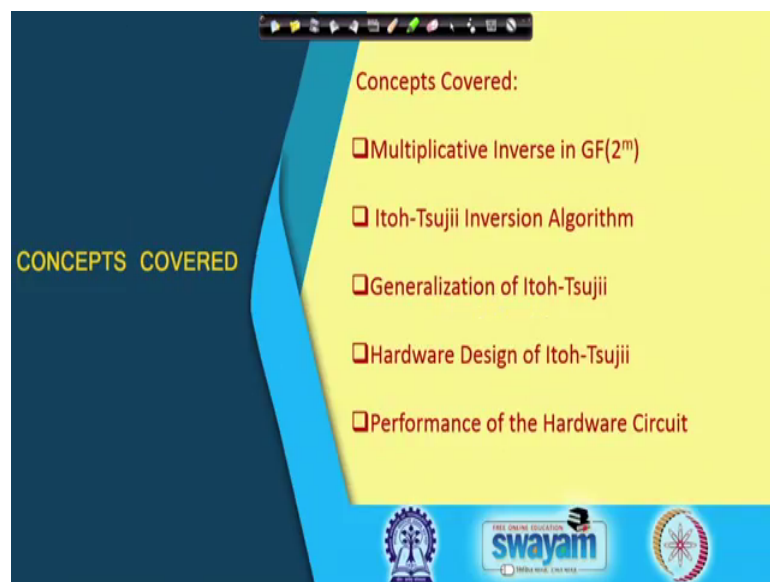


Hardware Security
Prof. Debdeep Mukhopadhyay
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 06
Hardware Design for Finite Inverse

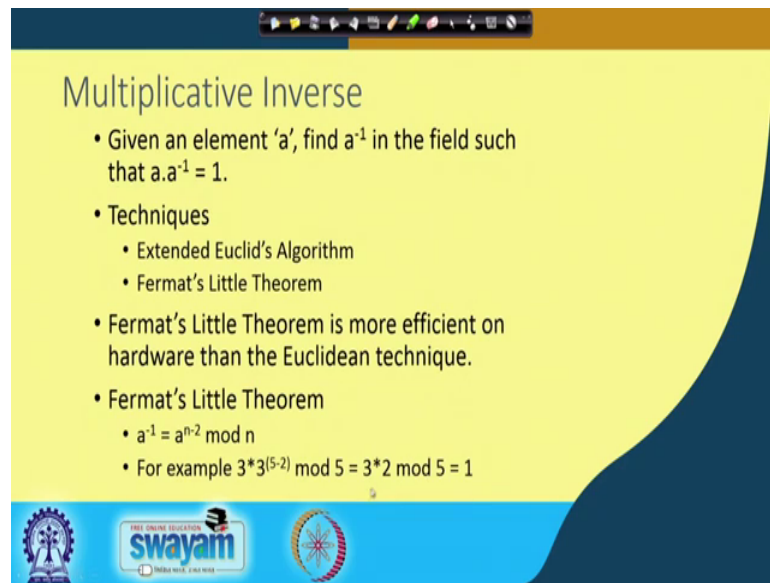
So, welcome to this class on Hardware Security. So, today we shall be talking about different topic, we will be talking about a finite field architecture like last class we discussed about multipliers. In today's class, we will discuss about another important class of arithmetic circuits in finite fields, we are called as finite field inverse. So, how to compute finite field inversions?

(Refer Slide Time: 00:39)



So, in more specifically, we shall be talking about multiplicative inverse in GF 2 to the power of m, which is a characteristic to field. We shall be talking about Itoh-Tsujii inversion algorithm, we shall be talking about a generalization of the Itoh-Tsujii inversion algorithm. And time permitting we shall be discussing about hardware design and also performance of the corresponding hardware circuit.

(Refer Slide Time: 01:03)



The slide is titled "Multiplicative Inverse" and contains the following text:

- Given an element 'a', find a^{-1} in the field such that $a \cdot a^{-1} = 1$.
- Techniques
 - Extended Euclid's Algorithm
 - Fermat's Little Theorem
- Fermat's Little Theorem is more efficient on hardware than the Euclidean technique.
- Fermat's Little Theorem
 - $a^{-1} = a^{n-2} \pmod n$
 - For example $3 \cdot 3^{(5-2)} \pmod 5 = 3 \cdot 2 \pmod 5 = 1$

At the bottom of the slide, there are logos for "swayam" and "INDIA WISE, LEAD WISE".

So, to start with how do we compute a multiplicative inverse or what is the definition of a multiplicative inverse? Multiplicative inverse means that given an element a , we want to find out its inverse, which is denoted as a^{-1} or a to the power of minus 1 in the field such that if I multiply a with a^{-1} , I get the unit or I get the unit element in the field or I get 1.

So, to start so you know like there are different techniques for finite field inversion computations. The most important ones I have kind of written here like the first one is the extended Euclidean algorithm, which is a more generalization of the Euclidean algorithm, which is used to compute the greatest common divisor of two numbers. There is another algorithm or there is another result, which is very famously known as the Fermat's little theorem, which also can be used to compute multiplicative inversions.

