open-source module `Algorithm::BitVector` that you can download from the CPAN archive.

```
#!/usr/bin/env perl

##   GF_Arithmetic.pl
##   Author: Avi Kak
##   Date:   February 5, 2016

##   Note: The code you see in this file has already been incorporated in
##         Version 1.24 and above of the Perl Algorithm::BitVector module.
##         If you like object-oriented approach to scripting, just use that
##         module directly.  The documentation in that module shows how to
##         make function calls for doing GF(2^n) arithmetic.

use strict;
use warnings;
use Algorithm::BitVector;

my $mod = Algorithm::BitVector->new( bitstring => '100011011' );              # AES modulus

my $a = Algorithm::BitVector->new( bitstring => '10000000' );
my $result = gf_MI( $a, $mod, 8 );
```

```
print "\n\nMI of  $a is: $result\n";

$a = Algorithm::BitVector->new( bitstring => '10010101' );
$result = gf_MI( $a, $mod, 8 );
print "\nMI of $a is: $result\n";

$a = Algorithm::BitVector->new( bitstring => '00000000' );
$result = gf_MI( $a, $mod, 8 );
print "\nMI of $a is: $result\n";



##  Using the arithmetic of the Galois Field GF(2^n), this function divides
##  the bit pattern $num by the modulus bit pattern $mod
sub gf_divide {
    my ($num, $mod, $n) = @_;
    die "Modulus bit pattern too long" if $mod->length() > $n + 1;
    my $quotient = Algorithm::BitVector->new( intVal => 0, size => $num->length() );
    my $remainder = $num->deep_copy();
    for (my $i = 0; $i < $num->length(); $i++) {
        my $mod_highest_power = $mod->length() - $mod->next_set_bit(0) - 1;
        my $remainder_highest_power;
        if ($remainder->next_set_bit(0) == -1) {
            $remainder_highest_power = 0;
        } else {
            $remainder_highest_power = $remainder->length() - $remainder->next_set_bit(0) - 1;
        }
        if (($remainder_highest_power < $mod_highest_power) or (int($remainder)==0)) {
            last;
        } else {
            my $exponent_shift = $remainder_highest_power - $mod_highest_power;
            $quotient->set_bit($quotient->length() - $exponent_shift - 1, 1);
            my $quotient_mod_product = $mod->deep_copy();
            $quotient_mod_product->pad_from_left($remainder->length() - $mod->length() );
            $quotient_mod_product->shift_left($exponent_shift);
            $remainder ^= $quotient_mod_product;
        }
    }
    $remainder = Algorithm::BitVector->new(bitlist =>
                    $remainder->get_bit([$remainder->length()-$n .. $remainder->length()-1]))
                    if $remainder->length() > $n;
    return ($quotient, $remainder);
}

##  Using the arithmetic of the Galois Field GF(2^n), this function multiplies
##  the bit pattern $arg1 by the bit pattern $arg2
sub gf_multiply {
    my ($arg1,$arg2) = @_;
    my ($a, $b) = ($arg1->deep_copy(), $arg2->deep_copy());
    my $a_highest_power = $a->length() - $a->next_set_bit(0) - 1;
    my $b_highest_power = $b->length() - $b->next_set_bit(0) - 1;
    my $result = Algorithm::BitVector->new( size => $a->length( )+ $b->length() );
    $a->pad_from_left( $result->length() - $a->length() );
    $b->pad_from_left( $result->length() - $b->length() );
    foreach my $i (0 .. $b->length() - 1) {
```

```
        my $bit = $b->get_bit($i);
        if ($bit == 1) {
            my $power = $b->length() - $i - 1;
            my $a_copy = $a->deep_copy();
            $a_copy->shift_left( $power );
            $result ^= $a_copy;
        }
    }
    return $result;
}

##  Using the arithmetic of the Galois Field GF(2^n), this function returns $a
##  divided by $b modulo the bit pattern in $mod
sub gf_multiply_modular {
    my ($a, $b, $mod, $n) = @_;
    my $a_copy = $a->deep_copy();
    my $b_copy = $b->deep_copy();
    my $product = gf_multiply($a_copy,$b_copy);
    my ($quotient, $remainder) = gf_divide($product, $mod, $n);
    return $remainder;
}

##  Using the arithmetic of the Galois Field GF(2^n), this function returns the
##  multiplicative inverse of the bit pattern $num when the modulus polynomial
##  is represented by the bit pattern $mod
sub gf_MI {
    my ($num, $mod, $n) = @_;
    my $NUM = $num->deep_copy(); my $MOD = $mod->deep_copy();
    my $x = Algorithm::BitVector->new( size => $mod->length() );
    my $x_old = Algorithm::BitVector->new( intVal => 1, size => $mod->length() );
    my $y = Algorithm::BitVector->new( intVal => 1, size => $mod->length() );
    my $y_old = Algorithm::BitVector->new( size => $mod->length() );
    my ($quotient, $remainder);
    while (int($mod)) {
        ($quotient, $remainder) = gf_divide($num, $mod, $n);
        ($num, $mod) = ($mod, $remainder);
        ($x, $x_old) = ($x_old ^ gf_multiply($quotient, $x), $x);
        ($y, $y_old) = ($y_old ^ gf_multiply($quotient, $y), $y);
    }
    if (int($num) != 1) {
        return "NO MI. However, the GCD of $NUM and $MOD is: $num\n";
    } else {
        ($quotient, $remainder) = gf_divide($x_old ^ $MOD, $MOD, $n);
        return $remainder;
    }
}
```

