

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

**Lecture – 22**  
**Hardware for Elliptic Curve Cryptography – IV**

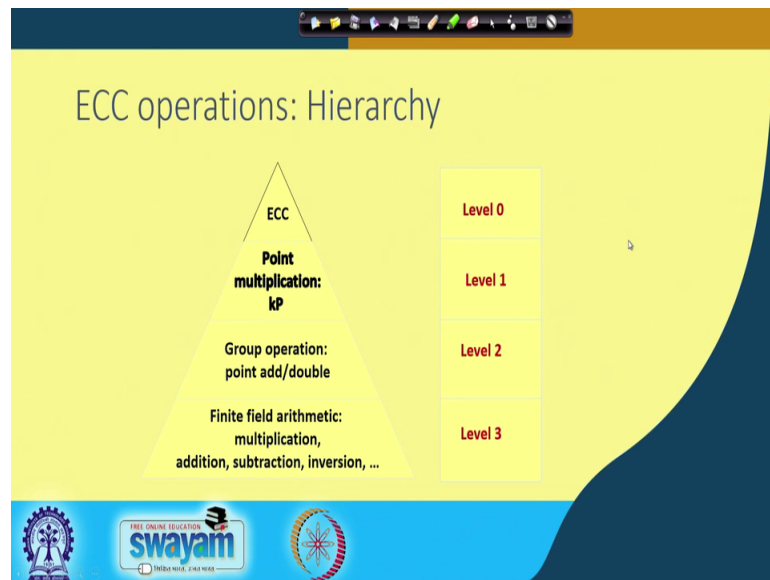
So, welcome to this lecture on Hardware Security. So, we shall be continuing our discussions on Elliptic Curve Cryptography. In particular today we shall be trying to understand about how to parallelize the Montgomery ladder that we were discussing. So, you will be discussing about some potential opportunities of parallelization. We shall be following with the detailed description about how to design an elliptic curve processor. So, we shall be looking into the overall architecture of an ECC processor.

(Refer Slide Time: 00:41)



We shall be trying to get into the various components of the architecture like the register bank, the ALU, how the control unit would be designed and finally, the inversion, when we convert back from the projective coordinates back to the affine coordinates. So, in particular I will be not talking about a design based on the Montgomery trick that we have seen, but rather this implementation will be on the plane you know like double and add algorithm that is the initial naive double and add algorithm. But, using this idea you can easily extend it to more to other variants of the elliptic curve processor.

(Refer Slide Time: 01:19)



So, so let us look start in that case. So, this is the ECC hierarchy that we were discussing. So, as we discussed there are different levels of the elliptic curve operations. So, the overall objective or overall what we try to do is, we basically try to have efficient implementations of point multiplication or scalar multiplication. And therefore, right if you can I mean if you when you are going in to hardware design right we always look for opportunities of parallelism.

So, that is the inherent or one of the important you know like selling points of hardware design over software. So therefore, right I mean we would like to of course, parallelize, but whether we can parallelize is always constrained by the amount of resource that we have at hand ok.

(Refer Slide Time: 02:03)

**Parallel Strategies for Scalar Point Multiplication**

- **Point Doubling**
  - Cycle 1:  $T=X_1^2$ ,  $M=cZ_1^2$ ,  $Z_2=T.Z_1^2$
  - Cycle 1a:  $X_2=T^2+M^2$
- **Point Addition**
  - Cycle 1:  $t_1=(X_1,Z_1)$ ;  $t_2=(Z_1,X_2)$
  - Cycle 1a:  $M=(t_1+t_2)$ ,  $Z_1=M^2$
  - Cycle 2:  $N=t_1.t_2$ ,  $M=xZ_1$
  - Cycle 2a:  $X_1=M+N$

**1 multiplier**

**2 multipliers**

We assume that squarings and multiplications with constants can be performed without multipliers...

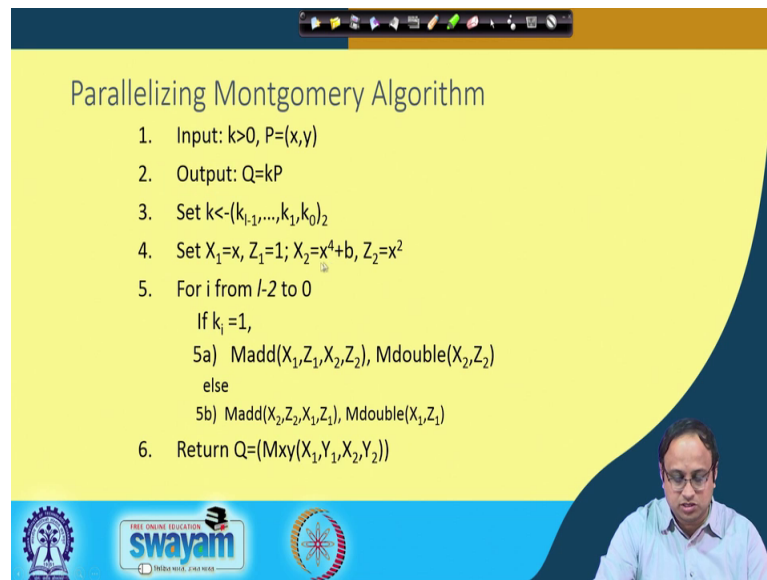
swayam

So therefore, right let us look into or have a quick look into the Montgomery structure where we essentially had a point doubling and a point addition. So, these are the point doubling and the point addition equations actually ok. So, now if you observe that I have tried to give potential parallel strategies for scalar point multiplication depending upon the fact or assumption that suppose you have got 1 multiplier or 2 multiplier resource at hand ok. Because, multiplier the typically quite consuming in terms of resources.

