CMPE-661 Interface & Digital Electronics

Exercise 2

Binary Finite Field Arithmetic

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students. Other than code provided by the instructor for this exercise, all code was developed by me.

Jacob LaPietra

Performed February 3, 2023

Submitted February 24, 2023

Lecture Section 01

Instructor:

Lukowiak

Kurdziel

TA:

Payton Burak

Matthew Krebs

Abstract:

The main purpose of this exercise was to learn how to create functions to compute addition, multiplication, division, and multiplicative inverse for an embedded processor. There were three main parts to the exercise where methods were both created and tested on the Zynq 7020 platform. After the arithmetic functions were designed and tested some optimizations were explored in order to see what could be done to improve the functionality of our code.

Design Methodology:

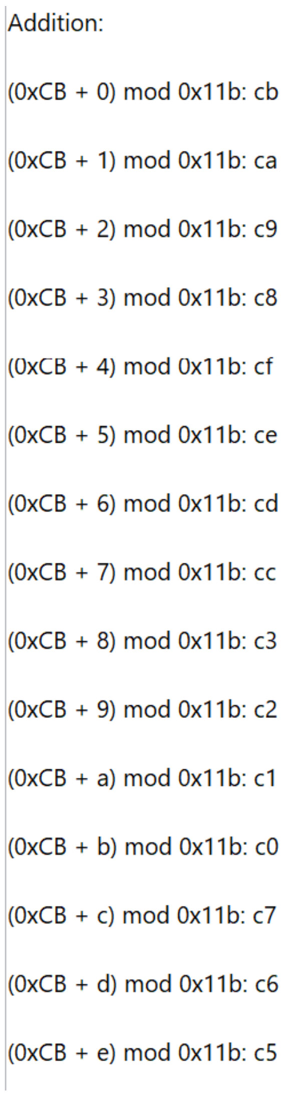
The first part of the exercise was to create methods for addition, multiplication and fining all of the generators in a Binary Finite Field. To compute the addition two thirty two bit integers were XORed together then the modulus was taken from the result and the field polynomial. The mod function was designed to find the remainder of two numbers being divided. To do this a while loop was used to check the difference in degree between the two parameters. Within the while loop the difference of degrees was taken and the second parameter shifted by the difference was added to the modulus result. After the addition the degree of the first parameter was updated to match the degree of the current modulus result. For the multiplication a for loop was used to loop up to the degree of the field polynomial. Within the for loop the current bit of the second parameter was checked to see if it was one. If it was one the first parameter was shifted by the current position of the current bit and added to the total, making sure to mod it if it was larger than the field polynomial. Next a method was written to find all of the generators of a field. To do this a two dimensional for loop was used to loop through all of the row and columns of a power table. Within the second for loop a current power value was updated and the modulus was taken with the field. If the modulus was equal to one the for loop would break. Outside of the second for loop the number of powers it took for the for loop to exit was checked. If that value was equal to the parameter stating the size of the field the row value was added to a list.

The second part of the exercise consisted of creating methods for division and multiplicative inverse. For division a similar method to the modulus was used. Within the while loop for the method the same calculations were used as in the modulus method, however a new variable was used to keep track of the intermediate quotient value. At each step the bit corresponding the difference in the degrees was added to the quotient. After the while loop the quotient value was returned. For the multiplicative inverse method Extended Euclidian Algorithm was used. This algorithm consisted of finding the quotient, remainder, alpha and beta values at each intermediate step. Initial values were loaded into alpha, beta and the remainder. At each step the quotient was found by dividing the remainder from two iterations ago with the previous remainder. The remainder from this calculation was then used for the current remainder value. Alpha and beta values were both calculated by adding the respective value from two iterations ago with the value from the previous iteration multiplied by the current quotient. This process continued until the current remainder was less than two. If the remainder was one the GCD between the two input parameters was one and the alpha and beta values were returned. If the remainder was zero, the GCD was the value of the previous remainder and alpha and beta were set to NULL because no inverse was found. The value of beta was the inverse of the second parameter being modded by the first parameter and the alpha value was the inverse of the first parameter being modded by the second parameter.

