Primitives for Finite Field Arithmetic in Network Switches

Daniel Seara[*] **Bernardo Conde**[*] Eduard Marin[†] Muriel Médard[‡] Muhammad Shahbaz[§] Fernando M. V. Ramos[*]

[*]INESC-ID, IST, ULisboa;
[†]Telefonica Research;
[‡]MIT;
[§]Purdue

Finite Field arithmetic: a primer

- “Conventional” arithmetic operations are done over infinite fields
- However, it is very common to want to perform arithmetic operations over fields that contain a **finite** number of elements!
    - These elements can all be encoded in a finite amount of space.
- For example, **cryptographic operations**, **network encoding**, …, are all done over sets of data of known size (e.g., blocks of 128 bits).

In all these common networking cases, we need to perform **finite field arithmetic**

Finite Field arithmetic: a primer

- Field: set of numbers with well defined basic operations: addition, subtraction, multiplication and division
    - For example: field of real numbers ($\mathbb{R}$), field of rational numbers ($\mathbb{Q}$)
- Finite: the set has a **finite** number of elements
    - $\mathbb{R}$ and $\mathbb{Q}$ have an **infinite** number of elements, so they are not finite fields
- Finite Fields also known as Galois Fields (GF)
    - Most common: $\mathbb{N}\ mod\ p^k$ where *p* is prime

$$GF(7) = \{0, 1, 2, 3, 4, 5, 6\}$$

$$GF(2^4) = \{0, 1, \ldots, 15\}$$

All numbers that fit in 4 bits!!!

Operations in Finite Fields

- Operations in these fields output results that are **different** from common arithmetic
- Why? All operations have to output a number **that is part of the field!**

$$\mathbb{R}$$

10 + 21 = 31 ✓

100 + 221 = 321 ✓

$$GF(2^8) = \{0, 1, \ldots, 255\}$$

10 + 21 = 31 ✓

100 + 221 = 321 ✗

Operations in Finite Fields

- Operations in these fields output results that are **different** from common arithmetic
- Why? All operations have to output a number **that is part of the field!**

$$\mathbb{R}$$

10 + 21 = 31 ✓

100 + 221 = 321 ✓

$$GF(2^8) = \{0, 1, \ldots, 255\}$$

10 + 21 = 31 ✓

100 + 221 = 185 ✓

Operations in Finite Fields

- Operations in these fields output results that are **different** from common arithmetic
- Why? All operations have to output a number **that is part of the field!**

$\mathbb{R}$	$GF(2^8) = \{0, 1, \ldots, 255\}$
10 + 21 = 31 ✓	10 + 21 = 31 ✓
100 + 221 = 321 ✓	100 + 221 = 185 ✓
100 - 221 = -121 ✓	100 - 221 = 185 ✓
10 * 221 = 2210 ✓	10 * 221 = 19 ✓
221 / 10 = 22.1 ✓	221 / 10 = 145 ✓

Outline

- Design approaches for network switches
  - Log/Antilog tables
  - Russian Peasant Algorithm
- A way forward
- Conclusion/Q&A

Design approaches for network switches

Addition and Subtraction in Finite Fields

- Additions and Subtractions in Finite Field GF(2^m) are **simple**
  - It is just a **simple bitwise XOR** between the operands

$$\begin{matrix} 100 = 01100100 \\ 221 = 11011101 \end{matrix} \xrightarrow[-]{+} 185 = 10111001$$

Multiplication in Finite Fields

- Multiplication is **hard**
- There are 2 main approaches
  - Memory intensive (using log/antilog tables)
  - Compute intensive (e.g., using the Russian Peasant Algorithm)

Note: division is very similar to multiplication, dividing *a* and *b* is the same as:

$$a/b = a * b^{-1}$$

Where *b*^-1 is the *inverse* of *b.*

Multiplication – Table method

$$a * b = g^{log_g(a)+log_g(b)}$$

- Idea: use **logarithm tables** to turn **multiplications** into **additions**
- Problem: requires storing the logarithms of all field values + all the antilogs

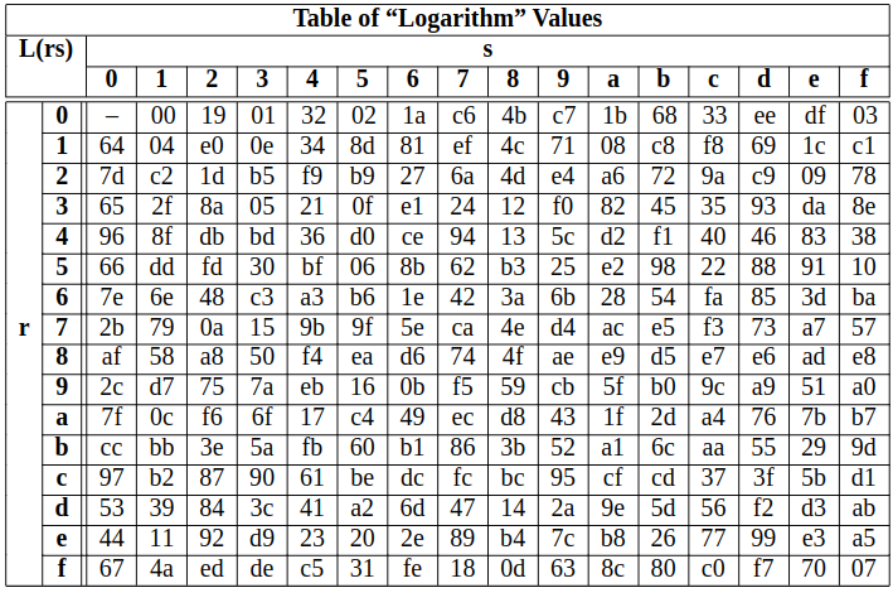


Table of “Logarithm” Values

L(rs)		s															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
r	**0**	–	00	19	01	32	02	1a	c6	4b	c7	1b	68	33	ee	df	03
	1	64	04	e0	0e	34	8d	81	ef	4c	71	08	c8	f8	69	1c	c1
	2	7d	c2	1d	b5	f9	b9	27	6a	4d	e4	a6	72	9a	c9	09	78
	3	65	2f	8a	05	21	0f	e1	24	12	f0	82	45	35	93	da	8e
	4	96	8f	db	bd	36	d0	ce	94	13	5c	d2	f1	40	46	83	38
	5	66	dd	fd	30	bf	06	8b	62	b3	25	e2	98	22	88	91	10
	6	7e	6e	48	c3	a3	b6	1e	42	3a	6b	28	54	fa	85	3d	ba
	7	2b	79	0a	15	9b	9f	5e	ca	4e	d4	ac	e5	f3	73	a7	57
	8	af	58	a8	50	f4	ea	d6	74	4f	ae	e9	d5	e7	e6	ad	e8
	9	2c	d7	75	7a	eb	16	0b	f5	59	cb	5f	b0	9c	a9	51	a0
	a	7f	0c	f6	6f	17	c4	49	ec	d8	43	1f	2d	a4	76	7b	b7
	b	cc	bb	3e	5a	fb	60	b1	86	3b	52	a1	6c	aa	55	29	9d
	c	97	b2	87	90	61	be	dc	fc	bc	95	cf	cd	37	3f	5b	d1
	d	53	39	84	3c	41	a2	6d	47	14	2a	9e	5d	56	f2	d3	ab
	e	44	11	92	d9	23	20	2e	89	b4	7c	b8	26	77	99	e3	a5
	f	67	4a	ed	de	c5	31	fe	18	0d	63	8c	80	c0	f7	70	07

Log table

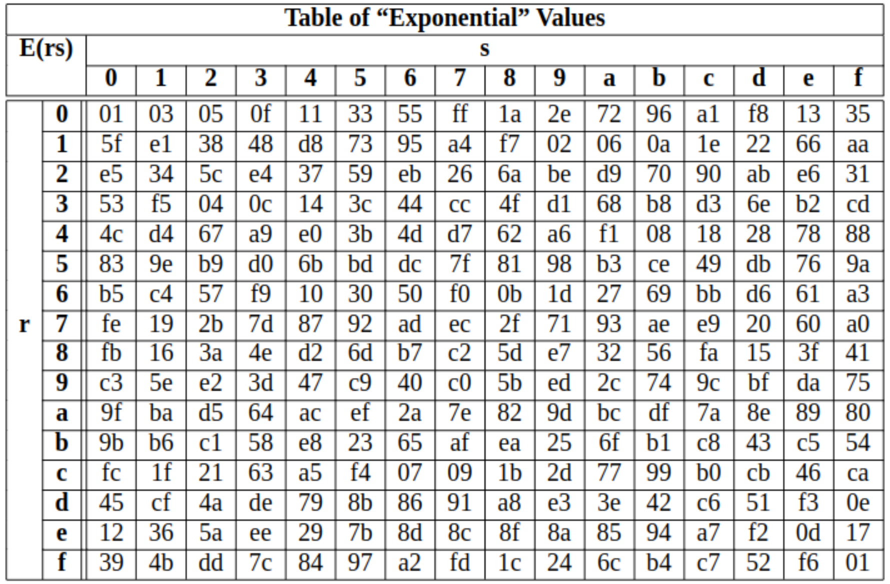


Table of “Exponential” Values

E(rs)		s															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
r	**0**	01	03	05	0f	11	33	55	ff	1a	2e	72	96	a1	f8	13	35
	1	5f	e1	38	48	d8	73	95	a4	f7	02	06	0a	1e	22	66	aa
	2	e5	34	5c	e4	37	59	eb	26	6a	be	d9	70	90	ab	e6	31
	3	53	f5	04	0c	14	3c	44	cc	4f	d1	68	b8	d3	6e	b2	cd
	4	4c	d4	67	a9	e0	3b	4d	d7	62	a6	f1	08	18	28	78	88
	5	83	9e	b9	d0	6b	bd	dc	7f	81	98	b3	ce	49	db	76	9a
	6	b5	c4	57	f9	10	30	50	f0	0b	1d	27	69	bb	d6	61	a3
	7	fe	19	2b	7d	87	92	ad	ec	2f	71	93	ae	e9	20	60	a0
	8	fb	16	3a	4e	d2	6d	b7	c2	5d	e7	32	56	fa	15	3f	41
	9	c3	5e	e2	3d	47	c9	40	c0	5b	ed	2c	74	9c	bf	da	75
	a	9f	ba	d5	64	ac	ef	2a	7e	82	9d	bc	df	7a	8e	89	80
	b	9b	b6	c1	58	e8	23	65	af	ea	25	6f	b1	c8	43	c5	54
	c	fc	1f	21	63	a5	f4	07	09	1b	2d	77	99	b0	cb	46	ca
	d	45	cf	4a	de	79	8b	86	91	a8	e3	3e	42	c6	51	f3	0e
	e	12	36	5a	ee	29	7b	8d	8c	8f	8a	85	94	a7	f2	0d	17
	f	39	4b	dd	7c	84	97	a2	fd	1c	24	6c	b4	c7	52	f6	01