So therefore, right if we say for example, that our architecture can support 1 multiplier then we easily see that the point doubling operation essentially will require you know like you can essentially perform them in this in the cycle. For example, you can essentially implement them using you know like; so, these are the equations. So therefore, this is the multiplier for example, where you are computing  $Z^2$  which is equal to  $T$  into  $Z^1$  square. So, there you are basically expending this multiplier ok.

Likewise right if you see the point addition equation in the point addition you see that I have got 2 multiplications, like for  $t_1$  I have got one multiplication and for  $t_2$ , I have got 1 multiplication ok. Likewise for cycle 1 a there is no multiplication ok, but for cycle 2 I have got again 1 multiplication required. So therefore, right I mean the whole idea is that depending upon the, we again assume that the squaring and the multiplications with constants can be performed without any multiplications or without any explicit multipliers ok.

(Refer Slide Time: 03:47)



Parallelizing Montgomery Algorithm

1. Input:  $k > 0, P = (x, y)$
2. Output:  $Q = kP$
3. Set  $k \leftarrow (k_{l-1}, \dots, k_1, k_0)_2$
4. Set  $X_1 = x, Z_1 = 1; X_2 = x^2 + b, Z_2 = x^2$
5. For  $i$  from  $l-2$  to  $0$ 
  - If  $k_i = 1,$ 
    - 5a)  $\text{Madd}(X_1, Z_1, X_2, Z_2), \text{Mdouble}(X_2, Z_2)$
    - else
    - 5b)  $\text{Madd}(X_2, Z_2, X_1, Z_1), \text{Mdouble}(X_1, Z_1)$
6. Return  $Q = (\text{Mxy}(X_1, Y_1, X_2, Y_2))$

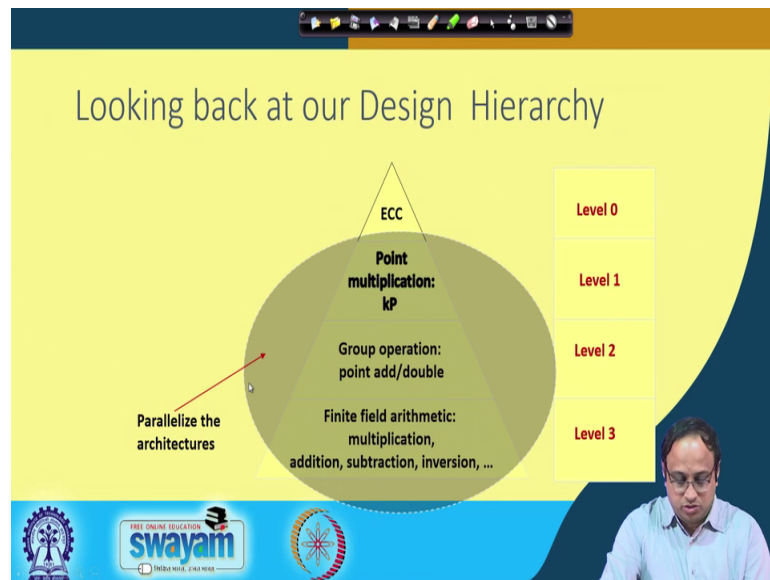
The slide also features logos for Swamyam and other educational institutions at the bottom, and a small video inset of a man speaking in the bottom right corner.

So therefore, right I mean with this with these equations in mind and with this kind of computations required for doubling and addition let us try to see that how we can parallelize the Montgomery algorithm. So, in Montgomery algorithm if you remember this was your scalar  $k$  and then you essentially initialized  $X_1$  and  $Z_1$  because, you do not operate on  $y$  coordinates here. So, you have got and you are doing on projective coordinate. So, you need the  $X$  and  $Z$  coordinates and for  $X_2$  it is a doubled result.

So therefore, these are the corresponding doubled equations. So, if  $k_i$  is equal to 1 you basically run an iterative algorithm from  $l$  minus 2 known to 0. So, this is the msb first algorithm, you basically add in  $X_1 Z_1$  and  $X_2 Z_2$  if the  $k_i$  is 1. And, you double  $X_2 Z_2$  and if it if  $k_i$  is equal to 0 then you do the opposite; that means, you add  $X_2 Z_2$  with  $X_1 Z_1$ , but you double  $X_1 Z_1$  ok.

And finally, the result is  $X_1 Y_1 X_2 Y_2$ , but from using that you finally, you know want to return the result back to affine coordinate. So, you make a transformation from the projective coordinates back to the affine coordinates. So therefore, right what we have written as a part of step 5, there are 5 steps are 5 a and 5 b. So, either that you are doing 5 a or you are doing 5 b operations ok.

(Refer Slide Time: 04:57)



So therefore, now what we are trying to do is we are trying to parallelize this point multiplication operation. And, let's see how we can do that.

(Refer Slide Time: 05:07)

Parallelizing Strategies

- **Parallelize level 1:** If we allocate one multiplier to each of Madd and Mdouble, then we can parallelize steps 5a and 5b. Thus 4 clock cycles are required for each iteration. **Total time is nearly 4I**
- **Parallelize level 2:** If we can parallelize the underlying Madd and Mdouble, then we cannot parallelize level 1, if we have a constraint of 2 multipliers. So, we have a sequential step 5a and 5b. **Total time is 3I.**