The third part of the exercise consisted of testing different compilation options to compare the difference in runtimes between specific optimizations. Four different configurations were used. The first two were enabling and disabling instruction and data cache while leaving compiler optimizations disabled and the last two were enabling and disabling the instruction and data cache while enabling compiler optimizations. The execution time in CPU cycles was measured while taking the multiplicative inverse of all of the numbers in the G(2^8) field.

Results:

After completing the necessary functions for the first part of the exercise the sum was computed for every number from zero to 255 with the value 0xCB to ensure that the function worked properly.

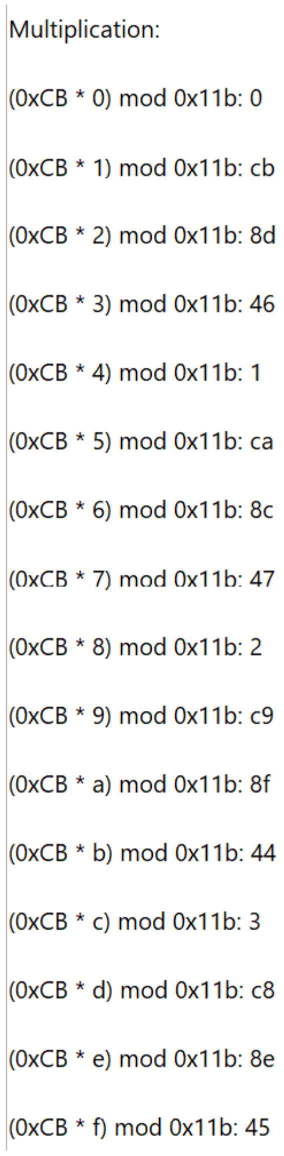


```
Addition:

(0xCB + 0) mod 0x11b: cb

(0xCB + 1) mod 0x11b: ca

(0xCB + 2) mod 0x11b: c9

(0xCB + 3) mod 0x11b: c8

(0xCB + 4) mod 0x11b: cf

(0xCB + 5) mod 0x11b: ce

(0xCB + 6) mod 0x11b: cd

(0xCB + 7) mod 0x11b: cc

(0xCB + 8) mod 0x11b: c3

(0xCB + 9) mod 0x11b: c2

(0xCB + a) mod 0x11b: c1

(0xCB + b) mod 0x11b: c0

(0xCB + c) mod 0x11b: c7

(0xCB + d) mod 0x11b: c6

(0xCB + e) mod 0x11b: c5
```

Figure 1: Addition Results

Figure 1 shows that the results from the additions yielded the proper answers proving that the algorithm was implemented properly. If the addition of two numbers would provide an answer larger than the field polynomial, then the function would automatically take the modulus of the result to keep the answer within bounds.

The next method that was tested was the field multiplication method. Similar to the addition, values from zero to 255 were multiplied with the value 0xCB to ensure that all of the results were correct.

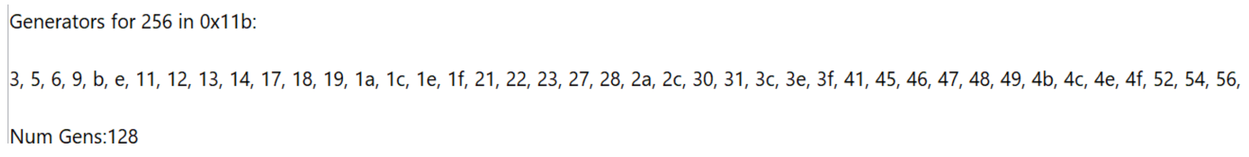


```
Multiplication:

(0xCB * 0) mod 0x11b: 0

(0xCB * 1) mod 0x11b: cb

(0xCB * 2) mod 0x11b: 8d

(0xCB * 3) mod 0x11b: 46

(0xCB * 4) mod 0x11b: 1

(0xCB * 5) mod 0x11b: ca

(0xCB * 6) mod 0x11b: 8c

(0xCB * 7) mod 0x11b: 47

(0xCB * 8) mod 0x11b: 2

(0xCB * 9) mod 0x11b: c9

(0xCB * a) mod 0x11b: 8f

(0xCB * b) mod 0x11b: 44

(0xCB * c) mod 0x11b: 3

(0xCB * d) mod 0x11b: c8

(0xCB * e) mod 0x11b: 8e

(0xCB * f) mod 0x11b: 45
```