Antilog table

Multiplication – Table method example

- Let’s multiply 10 by 25 using this method
  - 10 = 0x0A; 25 = 0x19
- Step 1: Go to log table and find the values of the logarithms

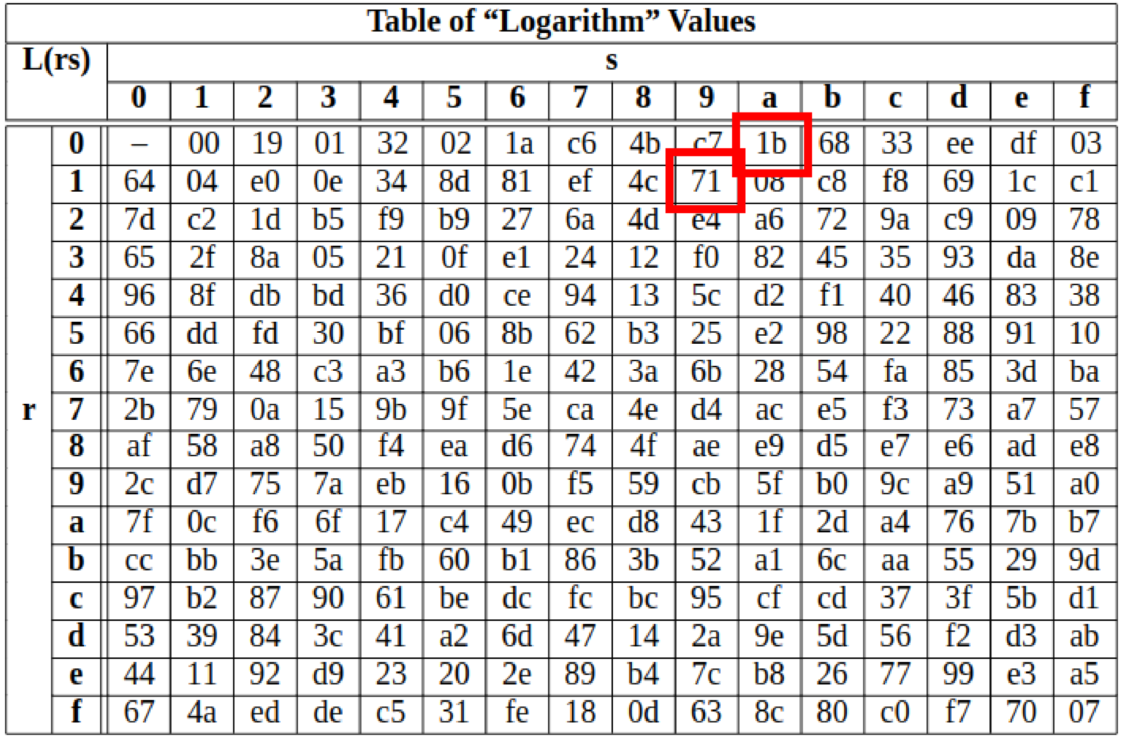


Table of “Logarithm” Values

L(rs)		s															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
	0	–	00	19	01	32	02	1a	c6	4b	c7	1b	68	33	ee	df	03
	1	64	04	e0	0e	34	8d	81	ef	4c	71	08	c8	f8	69	1c	c1
	2	7d	c2	1d	b5	f9	b9	27	6a	4d	e4	a6	72	9a	c9	09	78
	3	65	2f	8a	05	21	0f	e1	24	12	f0	82	45	35	93	da	8e
	4	96	8f	db	bd	36	d0	ce	94	13	5c	d2	f1	40	46	83	38
	5	66	dd	fd	30	bf	06	8b	62	b3	25	e2	98	22	88	91	10
	6	7e	6e	48	c3	a3	b6	1e	42	3a	6b	28	54	fa	85	3d	ba
r	**7**	2b	79	0a	15	9b	9f	5e	ca	4e	d4	ac	e5	f3	73	a7	57
	8	af	58	a8	50	f4	ea	d6	74	4f	ae	e9	d5	e7	e6	ad	e8
	9	2c	d7	75	7a	eb	16	0b	f5	59	cb	5f	b0	9c	a9	51	a0
	a	7f	0c	f6	6f	17	c4	49	ec	d8	43	1f	2d	a4	76	7b	b7
	b	cc	bb	3e	5a	fb	60	b1	86	3b	52	a1	6c	aa	55	29	9d
	c	97	b2	87	90	61	be	dc	fc	bc	95	cf	cd	37	3f	5b	d1
	d	53	39	84	3c	41	a2	6d	47	14	2a	9e	5d	56	f2	d3	ab
	e	44	11	92	d9	23	20	2e	89	b4	7c	b8	26	77	99	e3	a5
	f	67	4a	ed	de	c5	31	fe	18	0d	63	8c	80	c0	f7	70	07

Multiplication – Table method example

- We found 0x1B and 0x71
- Step 2: Add them
  - 0x1B + 0x71 = 0x8C

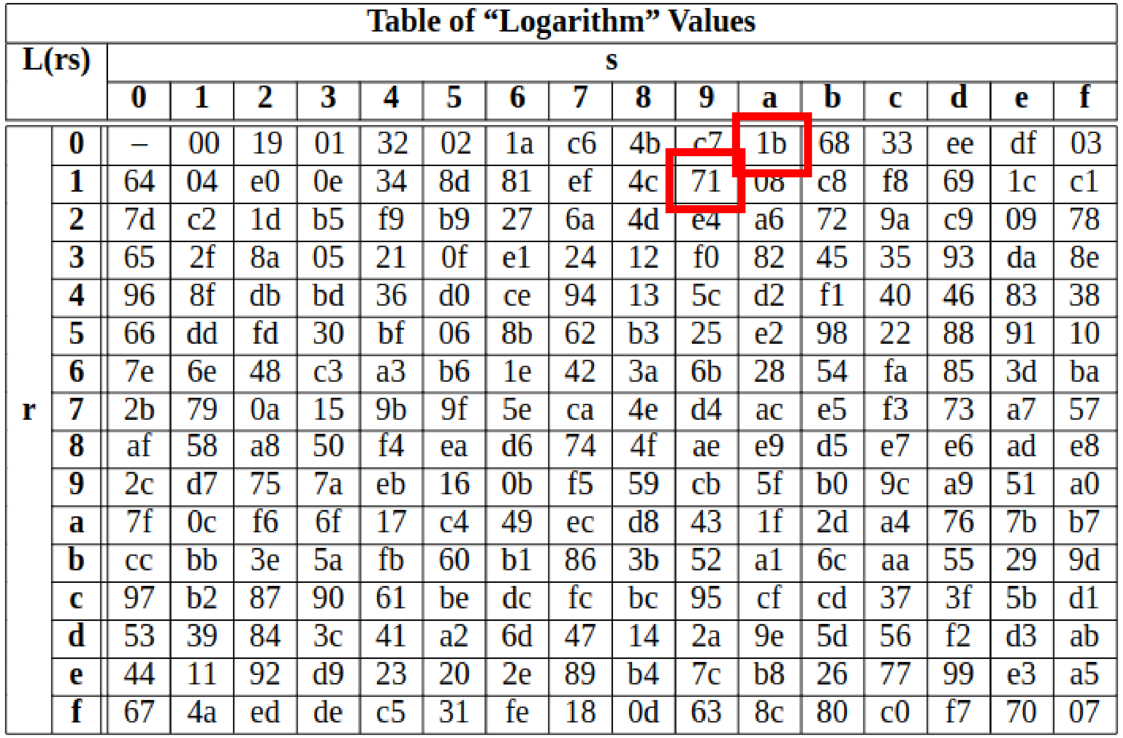


Table of “Logarithm” Values

L(rs)		s															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
r	0	–	00	19	01	32	02	1a	c6	4b	c7	1b	68	33	ee	df	03
	1	64	04	e0	0e	34	8d	81	ef	4c	71	08	c8	f8	69	1c	c1
	2	7d	c2	1d	b5	f9	b9	27	6a	4d	e4	a6	72	9a	c9	09	78
	3	65	2f	8a	05	21	0f	e1	24	12	f0	82	45	35	93	da	8e
	4	96	8f	db	bd	36	d0	ce	94	13	5c	d2	f1	40	46	83	38
	5	66	dd	fd	30	bf	06	8b	62	b3	25	e2	98	22	88	91	10
	6	7e	6e	48	c3	a3	b6	1e	42	3a	6b	28	54	fa	85	3d	ba
	7	2b	79	0a	15	9b	9f	5e	ca	4e	d4	ac	e5	f3	73	a7	57
	8	af	58	a8	50	f4	ea	d6	74	4f	ae	e9	d5	e7	e6	ad	e8
	9	2c	d7	75	7a	eb	16	0b	f5	59	cb	5f	b0	9c	a9	51	a0
	a	7f	0c	f6	6f	17	c4	49	ec	d8	43	1f	2d	a4	76	7b	b7
	b	cc	bb	3e	5a	fb	60	b1	86	3b	52	a1	6c	aa	55	29	9d
	c	97	b2	87	90	61	be	dc	fc	bc	95	cf	cd	37	3f	5b	d1
	d	53	39	84	3c	41	a2	6d	47	14	2a	9e	5d	56	f2	d3	ab
	e	44	11	92	d9	23	20	2e	89	b4	7c	b8	26	77	99	e3	a5
	f	67	4a	ed	de	c5	31	fe	18	0d	63	8c	80	c0	f7	70	07