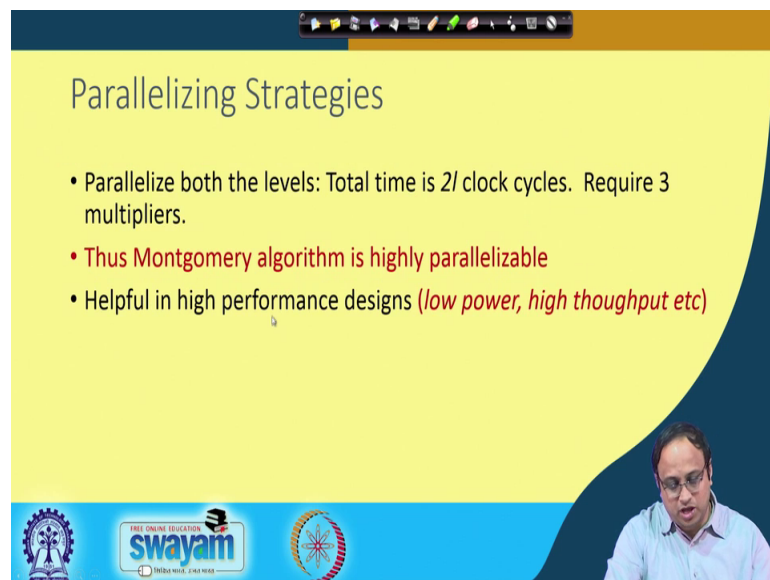
swayam

So, therefore, right so, there are these are the 2 corresponding strategies that we mentioned. Suppose if we allocate one multiplier to each of the Madd and Mdouble. That means, suppose you have got 2 multipliers and you allocate one multiplier each to Madd and Mdouble ok, then you can parallelize steps 5a and 5b. Because, now you can do this addition operation and double operation in parallel because, you

have attributed one multiplier to the, to addition and one addition one multiplication one multiplier to the doubling operation ok.

So therefore, right if you if you do that then you see that 4 clock cycles are required for each iteration ok. So, so therefore, right I mean so, so that means, the total time which is required is nearly 4 I I mean this nearly 4 I. So therefore, right I mean the, this is about the time that your algorithm we will take ok. On the other hand if you can parallelize the underlying Madd and Mdouble then we cannot parallelize level 1 ok. So, thus we if we have a constraint of 2 multipliers ok; so, we have a sequential step 5 a and 5 b. So now, the total time will be 3 I actually ok.

(Refer Slide Time: 06:31)



Parallelizing Strategies

- Parallelize both the levels: Total time is 2/ clock cycles. Require 3 multipliers.
- Thus Montgomery algorithm is highly parallelizable
- Helpful in high performance designs (*low power, high throughput etc*)

So so, therefore, right I mean you see that depending upon what parallel strategies you have you essentially I mean you know like you have got 2 alternatives ok. So, but on the other hand if you can parallelize both the levels; that means, level 1 and level 2 then you can actually reduce this time to 2 I clock cycles ok. You can reduce it to 2 I clock cycles, but this will required 3 multipliers now ok. So, you see that the more resources you can expand, you can actually parallelize the Montgomery ladder quite significantly ok. And, that is why right we conclude the Montgomery algorithm is highly parallelizable ok.

And, it is helpful in high performance designs and also right for low power and high throughput both kinds of you know like applications. Because, on one hand right you essentially can have probably resource constant environments where, you essentially can

still perform Montgomery ladder reasonably well even with say 1 multiplier cost ok. On the other hand you can also go for quite high throughput designs actually ok. So therefore, all depends upon how much resource you have, but the point is right that the Montgomery algorithm has copes of parallelism. And therefore, you can try to capitalize on that ok.

(Refer Slide Time: 07:33)

Design of an ECC Processor:  
Parameters of the Design

- Characteristic 2 field:  $GF(2^{233})$
- Random Curve:  $y^2 + xy = x^3 + a.x^2 + b$ , where  $a = 1$
- /\* Basepoint for the curve, taken from FIPS 186-2 \*/
- Base-Point (X,Y):
  - $233'h0fac9dfcbac8313bb2139f1bb755fef65bc391f8b36f8f8eb7371fd558b$
  - $233'h1006a08a41903350678e58528bef8a0beff867a7ca36716f7e01f81052$
- /\* The constant b for the curve, from FIPS 186-2 again \*/
  - $233'h066647ede6c332c7f8c0923bb58213b333b20e9ce4281fe115f7d8f90ad$

[csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf](https://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf)

swayam

So, now with this background we are essentially you know like we let us look into a possible case study where, we would like to design ECC processor and these are the corresponding parameters of this design. So, this design is on GF 2 power of 233 which is essentially one of the fifth standards for elliptic curves in characteristic 2 fields. So, the curve is a random curve of this equation  $y^2 + xy = x^3 + ax^2 + b$  where  $a$  is equal to 1 ok. But, you can see that according to the fifth standard which you can also look into the nist corresponding website ok. You will see that the base points like the  $p$  for example, that we were talking about essentially are 233 bit constants ok.

So, here are just an example just to show you how it looks like. So, you see it is pretty you know like big number. So, you basically I have to do a arithmetic and computations in such kind of supporting such kind of large arguments. So, the constant  $b$  for the curve is again you know like given in fifth standard and this is essentially shown here ok.

These are just for the sake of complete test try to give us an idea about how the constants look like ok.

(Refer Slide Time: 08:45)

### Design Hierarchy

- Elliptic Curve Hierarchy

The slide features a yellow background with a blue header and footer. The footer contains logos for Swamyam and other educational institutions. A small video inset of the presenter is visible in the bottom right corner.

So, now we would like to essentially develop this elliptic curve design, again with this is our hierarchy we want to implement the scalar multiplication, but you have to start from the bottom. Because, you have to implement all of this multiplication and you know like the field operations and then the doubling and addition operation and finally, the scalar multiplication right. So, this is the elliptic curve hierarchy ok.