- As you'd exect, when you execute the file shown above, you get exactly the same output that you saw earlier for the Python version of the code.

- If you have fixed the value of n for a particular $GF(2^n)$ field (and if n is not too large), you can precompute the multiplicative inverses for all the elements of $GF(2^n)$ and store them away. (Recall that the MI of a bit pattern A in $GF(2^n)$ is a bit pattern B so that $A \times B \ = \ 1$.

- The table below shows the multiplicative inverses for the bit patterns of $GF(2^3)$. Also shown are the additive inverses. But note that every element x is its own additive inverse. Also note that the additive identity element is not expected to possess a multiplicative inverse.

	Additive Inverse	Multiplicative Inverse
000	000	-----
001	001	001
010	010	101
011	011	110
100	100	111
101	101	010
110	110	011
111	111	100

Back to TOC

7.12 USING A GENERATOR TO REPRESENT THE ELEMENTS OF $GF(2^n)$

- It is particularly convenient to represent the elements of a Galois Field $GF(2^n)$ with the help of a **generator element**.

  [As mentioned in Section 5.5 of Lecture 5, GF in the notation $GF(p^n)$ stands for "Galois Field" after the French mathematician Evariste Galois who died in 1832 at the age of 20 in a duel with a military officer who had cast aspersions on a young woman whom Galois cared for. The young woman was the daughter of the physician of the hostel where Galois stayed. Galois was the first to use the word "group" in the sense we have used in these lectures.]

- If g is a **generator element**, then every element of $GF(2^n)$, except for the 0 element, can be expressed as some power of g.

- Consider a finite field of order q. As mentioned previously in Section 7.8, the **order** of a finite field is the number of elements in the field. If g is the generator of this finite field, then the finite field can be expressed by the set

$$\{0,\ g^0,\ g^1,\ g^2,\ \ldots,\ g^{q-2}\}$$

- How does one specify a generator?

- A generator is obtained from the irreducible polynomial that went into the creation of the finite field. If $f(g)$ is the irreducible polynomial used, then g is that element which symbolically satisfies the equation $f(g) = 0$. You do not actually solve this equation for its roots since an irreducible polynomial cannot have actual roots in the underlying number system used, **but only use this equation for the relationship it gives between the different powers of g.**

- Consider the case of $GF(2^3)$ defined with the irreducible polynomial $x^3 + x + 1$. The generator g is that element which symbolically satisfies $g^3 + g + 1 = 0$, implying that such an element will obey

$$g^3 = -g - 1 = g + 1$$

- Now we can show that every power of g will correspond to some element of $GF(2^3)$.

- Shown below are the first several powers of g along with the element 0 at the very top:

$$\begin{aligned} & 0 \\ g^0 &= 1 \\ g^1 &= g \\ g^2 &= g^2 \\ g^3 &= g + 1 \end{aligned}$$

$$
\begin{aligned}
g^4 \;=\; g(g^3) \;=\; g(g+1) \;&=\; g^2 \;+\; g \\
g^5 \;=\; g(g^4) \;=\; g(g^2+g) \;=\; g^3 \;+\; g^2 \;&=\; g^2 \;+\; g \;+1 \\
g^6 \;=\; g(g^5) \;=\; g(g^2+g+1) \;=\; g^3 \;+\; g^2 \;+\; g \;&=\; g^2 \;+\; 1 \\
\\
g^7 \;=\; g(g^6) \;=\; g(g^2+1) \;=\; g^3 \;+\; g \;&=\; 1 \\
&\vdots
\end{aligned}
$$

- Note the powers g^0 through g^6 of the generator element, along with the element 0, correspond to the eight polynomials of $GF(2^3)$ shown on Slide 10.