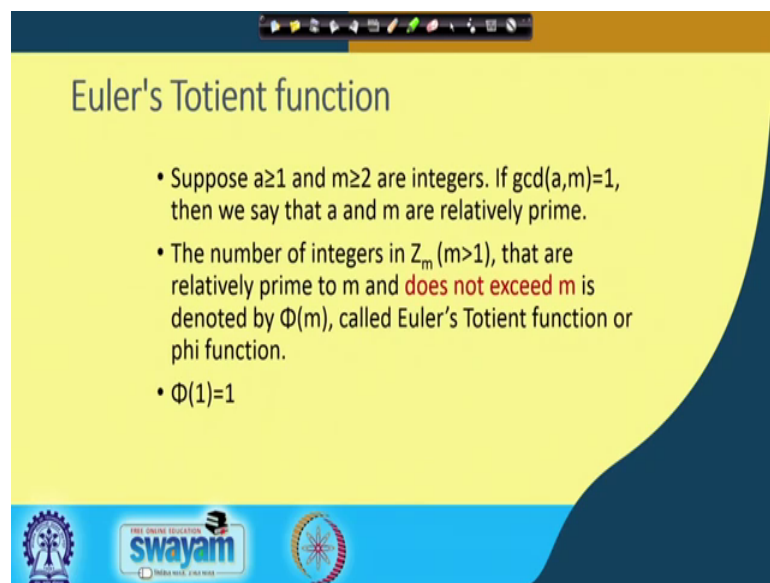
So, the (Refer Time: 01:58) here is that the Fermat's little theorem is more efficient on hardware than the Euclidean technique. Because, the Euclidean technique as you probably know is that it will be it will be resulting in a in a computation, which requires more intensive operations, and also essentially will require more number of clock cycles.

So, essentially right we resort to Fermat's little theorem and let us state how a Fermat's little theorem works. So, when we want to compute a^{-1} , the idea is that and I want to do a modulo of the field, then our modulo of the size of the field. Then

what we do is we try to compute a to the power of $n-1$ as a to the power of $n-2$ modulo n . Note here that the definition of n is such that n can be any number as such, but a has to be co-prime with n that means, the greatest common divisor of a and n has to be 1 or in another sense, we can also say this as a belongs to the multiplicative group, which is created by n , which is denoted of n as \mathbb{Z}_n^* ok.

So, you can also say for simplicity that let us choose n as a prime number in which case all the elements, which are non-zero like starting from 1 to $n-1$ belong to it is multiplicative group. So, just let us take a small example. So, suppose I take n equal to 5, which is a prime number, so then you can easily see that 3 to the power of 5 minus 2 modulo 5 should be it is inverse, you can check it that this turns out to be 2. And therefore, if I multiply 3 with 2, then I get 6 which modulo 5 is 1 ok. So, therefore 2 is indeed the multiplicative inverse of 3 modulo 5.

(Refer Slide Time: 03:43)



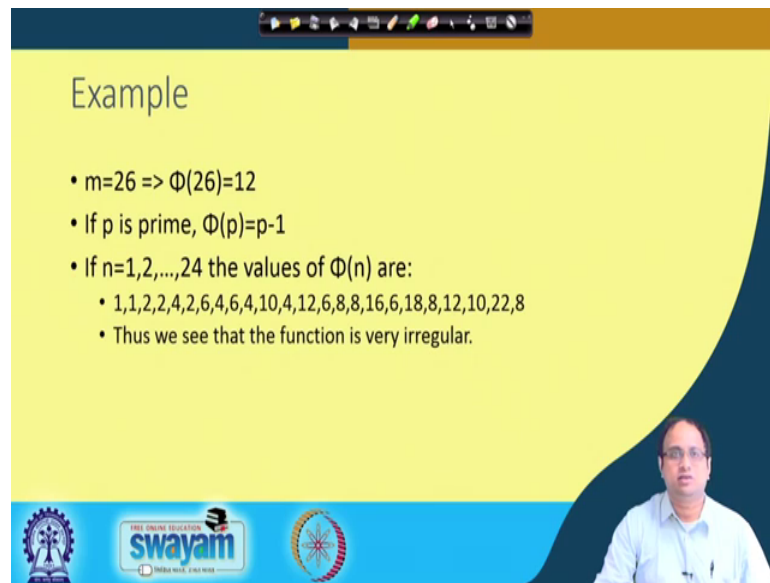
The slide is titled "Euler's Totient function" and contains the following text:

- Suppose $a \geq 1$ and $m \geq 2$ are integers. If $\gcd(a, m) = 1$, then we say that a and m are relatively prime.
- The number of integers in \mathbb{Z}_m ($m > 1$), that are relatively prime to m and **does not exceed m** is denoted by $\Phi(m)$, called Euler's Totient function or phi function.
- $\Phi(1) = 1$

At the bottom of the slide, there are logos for "swayam" and "INDIA'S EDUCATION DIGITALIZATION" along with a circular logo featuring a gear and a book.

So, just to generalize this, we the generalization is based upon a particular parameter a particular function, which is called as the Euler's Totient function often denoted as ϕ of m ok. So, suppose that a and m are integers such that the greatest common divisor of a and m is 1 that means, a and m are mutually co-prime or relatively prime. And then the number of integers in \mathbb{Z}_m that are relatively prime to m and does not exceed m is often denoted as ϕ m ok. And this is called also as the Euler's Totient function or the phi function ok.

(Refer Slide Time: 04:29)



Example

- $m=26 \Rightarrow \Phi(26)=12$
- If p is prime, $\Phi(p)=p-1$
- If $n=1,2,\dots,24$ the values of $\Phi(n)$ are:
 - 1,1,2,2,4,2,6,4,6,4,10,4,12,6,8,8,16,6,18,8,12,10,22,8
 - Thus we see that the function is very irregular.

swayam

So, just to define write the phi of 1 is taken as 1, this is an initialize the phi function ok. And subsequently write if I for example, 1 to compute 26, m equal to 26, and I want to calculate phi of 26, then it turns out to be 12 that means, there are 12 numbers are co-prime with 26. If p is prime, then it is natural to understand that phi of p will be phi will be p minus 1 ok, just as we saw in the example.

And if n is equal to let us say here, I have just tried to kind of tabulate the values of phi n . So, you will see that phi n is a pretty much irregular function ok, it is neither monotonically increasing nor monotonically decreasing, it is kind of an erratic function. But, luckily like if you know the prime factors of n then, you can compute phi n quite efficiently ok.