(Refer Slide Time: 09:07)

### Code Hierarchy

```
module ecsmul(clk, nrst, key, sx, sy, done);  
regbank regs(clk, cwh, c0r, c1r, a0, a1, a2, a3);  
ec_alu alu(cwl, a0, a1, a2, a3, c0a, c1a);  
multiplier mul(minA, minB, mout);  
module squarer(a, d);  
module bquadrblk(en, in, sel, out);
```

The slide features a yellow background with a blue header and footer. The footer contains logos for Swamyam and other educational institutions. A small video inset of the presenter is visible in the bottom right corner.



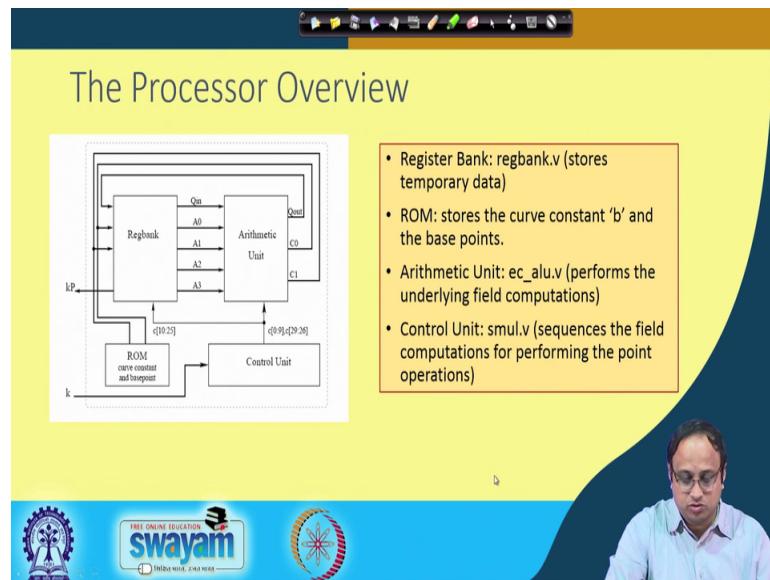
And if you look into the code hierarchy; so, you will be try to give you some indications about potentially how we can write for example, in RTL code. So, I will using Verilog, but you can easily generalized it to other RTL languages ok. So, so little bit of background on Verilog would be handy here and though not much in not much did not in much details.

So for example, like the for the field operation as you can see that the basic blocks are multiplier squared and a quad block ok. So, if you remember in one of our previous lectures light we have been discussing about how to realize these architectures where, we had looked into the Karatsuba multiplier. Where, we have seen the squarer architecture like how we can do squaring in characteristic 2 fields and also how you can do quading and what are the advantages of doing that in a fpj topologies ok.

So, now we would like to build upon a top architecture using these components and finally, right the computations will be done or accumulated are done in an arithmetic logic unit. So, this is the elliptic curve logic unit or alu and there is a corresponding register bank which essentially is kind of stores temporary data, also takes initial data like the constants are loaded into the register bank and all the computations are done kind of; so, it is like a scratch pad.

They where you are basically doing your computations and kind of storing storing the temporary result and finally, all of these are kind of combine into a scalar multiplication. This is your top level module or ecmul where you are essentially you know like taking the input that is the corresponding you know like key and based upon that you are doing your scalar multiplication. And finally, when the result is done then you essentially get a done signal to flag that the result or the computation is complete ok.

(Refer Slide Time: 10:55)



So, now let us look into the processor overview. So, here are the 4 important components of my design. So, I have got as I said that the register bank and the arithmetic unit which are kind of handshaking with each other. The register bank is essentially storing temporary data and the ROM essentially stores the curve constant and the base point. So, if this stores you know like the  $p \times p$   $y$  and also the base point that this is the base point and also the curve constant. So,  $A$  is equal to 1 in this curve, but  $B$  essentially has a non-trivial value which is stored in the ROM ok.

So, this data essentially is sometimes passed to the register bank or the register bank also takes data from the arithmetic unit; that means, the arithmetic units perform some computations. And, then store the result back to the register bank ok. Now, the register bank and the arithmetic unit also do handshaking with each other. So; that means, like when the arithmetic unit required some data, it essentially looks into the register bank that is a scratch pad and from there it gets temporary results to do further computations.

But, then the entire you know like ring leader sort of or you know the controller is the control unit with basically essentially performs or performs the rather sequences the field operation ok. Like the whatever operations will be performed the sequences in which they will be performed. So, that decision is done or made by the controller or the control unit ok.

So, roughly speaking as I said previously also it is very important to understand the data path and the control path of your design. So, this control unit is your control path whereas, this essentially comprises of your data path of the circuit ok. So, we essentially have return you know like individual Verilog codes for all of them. For example, the register bank is being written as the regbank dot v and the arithmetic unit as ec underscore alu dot v and the controller using another Verilog file ok. So, all of them have been built in a modular fashion.

(Refer Slide Time: 12:53)

**Register Bank**

Heart of the register file is 8 registers of size 233 bits.

- Organized as 3 banks: RA, RB, RC
- Dual port Distributed RAMs of the Xilinx FPGAs
- Asynchronous Read
- Synchronous Write
- we: write enable signal
- Inputs:  $C0$ ,  $C1$ ,  $Q_{out}$
- Outputs:  $A0$ ,  $A1$ ,  $A2$ ,  $A3$  and  $Q_m$

Register	Description
$RA_1$	1. During initialization it is loaded with $P_x$ . 2. Stores the $x$ coordinate of the result. 3. Also used for temporary storage.
$RA_2$	Stores $P_x$ .
$RB_1$	1. During initialization it is loaded with $P_y$ . 2. Stores the $y$ coordinate of the result. 3. Also used for temporary storage.
$RB_2$	Stores $P_y$ .
$RB_3$	Used for temporary storage.
$RB_4$	Stores the curve constant $b$ .
$RC_1$	1. During initialization it is set to 1. 2. Store $z$ coordinate of the projective result. 3. Also used for temporary storage.
$RC_2$	Used for temporary storage.