Multiplication – Table method example

- Step 3: Check the antilog table for the final value (the result was 0x8C)
  - 0x8C -> 0xFA = 250

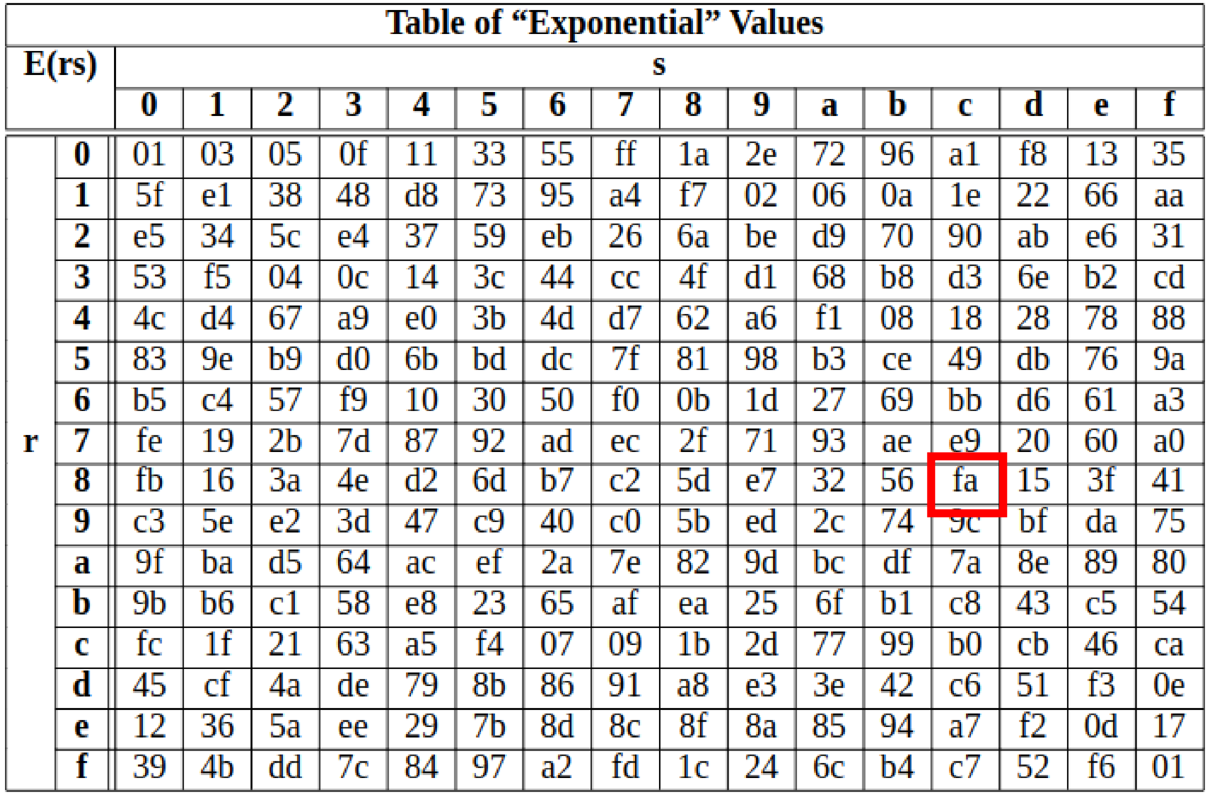


Table of “Exponential” Values

E(rs)		s															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
r	**0**	01	03	05	0f	11	33	55	ff	1a	2e	72	96	a1	f8	13	35
	1	5f	e1	38	48	d8	73	95	a4	f7	02	06	0a	1e	22	66	aa
	2	e5	34	5c	e4	37	59	eb	26	6a	be	d9	70	90	ab	e6	31
	3	53	f5	04	0c	14	3c	44	cc	4f	d1	68	b8	d3	6e	b2	cd
	4	4c	d4	67	a9	e0	3b	4d	d7	62	a6	f1	08	18	28	78	88
	5	83	9e	b9	d0	6b	bd	dc	7f	81	98	b3	ce	49	db	76	9a
	6	b5	c4	57	f9	10	30	50	f0	0b	1d	27	69	bb	d6	61	a3
	7	fe	19	2b	7d	87	92	ad	ec	2f	71	93	ae	e9	20	60	a0
	8	fb	16	3a	4e	d2	6d	b7	c2	5d	e7	32	56	fa	15	3f	41
	9	c3	5e	e2	3d	47	c9	40	c0	5b	ed	2c	74	9c	bf	da	75
	a	9f	ba	d5	64	ac	ef	2a	7e	82	9d	bc	df	7a	8e	89	80
	b	9b	b6	c1	58	e8	23	65	af	ea	25	6f	b1	c8	43	c5	54
	c	fc	1f	21	63	a5	f4	07	09	1b	2d	77	99	b0	cb	46	ca
	d	45	cf	4a	de	79	8b	86	91	a8	e3	3e	42	c6	51	f3	0e
	e	12	36	5a	ee	29	7b	8d	8c	8f	8a	85	94	a7	f2	0d	17
	f	39	4b	dd	7c	84	97	a2	fd	1c	24	6c	b4	c7	52	f6	01

Multiplication – Table method issues

- Although we only need 3 lookups…
- It **does not scale** with respect to memory:
  - GF(2^8) -> 256 values, 1B each value * 2 tables = 256 Bytes per table
  - GF(2^128) -> 2^128 values, 16B each * 2 tables = **10^39 Bytes of memory!**

(NB: 1 Petabyte = 10^15 bytes)

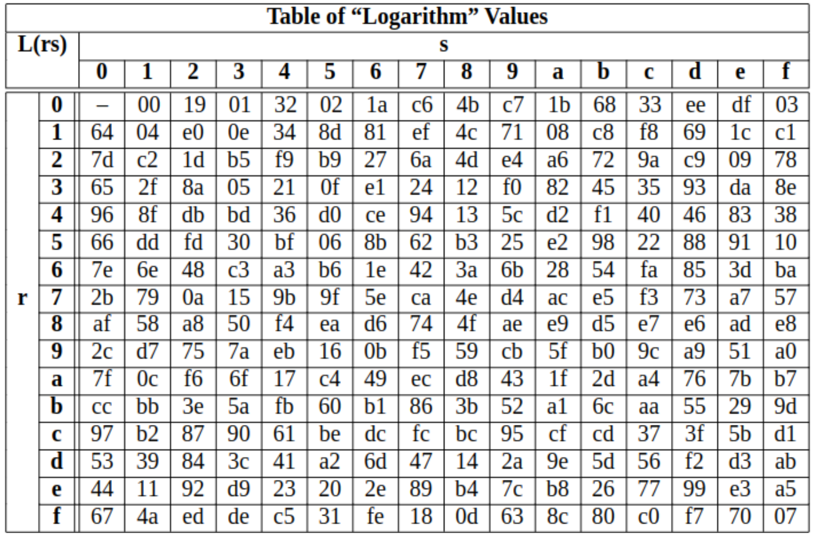


Table of "Logarithm" Values

L(rs)		s															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
	0	–	00	19	01	32	02	1a	c6	4b	c7	1b	68	33	ee	df	03
	1	64	04	e0	0e	34	8d	81	ef	4c	71	08	c8	f8	69	1c	c1
	2	7d	c2	1d	b5	f9	b9	27	6a	4d	e4	a6	72	9a	c9	09	78
	3	65	2f	8a	05	21	0f	e1	24	12	f0	82	45	35	93	da	8e
	4	96	8f	db	bd	36	d0	ce	94	13	5c	d2	f1	40	46	83	38
	5	66	dd	fd	30	bf	06	8b	62	b3	25	e2	98	22	88	91	10
	6	7e	6e	48	c3	a3	b6	1e	42	3a	6b	28	54	fa	85	3d	ba
r	**7**	2b	79	0a	15	9b	9f	5e	ca	4e	d4	ac	e5	f3	73	a7	57
	8	af	58	a8	50	f4	ea	d6	74	4f	ae	e9	d5	e7	e6	ad	e8
	9	2c	d7	75	7a	eb	16	0b	f5	59	cb	5f	b0	9c	a9	51	a0
	a	7f	0c	f6	6f	17	c4	49	ec	d8	43	1f	2d	a4	76	7b	b7
	b	cc	bb	3e	5a	fb	60	b1	86	3b	52	a1	6c	aa	55	29	9d
	c	97	b2	87	90	61	be	dc	fc	bc	95	cf	cd	37	3f	5b	d1
	d	53	39	84	3c	41	a2	6d	47	14	2a	9e	5d	56	f2	d3	ab
	e	44	11	92	d9	23	20	2e	89	b4	7c	b8	26	77	99	e3	a5
	f	67	4a	ed	de	c5	31	fe	18	0d	63	8c	80	c0	f7	70	07