(Refer Slide Time: 05:11)

The slide is titled "Fermat's Little Theorem and Finite Field Inverse". It contains three bullet points:

- If $\gcd(a,m)=1$, then $a^{\phi(m)} \equiv 1 \pmod{m}$
- If $m=p$, is a prime number, thus $a^{p-1} \equiv 1 \pmod{p}$
- For the field $GF(2^m)$, thus we have: $a^{2^m-1} \equiv 1 \Rightarrow a^{-1} \equiv a^{2^m-2}$

The slide also features a video feed of a presenter in the bottom right corner and logos for Swamyam and other educational institutions at the bottom.

However, I am not going into that definition here, but rather we will take a look into Fermat's little theorem, which is stated here using the phi function ok. So, for example if I take the greatest common divisor of a and m as 1 ok and then I can write a to the power of phi m is congruent to 1 modulo m, this equation is also called as the Euler's function or Euler's theorem ok. Fermat's little theorem is actually a special case of Euler's theorem, where m is a p which is a prime number ok. And if it is a prime number, then I can write phi of p as p minus 1, so what we get is a to the power of p minus 1 is congruent to 1 modulo p.

If you have the field, which is GF to the power of m, therefore right where there are 2^m elements in the field. So, this is the extension field that we saw in the last class ok, where you extend from GF 2 to the GF 2^m using an irreducible polynomial whose dimension is or degree is m ok. So, then we know that a to the power of $2^m - 1$ should be congruent to 1, which means that if I want to compute a to the power of minus 1, then I need to compute a to the power of $2^m - 2$ ok, because essentially that should be giving me the multiplicative inverse ok. So, therefore right how will I compute this a to the power of $2^m - 2$ is what we need to discuss. If I want to compute a to the power of minus 1 or the multiplet or the multiplicative inverse of a.

(Refer Slide Time: 06:33)

Addition Chain

- Naïve method would require $(m-1)$ squarings and $(m-2)$ field multiplications.
- Since, multiplication is costly we don't apply this method.
- Itoh-Tsujii algorithm reduces the number of field operation by using an addition chain for $(m-1)$.
- Addition Chain for a positive integer n is a sequence of natural numbers $U=(u_0, u_1, \dots, u_l)$, st. the following are satisfied:

$$u_0 = 1$$
$$u_l = n$$
$$u_i = u_j + u_k, \text{ for } i = j+k, \text{ and } u_j, u_k \in U$$

Example: addition chain for 162, $U=(1,2,4,5,10,20,40,80,81,162)$

So, here there is a technique, which is called which is basically based on what is called as an addition chain ok. So, if I want to knifely compute a to the power 2 to the power of m minus 2, then you can see that I will require m minus 1 squarings and m minus 2 field multiplications ok.

Now, multiplication is costly, so we do not apply this knife method. Instead, we essentially can apply an algorithm, which is called as the Itoh-Tsujii inversion algorithm, which uses the addition chain ok. So, let us define an addition chain how are what is the definition of addition chain? So, addition chain for a positive integer n is a sequence of n sequence of natural numbers starting from u_0 , u_1 , and so on to say u_l that means, I will say that l is the length of the addition chain such that I will initialize u_0 to 1 and u_l to n that means, since is the addition chain for n the starting point is 1 and the end point is n ok. The in between values, we need to determine ok.

But, there is a restriction the restriction is that any sequence in this addition chain should be obtained by the sum of previous two numbers that means, for example if that is u_l or say u_i , then u_i should be essentially expressible as a sum of u_j plus u_k that means, should be expressed as a sum of the previous two numbers ok and u such that u_j and u_k both belongs to this addition chain ok. So, therefore I should be able to express you know like u_i as the sum of u_j plus u_k ok.

So, for example right if I want to compute say you know like for example, you can see in this addition chain that this is an addition chain for 162, so I have started with 1 my endpoint is 162, you can observe that in this addition chain 2 can be obtained by adding 1 with itself ok. Likewise 4 can be obtained by adding 2 with itself. Likewise, 5 can you obtain by adding 4 with 1 ok, 10 can be obtained with 5 with 5, 20 with 10 with 10, 40 with 20 with 20, 80 with 40 with 40, 81 with 80 and 1 and 162 where we add 81 with 81 ok.

So, these kind of addition chain where one of the numbers is the previous number is also often called as a brewer chain ok. So, this is a specific (Refer Time: 08:55) name given better which is called as a brewer chain, which is essentially where your previous number is just also a part of the addition chain ok. And what we want right therefore, if you if you want to compute this a to the power 2 to the power of n minus 2, then what we want is that this addition chain should be optimal in length ok. Although, it is a hard problem or this difficult problem to find out like what is the optimal chain, but there are some propositions in literature, which essentially gives us efficient chains ok, which we can use to compute this multiplicative inverse.

(Refer Slide Time: 09:33)

Itoh-Tsujii Algorithm

- For $a \in GF(2^m)$, let $\beta_k(a) = a^{2^k-1}, k \in N$.
- Thus, $a^{-1} = a^{2^m-2} = (a^{2^{m-1}-1})^2 = (\beta_{m-1}(a))^2$
- Interestingly, β_{m-1} can be efficiently computed using recursions:

$$\begin{aligned} \beta_{k+j} &= a^{2^{k+j}-1} = a^{2^{k+j}-2^k+2^k-1} \\ &= a^{(2^j-1)2^k} a^{2^k-1} \\ &= (\beta_j)^{2^k} \beta_k \end{aligned}$$

So, taking this background or taking this addition chain into account, what we can do is this so if I want to now compute a to the power 2 to the power of m minus 2 which was my inverse, then what I can do is I can define a variable, which I denote as beta k ok. So,

beta k is nothing but a to the power 2 to the power of k minus 1 ok, so beta k is defined as a to the power of 2 to the power of k minus 1.