So, now let us look into the register bank. So, in the register bank which is in the heart of you know like the design. So, the essentially the so, there are basically like 8 registers each of size 233 bits. Because, my arguments are of size 233 bits, they are organized as 3 banks and there are implemented as dual port distributed RAMs which are present in the Xilinx FPGAs ok. But, pretty much you can find similar constructs in other architecture or other platforms as well ok.

So, the main characteristic of this memory bank is that the for example, if you look into this architecture and you see the  $RA_1$  which is essentially one of the banks, you see that there is a corresponding address ok. And, you essentially can perform and you know like asynchronous write I mean you can perform and asynchronous read and a synchronous write. That means, with the clock you can essentially set or enable you know like the

write signal here which is a  $\overline{WE}$  signal and you can actually you know like flag corresponding address and you can write into that address synchronously with the clock.

But you can also read. So, you can essentially read at any time; that means, a read is always enabled, but from where you want to read for that there is another address ok. So, you essentially have got an address 2. So, you will see that there is an address 1 typically which is for writing an address 2 which your using for reading ok. So, there are essentially the because you essentially; so, this is a dual port RAM. So, you can you know like right in to a different location and you can read from a different location. So, that essentially provision is provided here ok.

So, you can and you can observe that you can essentially read 2 data, but you can write into 1 1 address. So, likewise right if you see that in the register bank there are like 3 important registers RA 1 RA 2 and RC 1 or other RA I would say other RA RB and RC. The reason why we have got 3 register banks is because we are doing are computation in projective coordinates. So, we have got  $x$   $y$  and  $z$  ok. So, RA is essentially used for processing the  $x$  components whereas, this is for the  $y$  components and this is this is for the  $z$  components ok. So, what so if you read this table you will see that we kind of detail about the functionalities of each of these registers in a very high level in a very high level.

So for example, RA 1 is used during initialization it is loaded with  $P_x$  that is  $x$  coordinate of the base point. You it stores the  $x$  coordinate of the result ok that is the intermediate results when you are doing you are computation. It is also used for temporary storage ok. RA 2 is used for storing  $P_x$  ok. RB 1 is used for you know the during initialization it is loaded with  $P_y$  and it also stores the  $y$  coordinate of the result and is also used for temporary storage. RB 2 is used for storing  $P_y$ , RB 3 is also another registered which we use for temporary storage and RB 4 stores the curve constant which is small  $b$  ok.

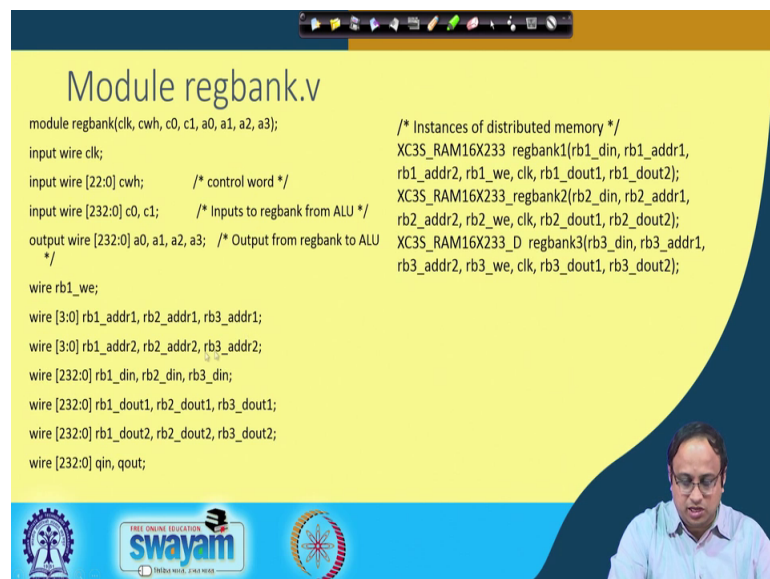
So, you take from the RAM and you store it in RB 4 for you know like faster access. And RC 1 essentially stores the  $z$  coordinates. So, you remember like initialize we initialize the  $z$  coordinate with 1. So, you basically initialize it and set it to 1 and you store the  $z$  coordinate of the projective result and you also store and it is also used for temporary

storage ok. So, RC 2 is again temporary register which we essentially use in our architecture ok.

So, another point which we will see later on is that this computation takes place in mixed coordinate; that mean you know like one of the coordinates is in projective domain and the other one is in affine domain. So, with this background we have more or less we have understood how or what are the functionalities of each of these registers. You see that there are some output ports and input ports for example, the output ports are A 0 A 1 A 2 A 3 and also Q in ok. So, these bosses these are bosses ok; that means, these are wide you know like n bit data which is being passed.

So, this data passes from the register bank and goes to the Arithmetic Logic Unit or the ALU and there are some inputs like C 0 C 1 and Q out. So, these inputs are the outputs of the arithmetic logic unit ok. So, they are essentially taken from the arithmetic logic unit and feedback into my register file ok.