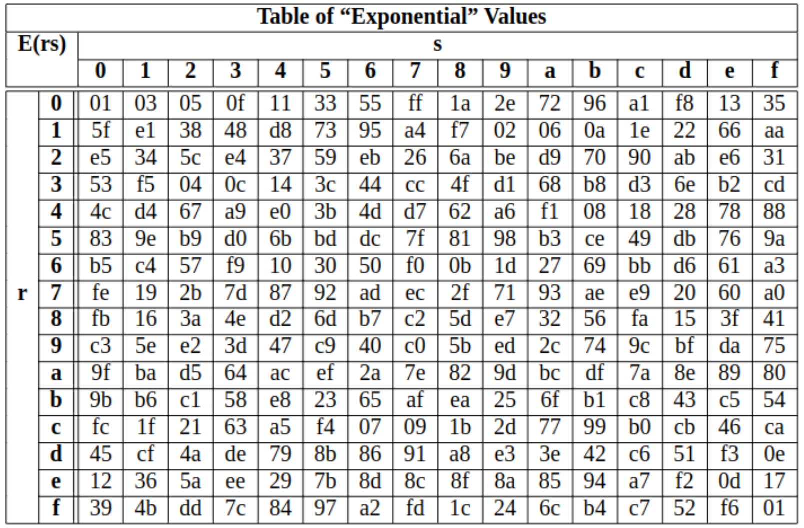
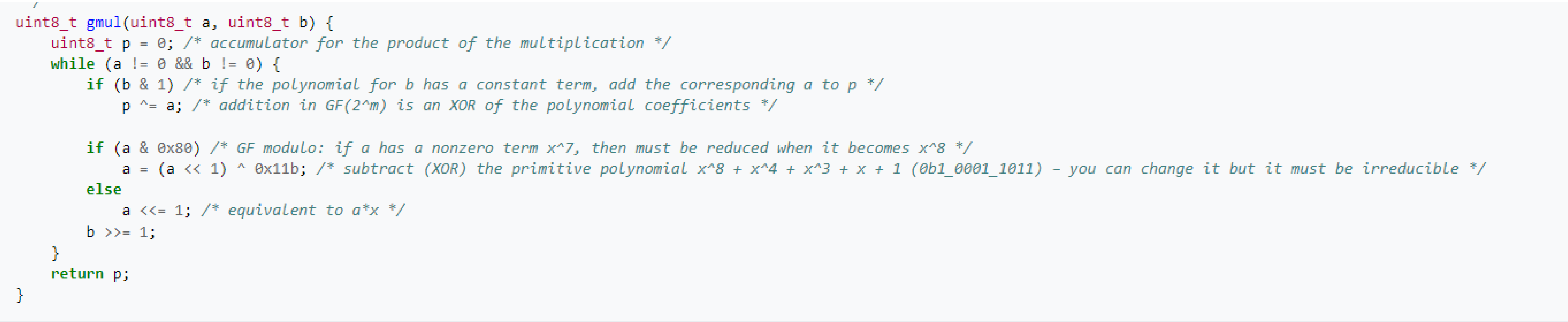


Table of "Exponential" Values

E(rs)		s															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
	0	01	03	05	0f	11	33	55	ff	1a	2e	72	96	a1	f8	13	35
	1	5f	e1	38	48	d8	73	95	a4	f7	02	06	0a	1e	22	66	aa
	2	e5	34	5c	e4	37	59	eb	26	6a	be	d9	70	90	ab	e6	31
	3	53	f5	04	0c	14	3c	44	cc	4f	d1	68	b8	d3	6e	b2	cd
	4	4c	d4	67	a9	e0	3b	4d	d7	62	a6	f1	08	18	28	78	88
	5	83	9e	b9	d0	6b	bd	dc	7f	81	98	b3	ce	49	db	76	9a
	6	b5	c4	57	f9	10	30	50	f0	0b	1d	27	69	bb	d6	61	a3
r	**7**	fe	19	2b	7d	87	92	ad	ec	2f	71	93	ae	e9	20	60	a0
	8	fb	16	3a	4e	d2	6d	b7	c2	5d	e7	32	56	fa	15	3f	41
	9	c3	5e	e2	3d	47	c9	40	c0	5b	ed	2c	74	9c	bf	da	75
	a	9f	ba	d5	64	ac	ef	2a	7e	82	9d	bc	df	7a	8e	89	80
	b	9b	b6	c1	58	e8	23	65	af	ea	25	6f	b1	c8	43	c5	54
	c	fc	1f	21	63	a5	f4	07	09	1b	2d	77	99	b0	cb	46	ca
	d	45	cf	4a	de	79	8b	86	91	a8	e3	3e	42	c6	51	f3	0e
	e	12	36	5a	ee	29	7b	8d	8c	8f	8a	85	94	a7	f2	0d	17
	f	39	4b	dd	7c	84	97	a2	fd	1c	24	6c	b4	c7	52	f6	01

Multiplication – RPA

- Use number decomposition to achieve the result
  - Russian Peasant Algorithm (RPA)
- No lookups necessary – a compute intensive approach

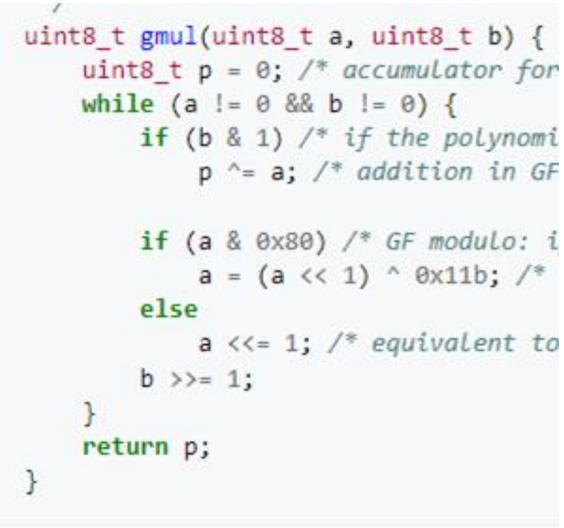


```c
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for the product of the multiplication */
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomial for b has a constant term, add the corresponding a to p */
            p ^= a; /* addition in GF(2^m) is an XOR of the polynomial coefficients */

        if (a & 0x80) /* GF modulo: if a has a nonzero term x^7, then must be reduced when it becomes x^8 */
            a = (a << 1) ^ 0x11b; /* subtract (XOR) the primitive polynomial x^8 + x^4 + x^3 + x + 1 (0b1_0001_1011) – you can change it but it must be irreducible */
        else
            a <<= 1; /* equivalent to a*x */
        b >>= 1;
    }
    return p;
}
```

Multiplication – RPA example

- Let’s multiply 10 by 25 using this method



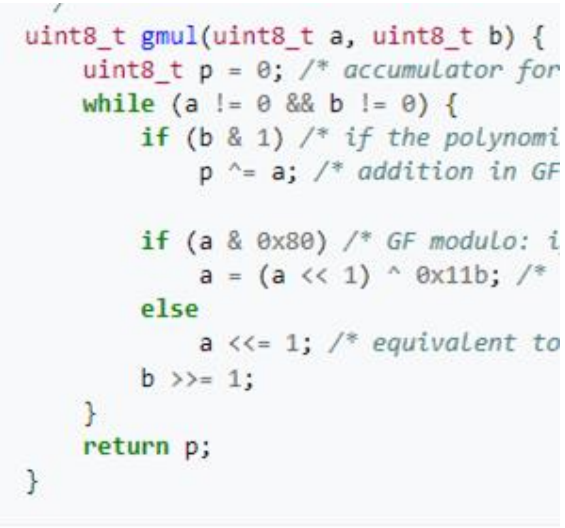
```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

Iteration 1: $a = 10 = 0b01010$ $b = 25 = 0b11001$ $p = 0$

Multiplication – RPA example

- Let’s multiply 10 by 25 using this method



```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: 
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