Figure 2: Multiplication Results

The results in Figure 2 show that the results from the multiplications shows are correct and if a value were to overflow over the field polynomial the method would automatically reduce the answer to be within bounds. It did this because at each addition in the process the modulus of the answer and the field polynomial was taken so that no calculation overflowed.

The field generators were then found, and the results were printed in a list.

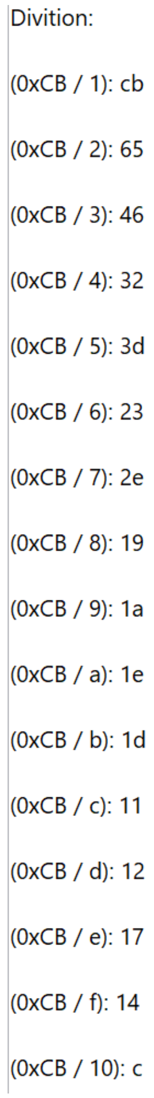


```
Generators for 256 in 0x11b:

3, 5, 6, 9, b, e, 11, 12, 13, 14, 17, 18, 19, 1a, 1c, 1e, 1f, 21, 22, 23, 27, 28, 2a, 2c, 30, 31, 3c, 3e, 3f, 41, 45, 46, 47, 48, 49, 4b, 4c, 4e, 4f, 52, 54, 56,

Num Gens:128
```

Figure 3: Field Generators

Figure 4 shows that the generators for the field G(2^8) were properly calculated and the number of generators cane out to the expected value of 128.

The next method that was tested was the division method. This was tested by dividing the value 0xCB with the numbers zero through 255.

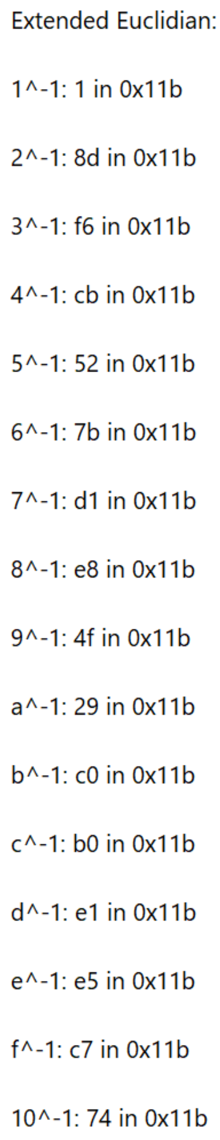


```
Divition:

(0xCB / 1): cb

(0xCB / 2): 65

(0xCB / 3): 46

(0xCB / 4): 32

(0xCB / 5): 3d

(0xCB / 6): 23

(0xCB / 7): 2e

(0xCB / 8): 19

(0xCB / 9): 1a

(0xCB / a): 1e

(0xCB / b): 1d

(0xCB / c): 11

(0xCB / d): 12

(0xCB / e): 17

(0xCB / f): 14

(0xCB / 10): c
```

Figure 4: Division Results

Figure 4 shows that the division method works properly since all of the results of the division computed properly. This method did not check for any overflows since an assumption was made that all of the inputs would fall within the bounds of the field and since the result of the division will always be a smaller number the result would never be able to overflow from the field.

The multiplicative inverses were the next calculations that were verified. All of the values from one through 255 were checked, skipping zero because there would be no multiplicative inverse.