(Refer Slide Time: 17:29)



```
Module regbank.v

module regbank(clk, cwh, c0, c1, a0, a1, a2, a3);
input wire clk;
input wire [22:0] cwh; /* control word */
input wire [232:0] c0, c1; /* Inputs to regbank from ALU */
output wire [232:0] a0, a1, a2, a3; /* Output from regbank to ALU */
/*
wire rb1_we;
wire [3:0] rb1_addr1, rb2_addr1, rb3_addr1;
wire [3:0] rb1_addr2, rb2_addr2, rb3_addr2;
wire [232:0] rb1_din, rb2_din, rb3_din;
wire [232:0] rb1_dout1, rb2_dout1, rb3_dout1;
wire [232:0] rb1_dout2, rb2_dout2, rb3_dout2;
wire [232:0] qin, qout;

/* Instances of distributed memory */
XC3S_RAM16X233 regbank1(rb1_din, rb1_addr1,
rb1_addr2, rb1_we, clk, rb1_dout1, rb1_dout2);
XC3S_RAM16X233 regbank2(rb2_din, rb2_addr1,
rb2_addr2, rb2_we, clk, rb2_dout1, rb2_dout2);
XC3S_RAM16X233_D regbank3(rb3_din, rb3_addr1,
rb3_addr2, rb3_we, clk, rb3_dout1, rb3_dout2);
*/

```

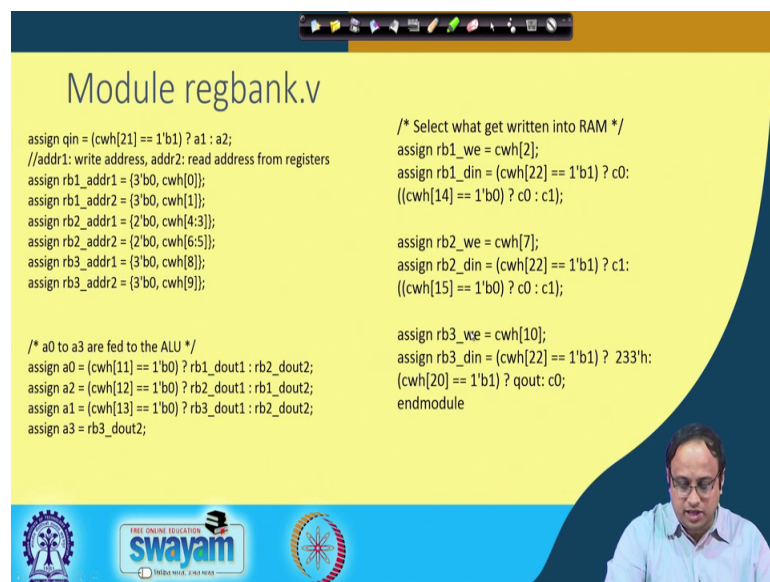
So, this is a potential Verilog design of the register bank and if you can see that this is the module of the register bank. In particular I would like to highlight few things for example; I would like to highlight the hierarchy here. So, you can see and easily understand this part that is there is clock of course, which because there is a synchronous event which is taking place as I said that the write is synchronous. There is a, you know like part of the control word which is used. So, the control word is essentially divided

into 2 parts: one is a lower part and the one is higher part. So, the higher part is integrator cwh ok. There there are inputs like c 0 and c 1 which are essentially inputs which are received and there are outputs like a 0, a 1, a 2 and a 3 ok.

Another important point which may be observed is that how we have instantiated the distributed memory. So, these are you know like the 3 components are register bank RA RB and RC. So therefore, we have got 3 instantiations of the register bank ok. Again this is a particular instantiation which was which was done on a vortex 4 kind of environment for a different platform platform we have to look into what exactly the module description should be ok. So, there are 3 register bank register bank 1 register bank 2 and register bank 3.

All of them have got din which is the data input which gets in. There is there are 2 addresses address 1 and address 2 as I mention address 1 is used for writing and address 2 is used for reading. There is a write enable because, the writing cannot be done always, it has to be synchronous with the clock. But, the reading can be done always and there are 2 read ports like dout 1 and dout 2 for each of these register banks ok.

(Refer Slide Time: 19:05)



```
Module regbank.v

assign qin = (cwh[21] == 1'b1) ? a1 : a2;
//addr1: write address, addr2: read address from registers
assign rb1_addr1 = {3'b0, cwh[0]};
assign rb1_addr2 = {3'b0, cwh[1]};
assign rb2_addr1 = {2'b0, cwh[4:3]};
assign rb2_addr2 = {2'b0, cwh[6:5]};
assign rb3_addr1 = {3'b0, cwh[8]};
assign rb3_addr2 = {3'b0, cwh[9]};

/* a0 to a3 are fed to the ALU */
assign a0 = (cwh[11] == 1'b0) ? rb1_dout1 : rb2_dout2;
assign a2 = (cwh[12] == 1'b0) ? rb2_dout1 : rb1_dout2;
assign a1 = (cwh[13] == 1'b0) ? rb3_dout1 : rb2_dout2;
assign a3 = rb3_dout2;

/* Select what get written into RAM */
assign rb1_we = cwh[2];
assign rb1_din = (cwh[22] == 1'b1) ? c0 :
((cwh[14] == 1'b0) ? c0 : c1);

assign rb2_we = cwh[7];
assign rb2_din = (cwh[22] == 1'b1) ? c1 :
((cwh[15] == 1'b0) ? c0 : c1);

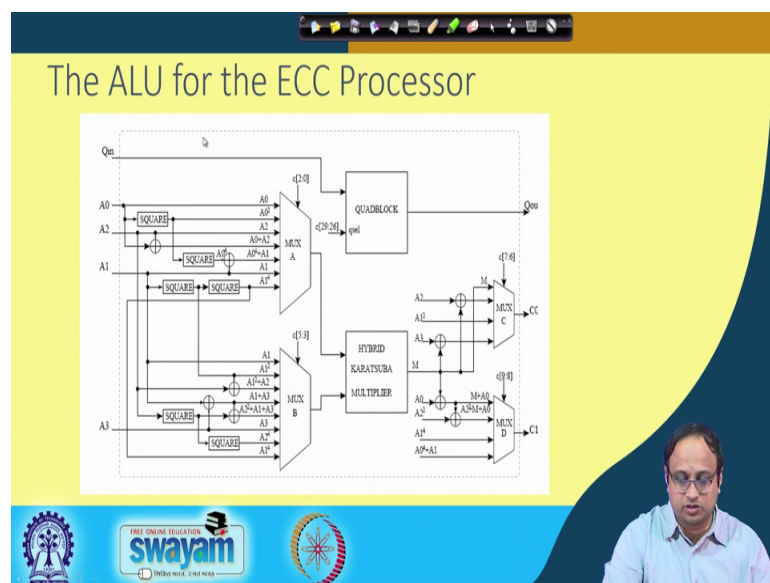
assign rb3_we = cwh[10];
assign rb3_din = (cwh[22] == 1'b1) ? 233'h :
(cwh[20] == 1'b1) ? qout : c0;
endmodule
```