Iteration 1: $a = 10 = 0b01010$ $b = 25 = 0b11001$ $p = 0b01010$

$0b11001 \; \& \; 1$ ✓ $p = p \; XOR \; a$

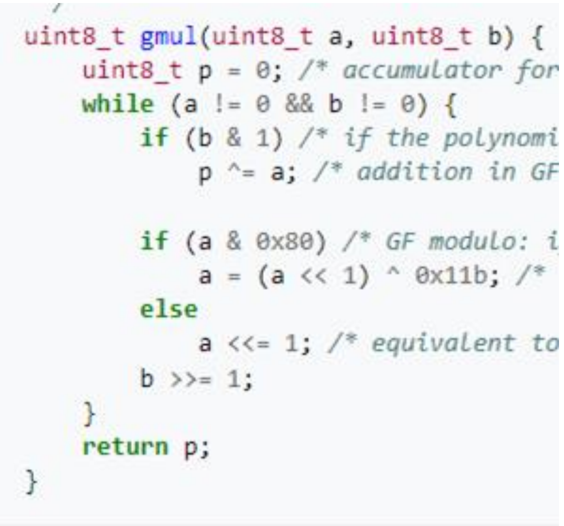
Multiplication – RPA example

- Let’s multiply 10 by 25 using this method

Iteration 1: $a = 0b10100$ $b = 0b01100$ $p = 0b01010$

$0b11001 \ \& \ 1$ ✓ $p = p \ XOR \ a$

$0b01010 \ \& \ 0x80$ ✗



```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

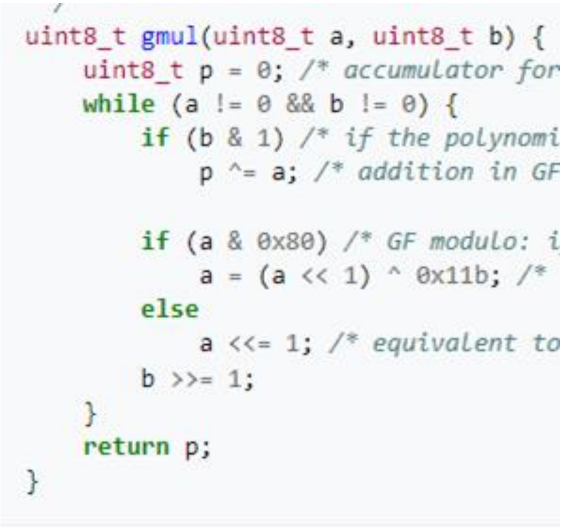
Multiplication – RPA example

- Let’s multiply 10 by 25 using this method

Iteration 2: $a = 0b10100$ $b = 0b01100$ $p = 0b01010$

$0b01100 \,\&\, 1$ ✗

$0b10100 \,\&\, 0x80$ ✗

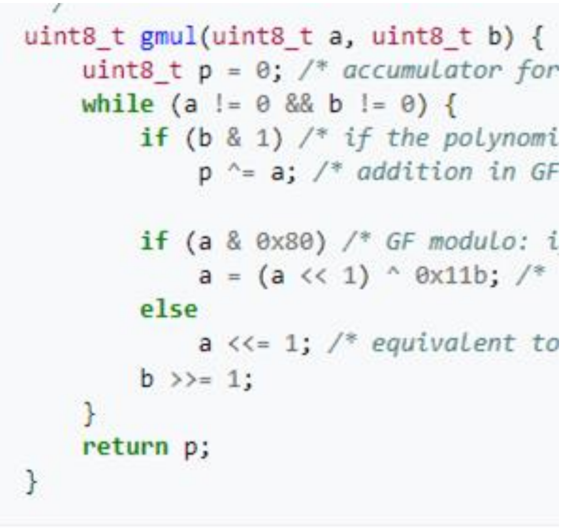


```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

Multiplication – RPA example

- Let’s multiply 10 by 25 using this method



```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

Iteration 2: $a = 0b101000$ $b = 0b00110$ $p = 0b01010$

$0b01100 \ \& \ 1$ ❌

$0b10100 \ \& \ 0x80$ ❌

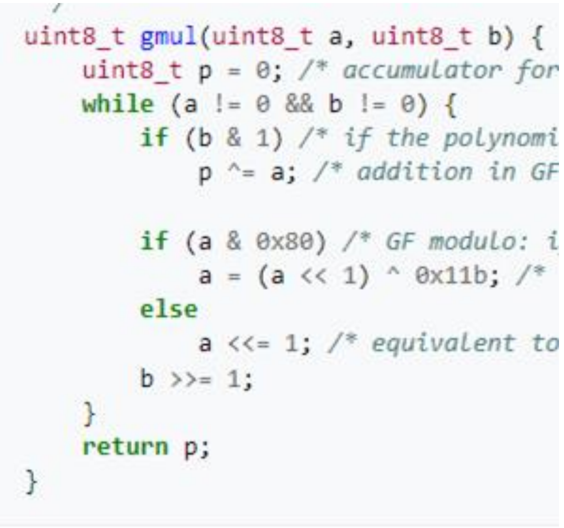
Multiplication – RPA example

- Let’s multiply 10 by 25 using this method

Iteration 3: $a = 0b101000$ $\quad b = 0b00110$ $\quad p = 0b01010$

$0b00110 \ \& \ 1$ ✖

$0b101000 \ \& \ 0x80$ ✖



```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

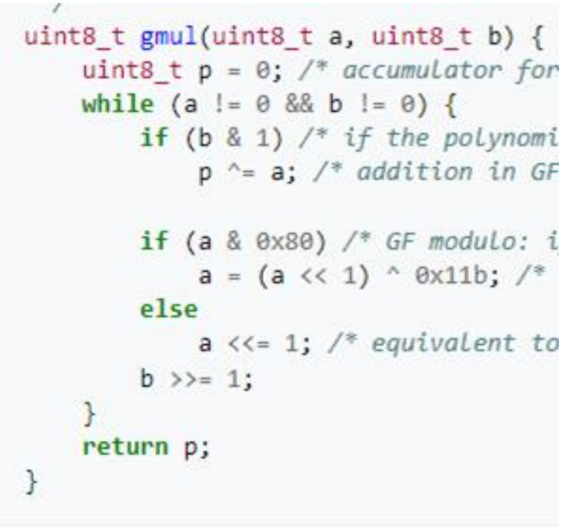
Multiplication – RPA example

- Let’s multiply 10 by 25 using this method

Iteration 3: $a = 0b1010000$ $b = 0b00011$ $p = 0b01010$

$0b00110 \ \& \ 1$ ✗

$0b101000 \ \& \ 0x80$ ✗

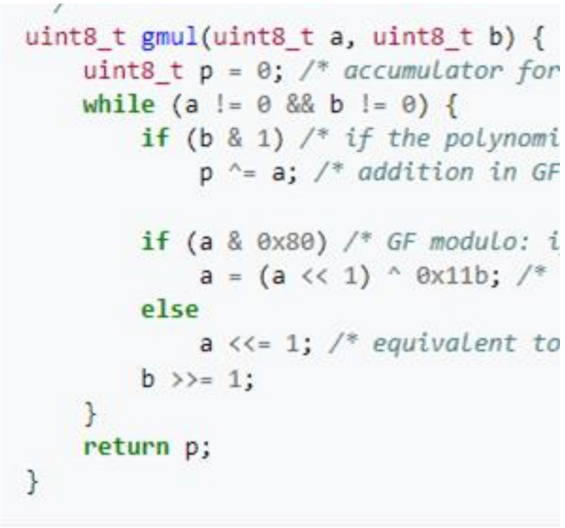


```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

Multiplication – RPA example

- Let’s multiply 10 by 25 using this method



```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

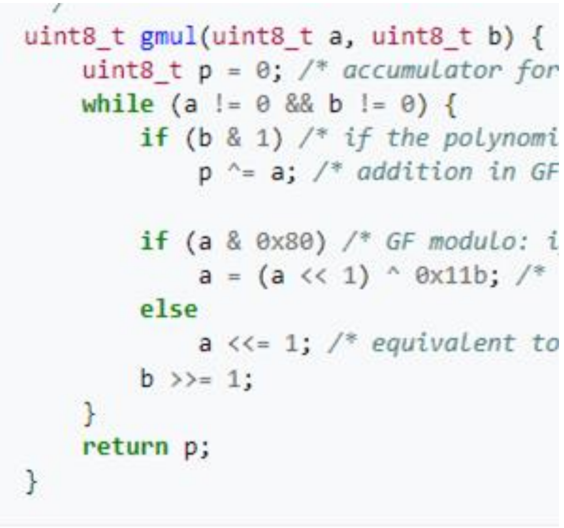
Iteration 4: $a = 0b1010000$ $b = 0b00011$ $p = 0b1011010$

$0b00011 \,\&\, 1$ ✓ $p = p \, XOR \, a$

$0b1010000 \,\&\, 0x80$ ✗

Multiplication – RPA example

- Let’s multiply 10 by 25 using this method



```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

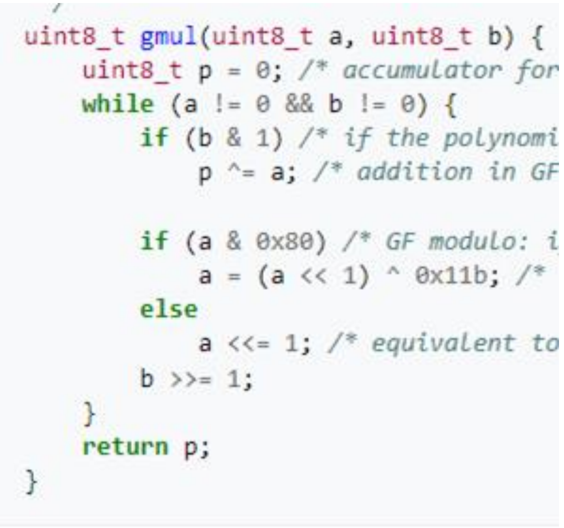
Iteration 4: $a = 0b10100000 \quad b = 0b00001 \quad p = 0b1011010$

$0b00011 \ \& \ 1$ ✓ $\quad p = p \, XOR \, a$

$0b1010000 \ \& \ 0x80$ ✗

Multiplication – RPA example

- Let’s multiply 10 by 25 using this method