So, if I want to calculate a to the power of minus 1, then therefore I as I said from Fermat's little theorem, I need to calculate a to the power 2 to the power of m minus 2, which is nothing but the square of a to the power 2 to the power of m minus 1 minus 1 whole square ok. So, what is this inside the parentheses is nothing but beta m minus 1 a ok.

So, therefore I need an efficient way to calculate beta m minus 1 and then finally I will square to get the inverse ok. So, this beta m minus 1 interestingly can be computed using a nice recursive formulation. So, what I can do is I can write beta k plus j, which is like some position in the addition chain and express them in terms of beta k and beta j ok. So, therefore what I do is I try to you know like derive beta k plus j, which is a higher beta value from the smaller beta values ok.

And therefore, I can do it efficiently. So, why does it work, so it is very easy to see like beta k plus j is nothing but e to the power 2 to the power of k plus j minus 1. So, therefore, what I can write is I can for convenience, I can introduce this minus 2 to the power of k, and then add plus 2 to the power of k, so basically it is the same minus 1.

And then from the first two factors, I take 2 to the power of k common ok. So, therefore I get 2 to the power j minus 1 into 2 to the power of k and the other term is a to the power 2 to the power of k minus 1. So, you can note easily that this that is a to the power 2 to the power of j minus 1 is nothing but beta j and I have done a beta j whole to the power 2 to the power of k, whereas the other term is beta k itself.

So, therefore I have been able to express beta k plus j in terms of smaller beta j and beta k values ok. Interestingly, you can also observe that I can decompose this in either you know like, I can also write this as in the other form like, I can also write this as beta k to the power 2 to the power of j into beta j right, so which one will I choose that may depend upon the fact that I probably would like to reduce the number of squarings, which is required here ok. Because, here I am doing beta j to the power 2 to the power of k. So, depending upon whether j is k small or k small, I can write the write the form accordingly ok. So, if k is small, then I will write in this form. If j is small, then I will

write this as β_k where k is the power of 2 to the power of j multiplied by β_j ok, which is also correct.

(Refer Slide Time: 12:27)

Example for $m=233$

- We need to find $a^{-1} = a^{2^m-2}$ for $m=233$
- We first define an addition chain for $m-1$
 - $(1, 2, 3, 6, 7, 14, 28, 29, 58, 116, 232)$
- Then, define $\beta_k = a^{2^k-1}$ then $a^{-1} = (a^{2^{232}-1})^2 = (\beta_{232})^2$
- Also define the recursion $\beta_{k+j} = (\beta_k)^2 \beta_j$

F.R. Henricquez et. al., A Fast Implementation of Multiplicative Inversion Over $GF(2^m)$, ITCC '05

So, therefore for example let us take an example, suppose I want to calculate for m equal to 233 last day we saw about a Karatsuba representation for 233. So, we will take that same multiplier applied here, so what we want to do is therefore, I want to compute a to the power of minus 1, which is equal to a to the power 2 to the power of m minus 2, where m equal to 233.


So, what do I need therefore, I need an addition chain right I need an addition chain for 232 ok, which is m minus 1. So, therefore here it is an addition chain for 232. So, you can observe that it satisfies all the properties of being an addition chain. And therefore, right what I can do is I can apply my recursive formulation and from there I can essentially calculate the value of β_{232} ok. If I know β_{232} , then I will just do a squarings to get the corresponding multiplicative inverse of a ok. So, let us try to see step by step how we can calculate these beta values to arrive at β_{232} and then finally get the inverse ok.

(Refer Slide Time: 13:27)

Computing the Inverse of 'a'

	$\beta_n(a)$	$\beta_{n+u_n}(a)$	Exponentiation
1	$\beta_1(a)$		a
2	$\beta_2(a)$	$\beta_{1+1}(a)$	$(\beta_1)^{2^1} \beta_1 = a^{2^2-1}$
3	$\beta_3(a)$	$\beta_{2+1}(a)$	$(\beta_2)^{2^1} \beta_1 = a^{2^3-1}$
4	$\beta_6(a)$	$\beta_{3+3}(a)$	$(\beta_3)^{2^2} \beta_1 = a^{2^6-1}$
5	$\beta_7(a)$	$\beta_{6+1}(a)$	$(\beta_6)^{2^1} \beta_1 = a^{2^7-1}$
6	$\beta_{14}(a)$	$\beta_{7+7}(a)$	$(\beta_7)^{2^2} \beta_1 = a^{2^{14}-1}$
7	$\beta_{28}(a)$	$\beta_{14+14}(a)$	$(\beta_{14})^{2^{14}} \beta_1 = a^{2^{28}-1}$
8	$\beta_{29}(a)$	$\beta_{28+1}(a)$	$(\beta_{28})^{2^1} \beta_1 = a^{2^{29}-1}$
9	$\beta_{58}(a)$	$\beta_{29+29}(a)$	$(\beta_{29})^{2^{29}} \beta_1 = a^{2^{58}-1}$
10	$\beta_{116}(a)$	$\beta_{58+58}(a)$	$(\beta_{58})^{2^{58}} \beta_1 = a^{2^{116}-1}$
11	$\beta_{232}(a)$	$\beta_{116+116}(a)$	$(\beta_{116})^{2^{116}} \beta_1 = a^{2^{232}-1}$

• In all we need 232 squarings and 10 multiplications.