- The higher powers of g obey the relationship $g^k \;=\; g^{k \bmod 7}$ for the example shown. As shown above, g^7 is the same as g^0.

- Since every polynomial in $GF(2^n)$ is represented by a power of g, multiplying any two polynomials in $GF(2^n)$ becomes trivial — we just have to add the exponents of g modulo $(2^n - 1)$.

- So we have the conclusion that if g is the generator element of a finite field of the form $GF(2^n)$, then all the powers of g from g^0 through g^{2^n-2}, along with the element 0, correspond to the elements of the finite field.

- That is, using the generator notation allows the multiplications of the elements of the finite field to be carried out without

reference to the irreducible polynomial.

Back to TOC

7.13 HOMEWORK PROBLEMS

1. This question is a litmus test of whether you understand the concepts presented in this lecture at a deep level: As mentioned in Section 7.2, there exist two different *irreducible polynomials* of degree 3 over $GF(2)$:

$$x^3 \; + \; x \; + \; 1$$

and

$$x^3 \; + \; x^2 \; + \; 1$$

Obviously, the finite field $GF(2^3)$ can be constructed with either of these two irreducible polynomials. Regardless of which polynomial we use, we end up with the same set of bit patterns: $\{000,\ 001,\ 010,\ 011,\ 100,\ 101,\ 110,\ 111\}$. The MI (multiplicative inverse) of 010 is 101 when you base $GF(2^3)$ on the irreducible polynomial $x^3 + x + 1$. (You can verify this fact by multiplying the polynomials x and $x^2 + 1$ and evaluating the result modulo the polynomial $x^3 + x + 1$.) The question you are being asked is whether the MI of 010 will be different when $GF(2^3)$ is based on $x^3 + x^2 + 1$?

2. When the set of all integers is divided by a prime, we obtain a set of remainders whose elements obey a certain special

property with regard to the modulo multiplication operator over the set. What is that property?

3. As computer engineers, our world of work is steeped in bits and bytes. Yet we seem to be obsessing about polynomials. Pourquoi?

4. When the set of all polynomials over $GF(p)$ for a prime p is divided by an irreducible polynomial, we obtain a set of remainders with some very special properties. What is so special about this set? How is such a set denoted?

5. How do we get a finite field of the form $GF(2^n)$?

6. If $GF(p)$ gives us a finite field (with p elements), why is that not good enough for us? Why do we need finite fields of the form $GF(2^n)$?

7. How will you prove that $GF(2^3)$ is at least an integral domain? How will you prove it is a finite field?

8. Let's say that our irreducible polynomial is $x^3 + x + 1$. Obviously, each polynomial in $GF(2^3)$ will be of degree 2 or less. Drawing a parallel between the polynomials and the bit

patterns, how many polynomials are there in $GF(2^3)$?

9. With polynomial coefficients drawn from $GF(2)$, let's use the irreducible polynomial $x^3 + x + 1$ to construct the finite field $GF(2^3)$. Now calculate

$$\begin{aligned} (x^2 + x + 1) + (x^2 + 1) &= ? \\ (x^2 + x + 1) - (x^2 + 1) &= ? \\ (x^2 + x + 1) \times (x^2 + 1) &= ? \\ (x^2 + x + 1) \;/\; (x^2 + 1) &= ? \end{aligned}$$

10. Given the following two 3-bit binary code words from $GF(2^3)$ with the modulus polynomial $x^3 + x + 1$:

$$\begin{aligned} B_1 &= 1\ 1\ 1 \\ B_2 &= 1\ 0\ 1 \end{aligned}$$

Now calculate:

$$\begin{aligned} B_1 + B_2 &= ? \\ B_1 - B_2 &= ? \\ B_1 \times B_2 &= ? \end{aligned}$$

$$B_1 \ / \ B2 \quad = \quad ?$$

Do you see any similarities between this question and the previous question? What would happen to the results in this question if we changed the modulus polynomial to $x^3 + x^2 + 1$?

11. **Programming Assignment:**

Write a Perl or Python script that can serve as a four function calculator for carrying out the arithmetic operations ($+$, $-$, $\times$, and $\div$) on the polynomials that belong to the finite field $GF(2^8)$ using the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. When started, your script should place the user in an interactive mode and wait for the user to enter expressions for evaluation. Your script should prompt the user for three items: 1) a bitstring that would serve as the first operand; 2) another bitstring that would serve as the second operand; and, finally, 3) the operator to be used. The bits in each input bit pattern supplied by the user would stand for the respective polynomial coefficients. The script should output a bitstring that is the result of the operation.