```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

Iteration 5: $a = 0b10100000$ $b = 0b00001$ $p = 0b11111010$

$0b00001 \,\&\, 1$ ✓ $p = p\ XOR\ a$

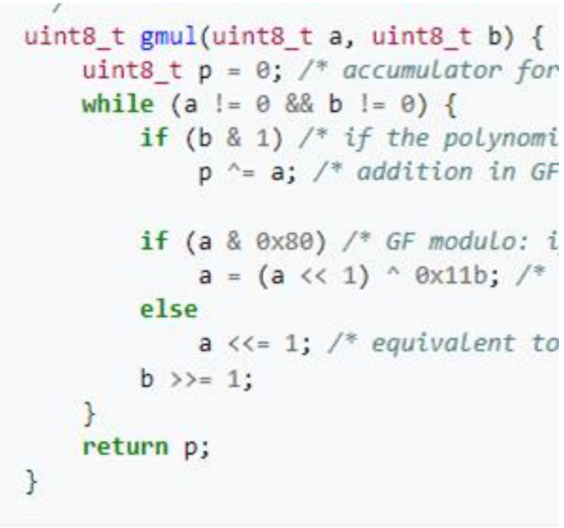
Multiplication – RPA example

- Let's multiply 10 by 25 using this method

Iteration 5: $a = 0b1011011$ $\quad b = 0b00001$ $\quad p = 0b11111010$

$0b00001 \,\&\, 1$ ✓ $\quad p = p\, XOR\, a$

$0b10100000 \,\&\, 0x80$ ✓ $\quad a = (a << 1)\hat{}0x11b$



```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

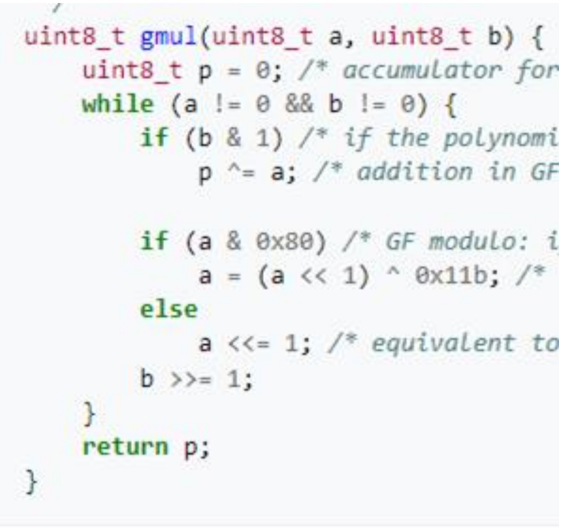
Multiplication – RPA example

- Let’s multiply 10 by 25 using this method

Iteration 5: $a = 0b1011011 \quad b = 0b00000 \quad p = 0b11111010$

$0b00001 \;\&\; 1$ ✓ $p = p \; XOR \; a$

$0b10100000 \;\&\; 0x80$ ✓ $a = (a << 1)\hat{}0x11b$

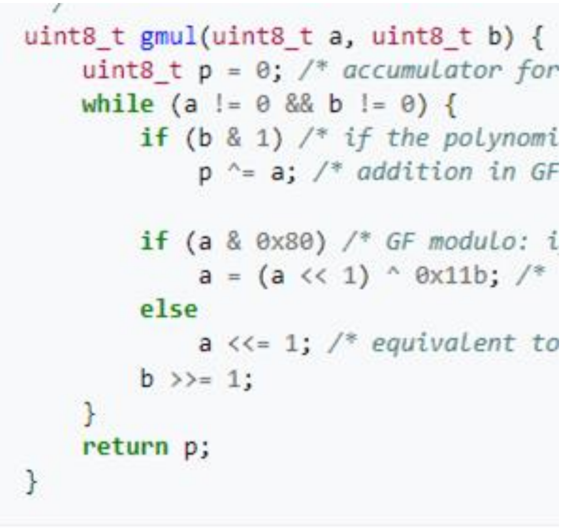


```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

Multiplication – RPA example

- Let’s multiply 10 by 25 using this method



```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

Iteration 6: $a = 0b1011011$ $\boxed{b = 0b000000}$ $p = 0b11111010$

b is 0 so we are done

$$p = 0b11111010$$

250

Multiplication – RPA issues

- Problem: computation has dependencies requiring **many pipeline stages**
  - Larger Fields -> More iterations
- Some good news: number of stages **scale linearly** with the field size!
  - Also, no memory needed for log/antilog tables
- However, implementations over large fields are **not suitable for current Tofino switches**
  - Our current proof of concept consumes 16 stages for multiplication in GF(2^8)

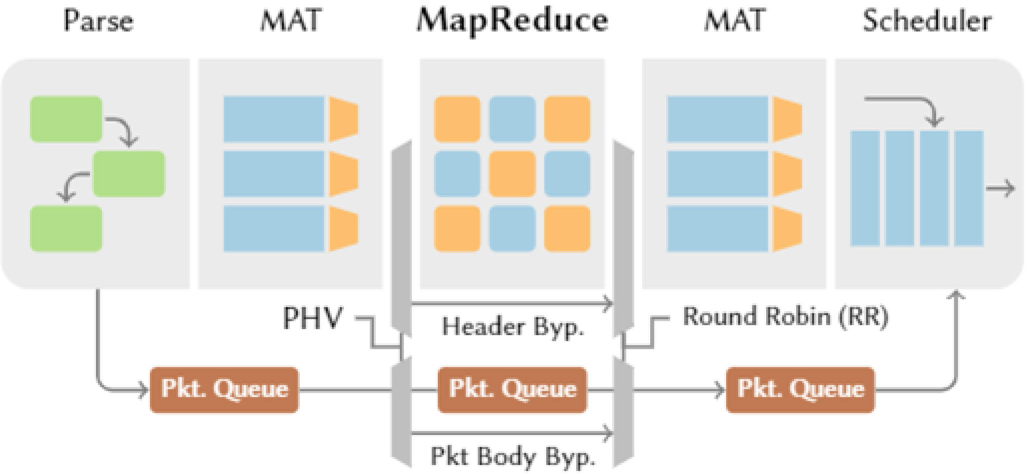
A way forward

A way forward

- Modern switch architectures are not enough for generic finite field operations (i.e., for large field sizes)
- However, other switch architectures have been proposed recently
- Question is: can we leverage any to perform Finite Field operations?
  - A preliminary investigation led us to **Taurus [ASPLOS’22]** as a good candidate

A way forward – Taurus

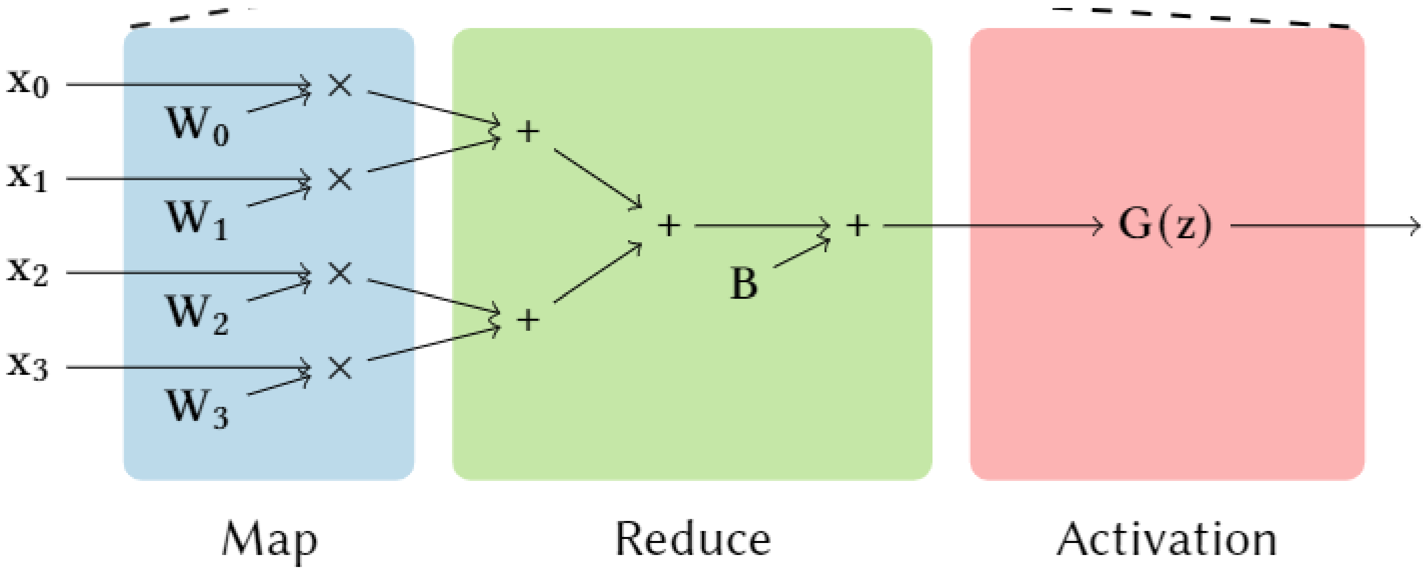
- Data plane architecture for running **ML inference per packet**
- MapReduce abstraction
  - VLIW (Current) vs SIMD (New)
- Parsing, Pre-Processing, Post-Processing and Scheduling all done like common architectures



ASPLOS'22 paper: Taurus: a data plane architecture for per-packet ML | Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems

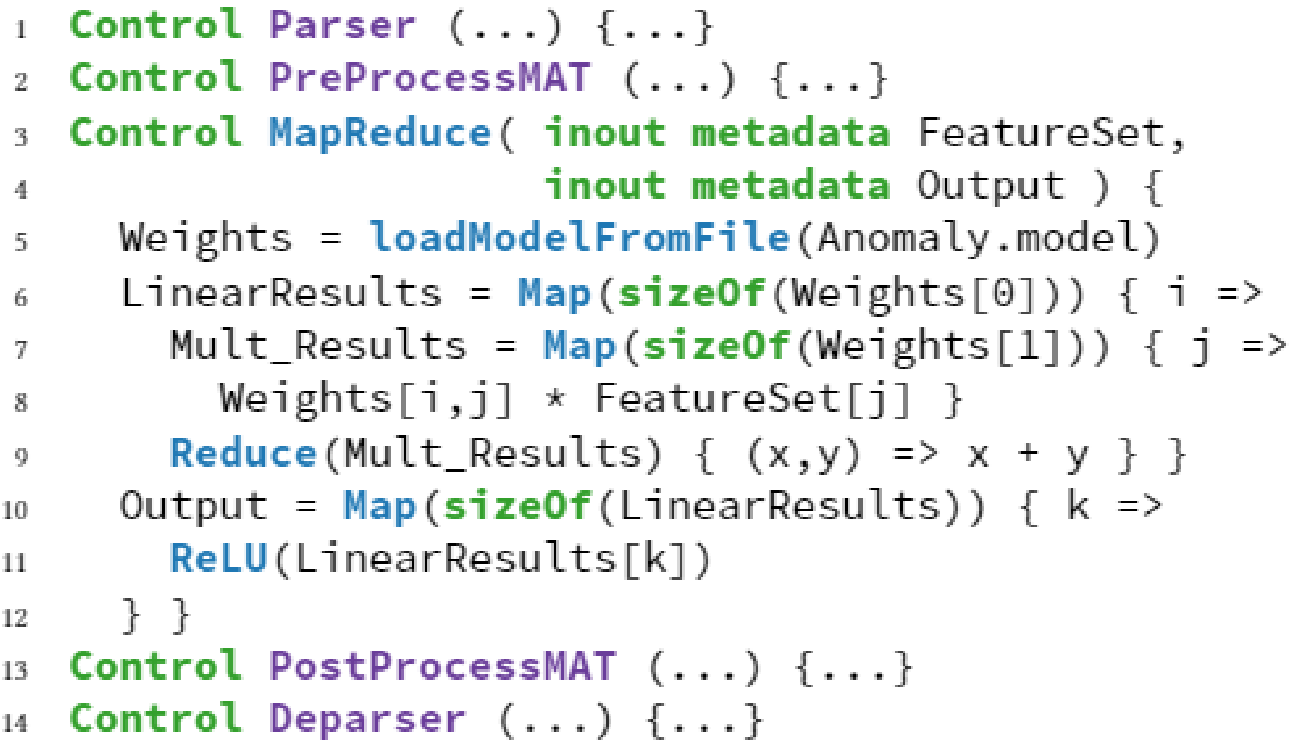
Taurus - MapReduce

- Map Operations
  - Element-wise vector operations (addition, multiplication, etc)
- Reduce Operations
  - Combine a vector of elements into a single scalar value
- Example:



Taurus - MapReduce

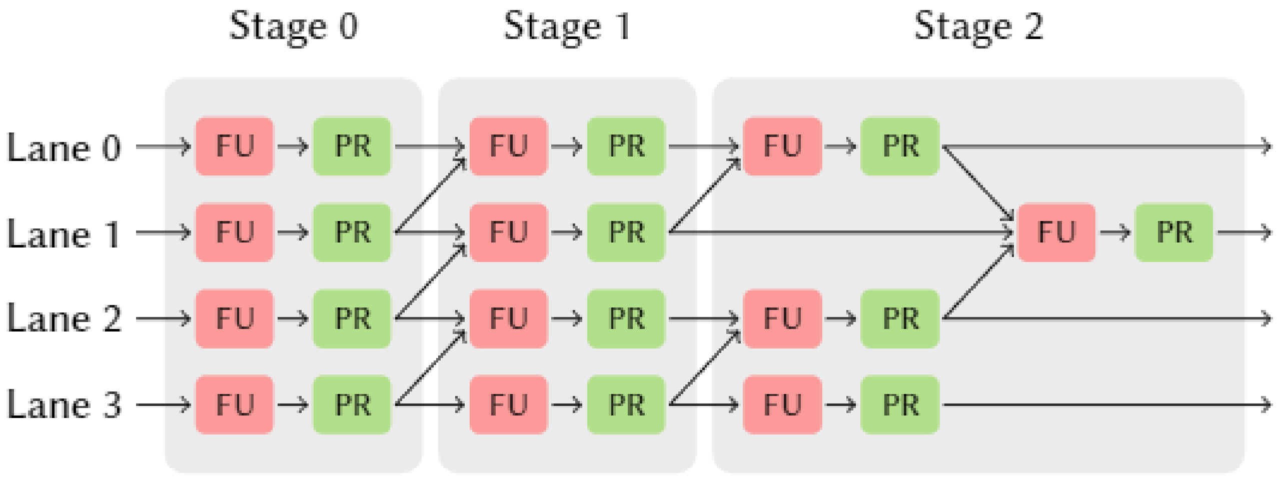
- MapReduce control block in P4