So, so what we do here is here is an corresponding tabular representation of how we can do that, so we start with beta 1 a which is a. So, we start with a and then we calculate beta 2 a ok. So, you can see that this like the if you observe the suffix here, so 1, 2, 3, 6, 7, 14 28, 29, 58, 116, 232, this essentially stands for your addition chain ok. So, these are the values, which you had in the addition chain.

So, now what you can do is that you can calculate beta 2 in terms of beta 1, because you have you can express 2 as 1 plus 1 like what so ever therefore, if you can if you want to calculate this, therefore you can apply the recursive formulation. So, therefore beta 1 plus 1 is nothing but beta 1 to the power of 2 to the power of 1 into beta 1 and therefore you can calculate this corresponding value. Likewise, you can take any beta for example, let us take the example of say beta 29 for example if I want to calculate beta 29, then I can express 29 being in the addition chain as term 20 plus 1 ok.

And therefore, what I will do is I will write beta 28 whole to the power 2 to the power 1 into beta 1. Note I could have also done that as beta 1 to the power 2 the other way round, but that would have incurred more number of squarings ok. So, therefore these are very natural thing that I will do to reduce the number of squarings. So, all in all here we need 232 squaring, so maybe you can count it down and you need 10 multiplications.

So, you can count for example, 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10, so you need 10 multiplications and if you add up the number of squarings, which you are doing here. So,

like for example, here you are doing is squaring, here you are doing squaring, here you are doing 3 squarings and so on, you will get 232 squarings which you are essentially doing.

Now, again remember that squarings in GF 2 to the power of m arithmetic is easy ok. So, therefore this is a nice trade-off in that sense, where you are basically optimizing the number of multiplications at the expense of squarings, which is actually quite conveniently implement in GF 2 arithmetic.

(Refer Slide Time: 15:33)

Circuit to Raise Input to Power of 2^k

- Number of squarings at a stage can be as high as $u/2$
 - A naïve way would require $u/2$ squarings at each step.
- Circuit cascades u squarers, where u_i is an element in the addition chain.
- If number of squarings required is less than u_i , a multiplexer is used to tap out interim outputs.
 - In this case, the output is obtained in one clock cycle.
- If the number of squarings is more than u_i , the output of the squaring block is fed back to get squares which are multiples of u_i . This would require $\lceil u_i / u_i \rceil$ clock cycles.

So, how will I realize this by an hardware circuit, so here is a proposal. So, of course if I want to do a knife implementation right, then I would have just have some squarer's, and then I would have repeatedly applied those squarings functions right. But, then that may not be efficient, because you can observe that in the addition chain as you go up higher, there are some places, where you are doing significantly in a large number of squarings like here you are doing say 116 squarings ok.

So, in the general case it may happen that you are doing, so it since we are doing a large number of squarings that would although the squarings circuit is efficiently implemented. But, still if you are cascading, so many squarings operations, then at the end of the day your performance will get affected ok. So, therefore what you can do is you can try to trade off between hardware and the clock cycles and rather than implementing 1 squarer,

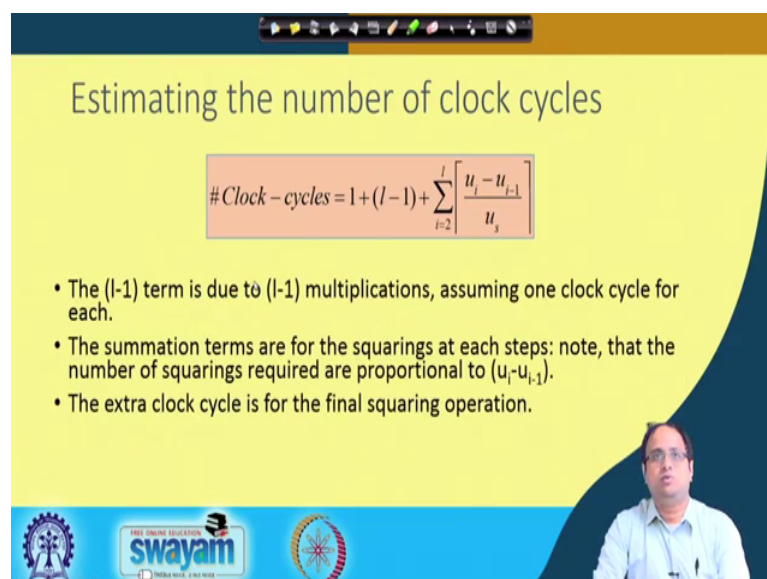
you can actually have a cascade of squarer's. So, you can have say U s a number of squarer's ok.

So, U s essentially is a value in the addition chain, which you have chosen by some kind of analysis I will try to elaborate later on, but imagine that you have got U s number of squarer's, which you have cascaded. And then you have followed it by a multiplexer circuit ok. So, now suppose I want you like to calculate a squarings I want to calculates sudden number of squarings, and if the number of squarings is less than U s, then what I can do is that I can just correspondingly multiplex out the output right.

For example, if I have got say 2 squarings required, then I will do I will pass the input here and what I will do is I will give an appropriate control here. So, that this signal, which is essentially after 2 squarings operations guess gets tapped out ok. On the other hand right if the number of squarings is more than U s, then you have to essentially feed this back result back ok.

And you have to do say U i by U s seal number of cycles ok. So, you have to basically keep on applying this 1 after the other and you have to get the corresponding result ok. So, depending upon you know like how many number of squarings you have, you essentially have to do this configurability of or you have to configure your hard work ok, you have to either give the output after 1 clock cycle or you have to spend more clock cycles for getting the result ok.