So therefore, right I mean with this background right you can see that there are of course, more descriptions depending upon you know like I mean essentially this is nothing, but descriptions about based upon the architecture. So, if you see this architecture; so, based upon that these are you know like corresponding Verilog code which has been written to

describe the hardware essentially of the register bank ok. For example, like you can see that you know like how this a0 to a3 are fed to the ALU.

So, that you can see that there are essentially these are nothing, but multiplexers and the multiplexers has been written using an assign statement in Verilog ok. So, these are the descriptions of these multiplexers ok. So, the so, likewise you can essentially realize the entire register bank unit and the other important component is the arithmetic logic unit for the elliptic curve processor.

(Refer Slide Time: 19:53)



So, in the arithmetic logic unit there are two important blocks: one is the hybrid Karatsuba multiplier, but the inverse is not a monolithic block, but rather as we implemented using the quad block and the multiplier. So, if you remember like we are different stages of the ethos uzi inversion algorithm. So, we try to implement the ethos uzi inversion algorithm which uses the quad block as a fundamental block and along with the multiplier. The multiplication has been used at every step of the inversion routine ok.

Now, along with these along with these two things right there are some other peripherals which has been used which are essentially quit efficient in terms of hardware. For example, the squarer routines and also the XORs for example, which are kind of essentially done the XOR is use for doing additions of the arguments ok. Now, why these kind of units are like A0, A0 square and so on will be probably understood as we

progress ok. But, this is kind of just to give an idea about how the arithmetic logic unit looks like.

(Refer Slide Time: 20:53)

The slide, titled "The ALU for the ECC Processor", features a circuit diagram on the left and two text boxes on the right. The top right text box lists the ALU's functions: point operations (doubling and addition) in Lopez Dahab Projective co-ordinates efficiently, inversion from projective to affine co-ordinates, and its 5 inputs and 3 outputs. The bottom left text box details the computation phases: point addition and doubling, inversion, and the ALU's components: a Hybrid Karatsuba Multiplier, a Quad block, and a Quad block with 14 cascade steps.

**The ALU performs:**

- point operations (doubling and addition) in Lopez Dahab Projective co-ordinates efficiently.
- Inversion (from projective co-ordinates) to affine co-ordinates.
- The AU has 5 inputs (the outputs of the regbank) and 3 outputs (the inputs of the regbank).

**The computation of the AU has 2 phases:**

- Point addition and doubling
- Inversion

**The AU consists of:**

- Hybrid Karatsuba Multiplier (used at all time steps)
- Quad block (used in phase 2 for the final inversion)
- The Quadblock has 14 cascade steps (as described before) and computes the quading operation repeatedly (as per the value of  $c[29..26]$ )

So, let us now try to understand what the arithmetic logic unit performs ok. So, the arithmetic logic unit performs point operations like doubling and addition in Lopez Dahab projective coordinates in particular mix projective coordinates and there is a final inversion from the projective coordinates to the affine coordinates. So, there are 2 phases of the arithmetic unit ok. The arithmetic unit has got 5 inputs. So, these are denoted so, these are the outputs of the register bank and 3 outputs which are in turn register inputs of the register bank.

Now, the computation of the arithmetic unit has got 2 phases point addition and doubling and the final inversion. And as I mention that the arithmetic unit consist of the hybrid Karatsuba multiplier which is used at all time steps. Now, these an important part that we kind of try to adhere in our scheduling. Since, the multiplier is very kind of costly we do not want to keep the multiplier ideal at any clock cycle. So, basically every clock cycle there is one multiplication which is monetarily performed. This is just to save the number of clock cycles ok.

Likewise there is also a quad block which is used in phase 2 for the final inversion, when you want to covert back from the projective coordinates to the affine coordinates. And, the quad block if you remember like that design that we showed essentially had 14



cascaded steps. So, each of these steps were like power 4 circuits; that means, which [ basic/basically] takes an argument raise to the power of 4 ok. And, it computes the quading repeatedly depending upon a particular or controller or select line. And, a select line essentially has got bits from 26 27 28 20 29 so; that means, you know you have got virtually 4 bits to implement that.

So, so depending upon the, what value you set here what you compute is any input that give to this quad to this quad block it is raise to the power of 4 to the power of select actually. So that means, if we alpha is my input; input to the quad block and select is the value of this select line then, what I compute or get as the result is alpha to the power of 4 to the power of select ok. So therefore, you basically kind of raise it accordingly ok.