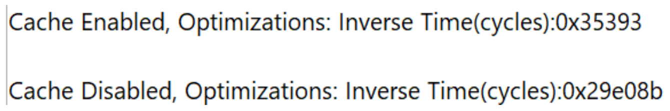


```
Extended Euclidian:

1^-1: 1 in 0x11b

2^-1: 8d in 0x11b

3^-1: f6 in 0x11b

4^-1: cb in 0x11b

5^-1: 52 in 0x11b

6^-1: 7b in 0x11b

7^-1: d1 in 0x11b

8^-1: e8 in 0x11b

9^-1: 4f in 0x11b

a^-1: 29 in 0x11b

b^-1: c0 in 0x11b

c^-1: b0 in 0x11b

d^-1: e1 in 0x11b

e^-1: e5 in 0x11b

f^-1: c7 in 0x11b

10^-1: 74 in 0x11b
```

Figure 5: Multiplicative Inverse Results

The results in Figure 5 show that the multiplicative inverses were calculated properly. If a irreducible polynomial was not used for the field polynomial then some of the values would not have a multiplicative inverse. Because a GCD of 1 is needed to find a inverse if the field polynomial was not relatively prime to all of the values in the field then some of the values in the field would have a GCD other than one with it.

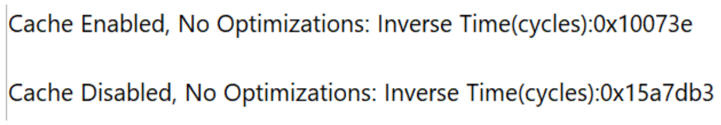
The lasts test that were run were comparing the optimizations that could be enabled for the code. The runtimes were checked by comparing the current CPU cycle numbers between the beginning and end of computing all of the multiplicative inverses in the field and taking the difference.



```
Cache Enabled, Optimizations: Inverse Time(cycles):0x35393

Cache Disabled, Optimizations: Inverse Time(cycles):0x29e08b
```

Figure 6: Runtimes with Compiler Optimizations Enabled

Figure 6 shows that with instruction and data cache enabled the code took an order of magnitude less time than without the caches enabled. This is expected since the caches should reduce the number of stall cycles the CPU needs to perform.



```
Cache Enabled, No Optimizations: Inverse Time(cycles):0x10073e

Cache Disabled, No Optimizations: Inverse Time(cycles):0x15a7db3
```

Figure 7: Runtimes with Compiler Optimizations Disabled

Figure 7 shows similar results to Figure 6 in that enabling the caches reduces the runtime of the code. The difference between the figures however is that Figure 7 shows that without compiler optimizations the runtime of the code is increased. This is expected since the compiler optimizations will reorder and unroll the code so that the CPU does not need to include as many stall cycles.

Conclusion:

The exercise was successful since the creation and use of the arithmetic functions was shown as well and the benefits of optimizations. These tools will be beneficial for future exercises since the arithmetic functions will be used in creating some of the cryptography functions that will be designed later. The optimizations will also be used and will allow the more complex functions that are written for the platform to run faster and more efficiently on the hardware.

Hands-on 2: Binary Finite Field Arithmetic

Student's Name: Jacob LaPietra Section: 1

Demo		Point Value	Points Earned	Date
Part 1: GF Arithmetic and Generators	Addition, Multiplication, Generators	20	20	PB 2/9/2023
Part 2: GF Arithmetic and multiplicative inverses	Long division, Multiplicative Inverse	20	20	PB 2/21/2023
Part 3: Routine Performance	Caches Disabled	10	10	PB 2/21/2023
	Caches Enabled	10	10	
	Caches Disabled No Optimizations	10	10	
	Caches Enabled No Optimizations	10	10	

Part 1 sign-offs are due on week from starting this assignment.

To receive any grading credit students must earn points for both the demonstration and the report.

Report		Point Value	Points Earned	Comments
Discussion		4		
Part 1: GF Arithmetic and Generators	Terminal Output	4		
Part 2: GF Arithmetic and multiplicative inverses	Terminal Output	4		
Part 3: Routine Performance	Terminal Output Caches Disabled	2		
	Terminal Output Caches Enabled	2		
	Terminal Output Caches Disabled No Optimizations	2		
	Terminal Output Caches Enabled No Optimizations	2		
Total for demo and report		100		