(Refer Slide Time: 17:49)



Estimating the number of clock cycles

$$\# \text{Clock-cycles} = 1 + (l-1) + \sum_{i=2}^l \left\lceil \frac{u_i - u_{i-1}}{u_s} \right\rceil$$

- The $(l-1)$ term is due to $(l-1)$ multiplications, assuming one clock cycle for each.
- The summation terms are for the squarings at each steps: note, that the number of squarings required are proportional to $(u_i - u_{i-1})$.
- The extra clock cycle is for the final squaring operation.

swayam

So, now what we yeah if we can estimate the number of clock cycles required here, so this expression shows the number of clock cycles. So, you can see that there are three components here. The $l - 1$ term is due to $l - 1$ multiplications. As I said that there will be $l - 1$ multiplications, which you are doing. And if you employ a combinational circuit like the Karatsuba multiply that we saw in the last class for every multiplication you are basically expending one clock cycle. So, you are doing $l - 1$ clock cycles.

You need one extra clock cycle, because if the extra clock cycle which you need for the final squarings operation, because you have to do a final squarings at the end. And this summation gives you the number of squarings for doing all the independent all the iterations which were there in the previous table. So, you can observe that the number of squaring, which you were doing right is $u_i - u_{i-1} + 1$. So, u_i is 1 addition chain $u_{i-1} + 1$ is the next addition chain. So, you are doing $u_i - u_{i-1} - 1$ number of squarings ok.

So, if you go back to this table for example, you can see that the number of squarings which you are essentially doing essentially is nothing but the summation of 1 is a subtraction or the difference from one u_i value with the previous one. Like; suppose it is 28, it is twenty 28 and 14, so you are doing 14 squarings operations here. So, likewise if these 3 and 2 you are doing 1 squarings operations here; if it is 232 and 116, here you are doing 116 squarings operation so, the difference ok.

So, therefore, right here you are, so therefore, the number of squarings that you need to do is $u_i - u_{i-1} - 1$. And since often you know like it may be more than u_s . So, as I said that you have to divide it by u_s , and then take the you know like the seal of that, and therefore, that gives you the number of clock cycles per iterations. And you have got l number of iterations, so you know like vary i^2 to 1, so that gives you a rough total number of clock cycles which you expect.

(Refer Slide Time: 19:47)

Quad Circuits vs Squarers

Comparison for LUTs required for Squarer and Quad for GF(2⁹)

Output bit	Squarer Circuit		Quad Circuit	
	$b(x)^2$	#LUTs	$b(x)^4$	#LUTs
0	b_0	0	b_0	0
1	b_5	0	b_7	0
2	$b_1 + b_5$	1	$b_5 + b_7$	1
3	b_6	0	$b_3 + b_7$	1
4	$b_2 + b_6$	1	$b_1 + b_3 + b_5 + b_7$	1
5	b_7	0	b_8	0
6	$b_3 + b_8$	1	$b_6 + b_8$	1
7	b_8	0	$b_4 + b_8$	1
8	$b_4 + b_8$	1	$b_2 + b_4 + b_6 + b_8$	1
Total LUTs		4		6

So, therefore, right once you have this right I mean it is fine, but at the same time you would like to optimize it ok. So, here is a possible optimization that you can try to sort of implement. So, as you see that in the normal Itoh-Tsujii the inversion algorithm there are a lot of squarings operation ok. So, we are so, you can actually try to do an interesting comparison that is little you can what you can try is you can try to compare a squarer circuit with a quad circuit ok. So, what is the quad circuit a quad circuit is nothing but which something which computes a to the power of 4 ok and squarer is something which computes a to the power of 2.

So, here is an example which we try to work out for GF 2 to the power of 9 ok. So, you can take an irreducible polynomial in x to the power of 9 plus and so on. And then you can calculate or create this field GF 2 to the power of 9 and compute the complexities or you know like the number of operations which you need to do, squarings and quads in these 2 for this field.

So, here is you know like the corresponding bits like 0 to 8, because there are 9 bits which are there in GF to the power of 8. And what we observe here is the circuits or the circuit complexity for calculating the squares ok. This one is for the quad operations. So, you can observe easily that quite intuitively the number of operations in the quad is more than the number of operations in square, because in square you are just doing power of 2, whereas in quad you are doing power of 4.

But, interestingly if you look now in terms of LUTs, you will see that and if you remember that in the last class some of the classes we discussed that in a lookup table, you essentially can fitting. Like suppose if I take a look up table with 4 inputs you can actually fit in 4 a Boolean function with 4 input variables. If we have got a Boolean function with 1 input variable or 2 input variable, then also that LUT gets consumed.

So, therefore, right if I want to calculate the number of lookup tables, you see that here there is not no function done, so there is no lookup table requirement. But, if you have got b_1 plus b_5 , then you have got 1 lookup table requirement ok. Likewise b_6 there is no lookup table requirement. If for $4 b_2$ plus b_6 we have got 1 lookup table requirement; like this we have got 4 look up tables which are being used.

On the other hand for the quad circuit, you will see that you still have got one lookup tables, but even for this value like b_1 plus b_3 plus b_5 plus b_7 , where you are doing more computation you are still using one lookup table ok. So, here at the end you are using 6 lookup tables. So, now, if we compare a quad circuit with a cascade of 2 squarers, where you are doing squarings and then squaring, then you can see that there is an improvement.


Because, here you are you are acquiring 6 look up tables, but here you would have required 4 into 2 that is 8 look up tables. But, in terms of delay, if you compare these two circuits both of them have got a delay of one lookup table ok, because they are consuming one look up tables ok. So, therefore, there seems to be an opportunity for trade off where you can try to optimize and have a better performing inversion circuit ok.

(Refer Slide Time: 22:47)

Using Quads Instead of Squarers

Field	Squarer Circuit		Quad Circuit		Size ratio $\frac{\#LUT_q}{2(\#LUT_s)}$
	#LUT _s	Delay (ns)	#LUT _q	Delay (ns)	
$GF(2^{233})$	153	1.48	230	1.48	0.75

- On an FPGA, Quads have better LUT utilization compared to squarers.
- Delay of a quad and a squarer is the same
- Therefore one can use a **Quad Itoh-Tsuji** algorithm which uses quad circuits instead of squarers.