```
Control Parser (...) {...}
Control PreProcessMAT (...) {...}
Control MapReduce( inout metadata FeatureSet,
                   inout metadata Output ) {
  Weights = loadModelFromFile(Anomaly.model)
  LinearResults = Map(sizeOf(Weights[0])) { i =>
    Mult_Results = Map(sizeOf(Weights[1])) { j =>
      Weights[i,j] * FeatureSet[j] }
    Reduce(Mult_Results) { (x,y) => x + y } }
  Output = Map(sizeOf(LinearResults)) { k =>
    ReLU(LinearResults[k])
  } }
Control PostProcessMAT (...) {...}
Control Deparser (...) {...}
```

Taurus – CUs and MUs

- MapReduce based on a CGRA, Plasticine [ISCA’17]
- Compute Units (CUs)
  - Composed of Functional Units (FU) and Pipeline Registers (PR)
  - Lanes
  - Stages

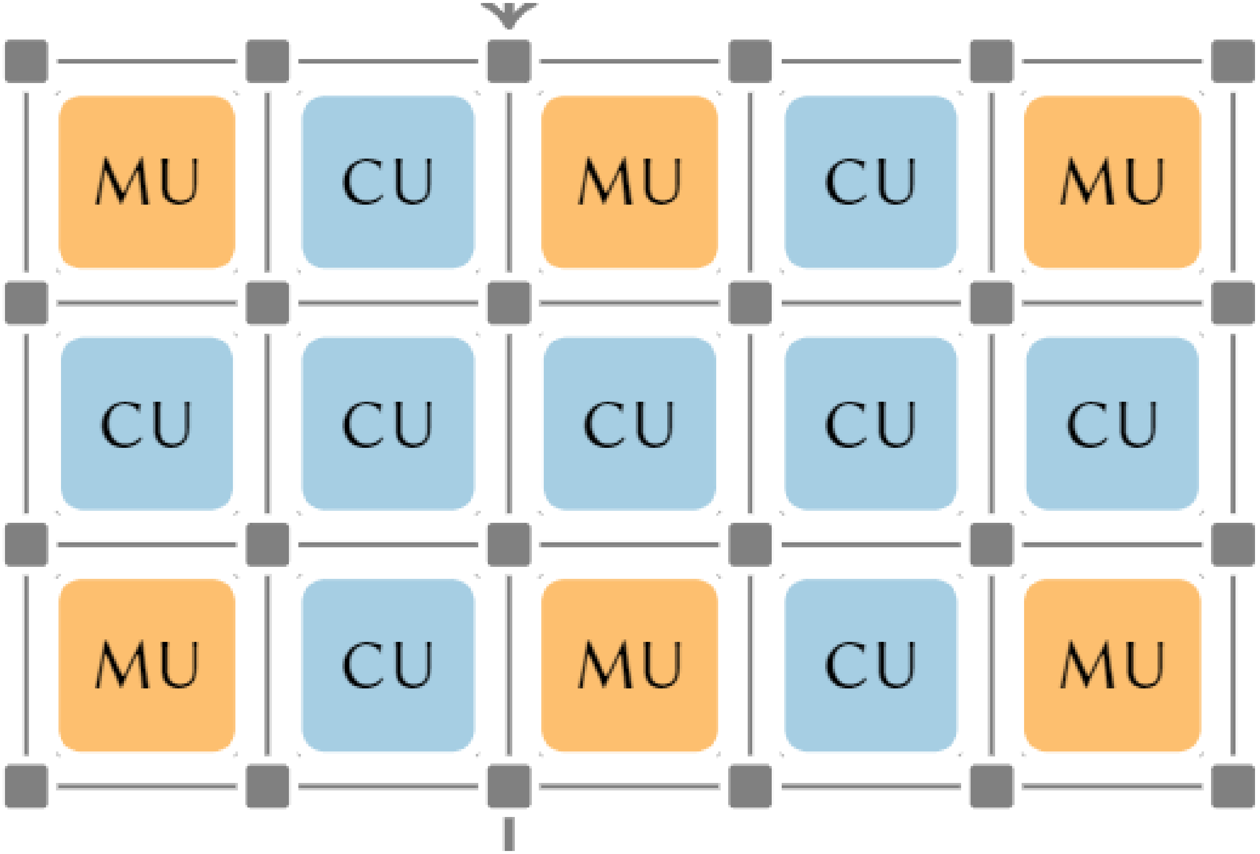


Taurus – CUs and MUs

- Memory Units (MUs)
  - Banked SRAMs
  - Interspersed with CUs
- Act like coarse grain Pipeline Registers
- At 1GHz, ensures nano-second level latencies
    - Requirement for modern Tbps switches!

Taurus – Full Mesh

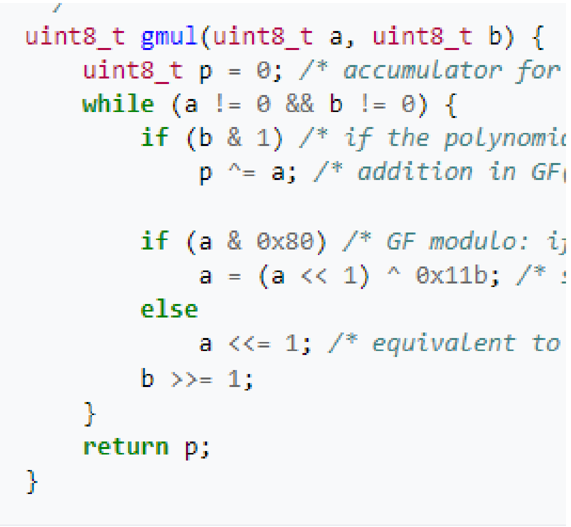
- A full mesh of CUs and MUs



Current research question

- Can we leverage the CUs to execute the iterations required by the RPA algorithm?
- Leveraging SIMD parallelism to perform multiple operations in parallel

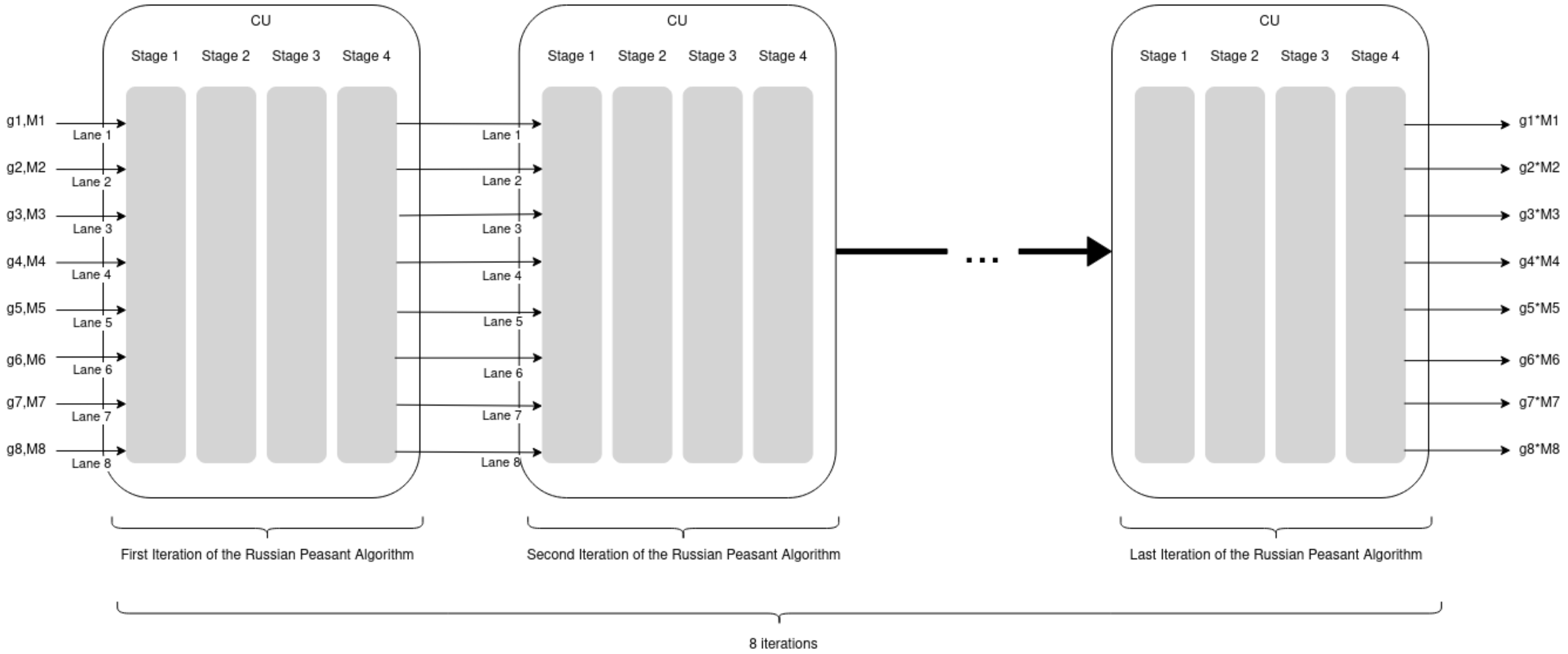
Finite Field Multiplication (RPA)



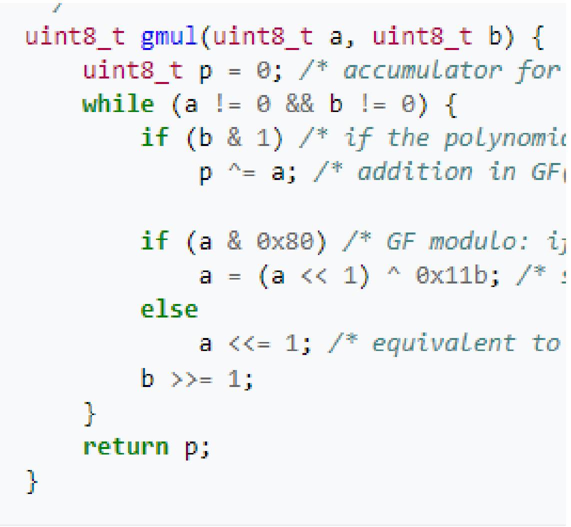
```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

- RPA in an iterative algorithm
- Each CU can be in charge of one iteration
    - 8 CUs -> GF(256); 16 CUs -> GF(65536)
- Number of lanes dictate how many multiplications can be done in parallel



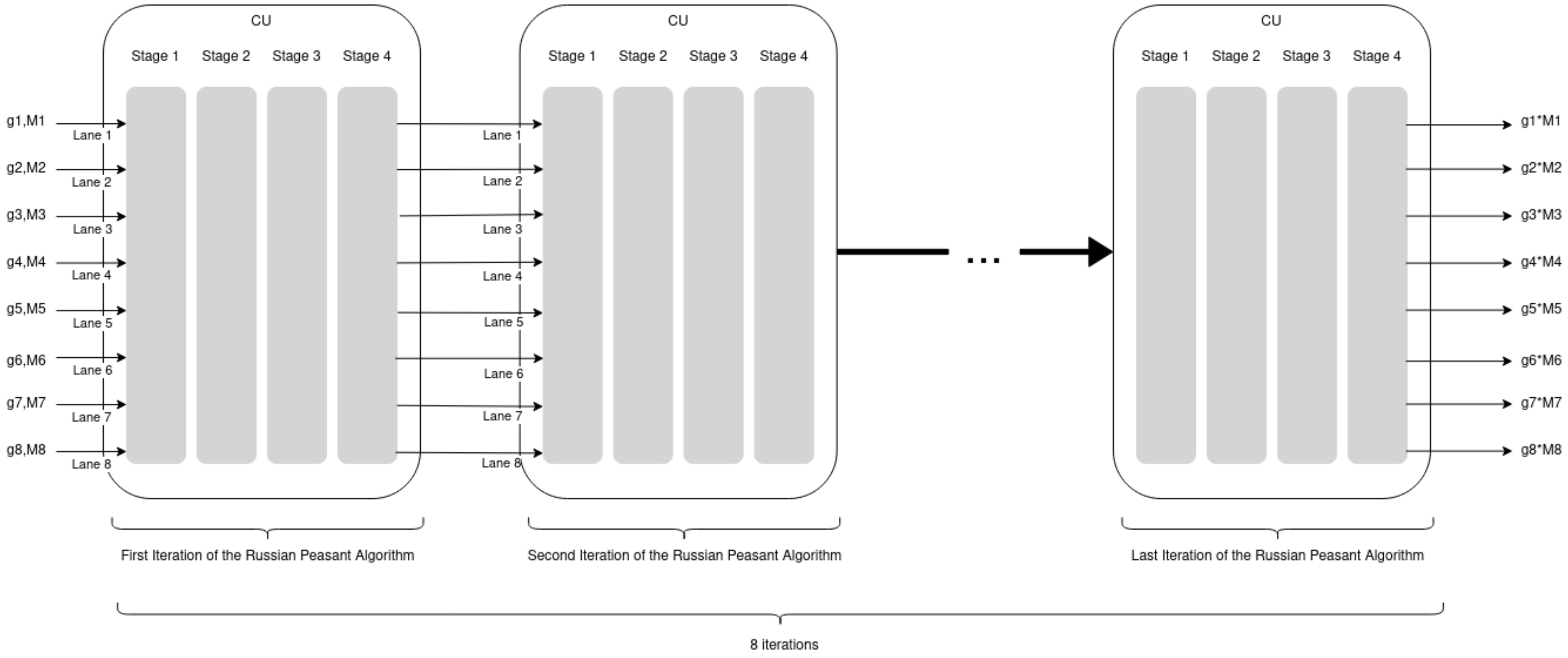
Finite Field Multiplication



```
uint8_t gmul(uint8_t a, uint8_t b) {
    uint8_t p = 0; /* accumulator for
    while (a != 0 && b != 0) {
        if (b & 1) /* if the polynomi
            p ^= a; /* addition in GF

        if (a & 0x80) /* GF modulo: i
            a = (a << 1) ^ 0x11b; /*
        else
            a <<= 1; /* equivalent to
        b >>= 1;
    }
    return p;
}
```

- Number of stages per CU is also configurable
  - One CU might be able to perform 2 iterations of RPA
  - That cuts the number of CUs needed in half



Next Steps

- We are currently working on a Proof of Concept that runs RPA in Taurus (or an architecture based on Taurus)

- Next step is to investigate the division operation.

- We have found an algorithm capable of finding the inverse of a number and are currently working on an implementation

Conclusion/Q&A

- Primitives for Finite Field operations are required by many net applications
    - crypto
    - network coding
    - etc.
- Current switch architectures make it hard to implement FF with large fields, due to memory and/or computational constraints

- New architectures (Taurus-based?) are a solution worth exploring

Thank You