So, what we observed is we try to kind of see it for other fields like larger fields. For example, if you see for GF 233 you will see that a squarer circuit will consume some 153 look up tables, whereas a quad will consume 233 look up tables. But, if you compare with again with 2 times 153, then you see that there is 25 percentage advantage; delay wise again both of them are same. So, therefore, naturally it tells us that you know like going from our squarer circuit to a quad circuit may give me an advantage, because I can utilize the look-up tables better. But the question is you also need to generalize like the Itoh-Tsuji algorithm to work for quad circuits.


(Refer Slide Time: 23:25)

Generalisation of Itoh-Tsuji

If, $a \in GF(2^m)$, $\alpha_{k_1}(a) = a^{2^{nk_1-1}}$, $\alpha_{k_2}(a) = a^{2^{nk_2-1}}$,
 $\Rightarrow \alpha_{k_1+k_2}(a) = (\alpha_{k_1}(a))^{2^{nk_2}} \alpha_{k_2}(a)$, $k_1, k_2, n \in \mathbb{N}$

$$a^{-1} = \begin{cases} \alpha_{\frac{m-1}{n}}(a)^2, & \text{when } n \text{ divides } (m-1) \\ [(\alpha_q(a))^r \beta_r(a)]^2, & \text{when } n \text{ not does divide } (m-1) \end{cases}$$

where, $nq + r = m-1$, and $n, q, r \in \mathbb{N}$



So, therefore, what we do is we essentially try to look into trying or trying to generalize this theorem and it turns out that the first one is very straight forward ok. So, now here instead of doing or defining beta, we try to define alphas ok. So, α^{k-1} is nothing but α to the power 2 to the power of $n^{k-1} - 1$ ok. So, in the previous case, what did we have we had a to the power of 2 to the power of $k-1$ but now we have generalized this and put in n equal n ok. So, we have generalize it to any integer n . So, likewise if you want to make it work as a quad circuit right, then I will make n equal to 2 ok.

And that will work as a to the power of 4 to the power of k right. And essentially we do power of 4 then power of 2 . So, likewise if you take α^{k-2} which is a to the power 2 to the power of $n^{k-2} - 1$, then we can see that you can actually calculate α^{k-1} plus $k-2$ in terms of α^{k-1} and α^{k-2} ok. Now, this proof is very straight forward similar to what we have already seen in the context of beta. So, I am not going to that.

What about inverse? So, once you have done this right and you have computed this series, then you can actually calculate e to the power of minus 1 also in a very straightforward manner. So, what you can do is depending upon you know like whether n divides $m - 1$. So, n what is n ? N is the parameter through which you are generalizing this quad or you know like higher powers of 2 ok. So, this n is essentially here.

So, therefore, when n divides $m - 1$, because n may divided $m - 1$, it may not be divided $m - 1$. If m divides $n - 1$, then α^{m-1} by n is defined, because $m - 1$ by n is there an integer. So, then if you can calculate α^{m-1} by n , then if you just squared it you will get a inverse ok. But, if it is not defined, then you can write $m - 1$ as $nq + r$ where nq and r are all natural numbers or integers, then you can write this a inverse as α^q to the power 2 to the power of r into $\beta^r a$ and then square the entire operation ok.

(Refer Slide Time: 25:27)

Justification

- Case 1: When n divides (m-1):

$$[\alpha_{\frac{m-1}{n}}(a)]^2 = [a^{2 \cdot \frac{m-1}{n} - 1}]^2 = [a^{2^{m-1} - 1}]^2 = a^{-1}$$
- Case 2: When n does not divide (m-1):

$$[[\alpha_q(a)]^{2^r} \beta_r(a)]^2 = [(a^{2^{nq-1}})^{2^r} (a^{2^r-1})]^2 = [a^{2^{nqr} - 1}]^2 = [a^{2^{m-1} - 1}]^2 = a^{-1}$$

So, you can try to observe this very see a very, very clearly here that is suppose you have got alpha m minus 1 by n. So, then alpha m minus 1 by n is nothing but you can write this as a to the power 2 to the power of m minus 1 by n multiplied by n. So, therefore, this n and n cancel. So, you have got alpha 2 to the power of 2 to the power of n minus 1 minus 1, which essentially is nothing but the inverse from for Fermat's little theorem ok. When n does not divide m minus 1, then you cannot write this in this form, because m minus 1 by n is not an integer.

So, there you see that alpha q that I mean is not there the right hand side of the previous equation which is alpha q a whole to the power 2 to the power of r into beta r is nothing but I can elaborate this alpha q as a to the power 2 to the power of n q minus 1 and then raise it to the power 2 to the power of r multiplied it with a to the power 2 to the power of r minus 1. Note that it is beta actually not alpha so they and then square it. So, therefore, what is this is nothing but alpha or rather a to the power 2 to the power of n q plus r minus 1 and this is squared ok.

So, therefore, here this 2 to the power of r cancels with this minus 2 to the power of r. And therefore, you have got 2 to the power of n q plus r and n q and n q plus r was m minus 1. So, therefore, I can write this as a to the power 2 to the power of m minus 1 minus 1 and which I which a very square I again get a inverse ok. So, therefore, in both cases a inverse is correctly computed.

(Refer Slide Time: 26:55)

Quad Itoh Tsujii Algorithm

	α_k	$\alpha_{ij+\alpha_k}$	Exponentiation
1	α_1		a^3
2	α_2	α_{1+1}	$(\alpha_1)^4 \alpha_1 = a^{4^2-1}$
3	α_3	α_{2+1}	$(\alpha_2)^4 \alpha_1 = a^{4^3-1}$
4	α_6	α_{3+3}	$(\alpha_3)^4 \alpha_3 = a^{4^6-1}$
5	α_7	α_{6+1}	$(\alpha_6)^4 \alpha_1 = a^{4^7-1}$
6	α_{14}	α_{7+7}	$(\alpha_7)^4 \alpha_7 = a^{4^{14}-1}$
7	α_{28}	α_{14+14}	$(\alpha_{14})^4 \alpha_{14} = a^{4^{28}-1}$
8	α_{29}	α_{28+1}	$(\alpha_{28})^4 \alpha_1 = a^{4^{29}-1}$
9	α_{58}	α_{29+29}	$(\alpha_{29})^4 \alpha_{29} = a^{4^{58}-1}$
10	α_{116}	α_{58+58}	$(\alpha_{58})^4 \alpha_{58} = a^{4^{116}-1}$


We set n=2

Initial Pre-computation:
To compute the cube, the multiplier is used for 2 clock cycles. Remember there is no squarer.

Likewise, the final squaring also is done by a multiplication step.

So, in total 12 multiplications.

• We now require 115 quads (instead of 232) and 9 multiplications. We save 7 clock cycles.



So, what we try to do is that we just try to see if I apply quad Itoh-Tsujii now ok. And then we will see that how or what is the amount of advantage or improvement that you get over the normal Itoh-Tsujii inversion algorithm. So, if you implement this, you will see that now, we will start alpha 1, we will initialize alpha 1 to say a cube that is a to the power of 3 and then I will try to do the addition chain, but right now I do not need to go to 232, but rather I can stop at 116. So, what I will basically now do is, I will try to calculate alpha 2 as alpha 1 plus 1 alpha 2 plus 1 exactly like what we have done previously, but now I have changed this to this power of force ok.

So, therefore, the computation is being done fast actually it is being done at a faster rate, but again as you observe that if you compare a quad with us 2 squarings, then the delay is same. So, therefore, right in the initial pre-computation here, there is an amount of cost that you have to pay, because in the previous case you were just initializing with a, but now you are initializing with a cube ok. And remember that you do not have a squarings circuit. So, therefore, if you want to implement a power of 3, you need 3 multiplication operations ok. So, you need to actually you need 2 clock cycles, because you have to do a into a into a so, you need 2 clock cycles.

And you also have to do a final squarings operation which you do with a multiplication, because you do not have a squarings ok. So, you have got 3 extra operations which you are doing ok. And likewise so and in total right you will see that there are 12

multiplications, because there are 9 multiplications here and if you also accommodate these extra 3 multiplications then you need to do 12 multiplications. Number of quads operations, you have to do 115 quad operation instead of 232 ok, which you did in the corner spawn when we have when we were operating with beta. And you also say save 7 clock cycles in the process.

(Refer Slide Time: 28:47)

Estimating the number of clock cycles

- #Multiplications: $l-2+3=l+1$. Note that 2 multiplications are needed initially for pre-computation, and one for final squaring.

$$\# \text{ Clock - cycles} = (l + 1) + \sum_{i=2}^{l-1} \left[\frac{u_i - u_{i-1}}{u_s} \right]$$

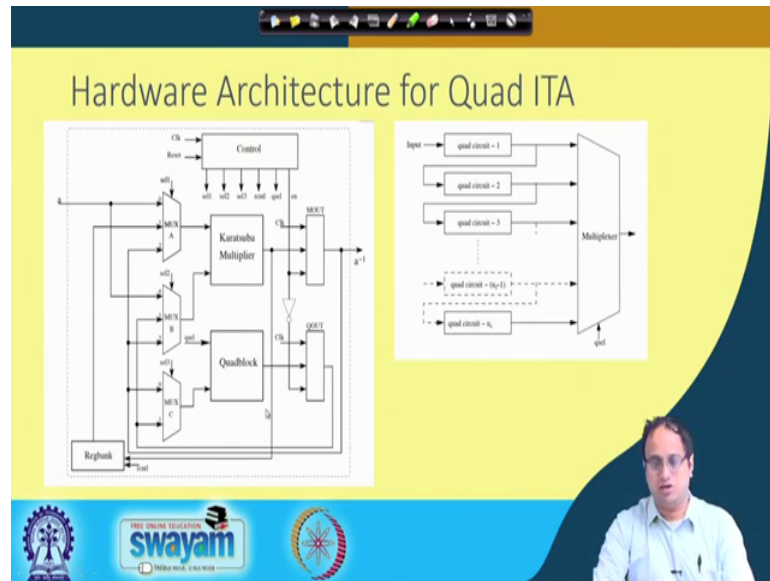
- Difference in clock cycle: $\left[\frac{u_i - u_{i-1}}{u_s} \right] - 1$

This can be as high as $(m-1)/2$ for $GF(2^m)$

So, therefore, right if you just write this entire stuff, then you will see that the number of clock cycles is as follows 1 plus 1 plus sigma now i goes from 2 to l minus 1, in the previous case I was running from 2 to l. So, therefore, the number of savings in terms of clock cycles is quite interesting it is $u_l - u_{l-1}$ divided by u_s minus 1 that is this extra step.

And there is one extra clock cycle here, so I subtract out minus 1 to accommodate that ok. So, therefore, right you see that this essentially can be as high as $(m-1)/2$ for $GF(2^m)$. So, therefore, if you have got larger dimensions, then actually this advantage could be even more.

(Refer Slide Time: 29:25)



So, with this I will you know like briefly stop here. And I will start discussing about the corresponding hardware architecture, when I am trying to realize this in the form of architecture in my next class.

Thank you.