(Refer Slide Time: 23:05)

The verilog code for ALU

```

module ec_alu(cw, a0, a1, a2, a3, c0, c1);
    input wire [232:0] a0, a1, a2, a3; /* the
    inputs to the alu */
    input wire [9:0] cw; /* the control
    word */
    output wire [232:0] c0, c1; /* the alu
    outputs */

    /* Temporary results */
    wire [232:0] a0sq, a0qu;
    wire [232:0] a1sq, a1qu;
    wire [232:0] a2sq, a2qu;
    wire [232:0] sa2, sa4, sa5, sa7, sa8, sa8_1;
    wire [232:0] scl;
    wire [232:0] sd2, sd2_1;

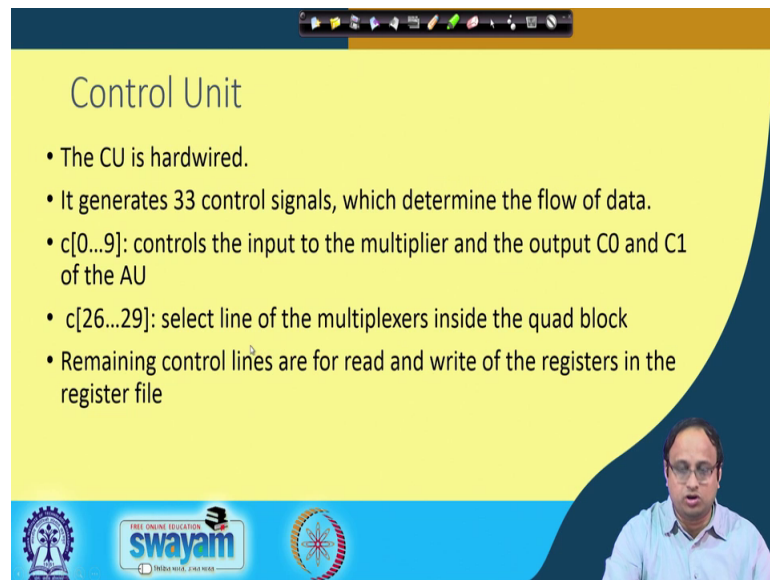
    /* Multiplier inputs and output */
    wire [232:0] minA, minB, mout;

    multiplier mul(minA, minB, mout);
    squarer sq1_p0(a0, a0sq);
    squarer sq_p1(a1, a1sq);
    squarer sq_p2(a2, a2sq);

    squarer sq2_p2(a2sq, a2qu);
    squarer sq2_p1(a1sq, a1qu);
    squarer sq2_p3(a0sq, a0qu);
    
```

So, this is again you know like a Verilog description for the arithmetic logic unit nothing must to see except that, you see that essentially the entire design has been done in a structural way. So, we basically kind of you know like realize the squarer multiplications and separate the (Refer Time: 23:18) separate Verilog codes and just instantiate in our design. So, likewise the entire description has been given for the arithmetic logic unit.

(Refer Slide Time: 23:31)



### Control Unit

- The CU is hardwired.
- It generates 33 control signals, which determine the flow of data.
- $c[0...9]$ : controls the input to the multiplier and the output  $C_0$  and  $C_1$  of the AU
- $c[26...29]$ : select line of the multiplexers inside the quad block
- Remaining control lines are for read and write of the registers in the register file

So, now you have got a controller and the controller is essentially a hardwired unit ok; that means it is essentially you know you can essentially change the control unit values as well. So, it generates 33 control signals and you can kind of reprogram the control you need to you know like for other kind of operations as well. So, it is kind of programmable and the so, the objective of this control signals is basically try to kind of determine the flow of data ok. For example, you know like  $c_0$  to  $c_9$ , this like one part of the controller. It with controls the input to the multiplier and the output  $C_0$  and  $C_1$  of the arithmetic unit ok.

So, essentially it compute it controls the input to your multiplier that is hybrid Karatsuba multiplier and the output  $C_0$  and  $C_1$  of the control unit. And, there as I said the  $c_{26}$  to  $c_{29}$  is the select line of the multiplexers which is used as an input to the quad block ok. The remaining control lines; that means, out of the 33 control lines the remaining control lines they are used for read and write of the registers in the register file ok. That means, there are used to control which registered you want to read, I mean write and from where you want to read ok.

(Refer Slide Time: 24:45)

**Projective Point Arithmetic**

- Point Doubling:
  - Input :  $(X_1, Y_1, Z_1)$
  - Output:  $(X_4, Y_4, Z_4)$
- Constraint: One multiplier
  - Can perform one multiplication per clock cycle
  - Hence needs four time steps

**Point Doubling in Projective Coordinates**

$$Z_4 = X_1^2 \cdot Z_1^2, X_4 = X_1^4 + b \cdot Z_1^4$$
$$Y_4 = b \cdot Z_1^4 \cdot Z_4 + X_4 \cdot (a \cdot Z_4 + Y_1^2 + b \cdot Z_1^2)$$

**Algorithm 6.1: Hardware Implementation of Doubling on ECCP**

**Input:** LD Point  $P=(X_1, Y_1, Z_1)$  present in registers  $(RA, RB, RC)$  respectively. The curve constant  $b$  is present in register  $RB_1$

**Output:** LD Point  $2P=(X_4, Y_4, Z_4)$  present in registers  $(RA_4, RB_4, RC_4)$  respectively:

- 1  $RB_3 = RB_1 \cdot RC_1^2$
- 2  $RC_3 = RA_1^2 \cdot RC_1^2$
- 3  $RA_4 = RA_1^4 + RB_3$
- 4  $RB_4 = RB_3 \cdot RC_3 + RA_4 \cdot (RC_3 + RB_3^2 + RB_3)$

swayam

So, so they let us now look into you know like you not to understand the arithmetic unit better, let us try to look into the corresponding computations ok. For example, let us try to look into the projective point arithmetic unit. So, here these are my computations. So, if you remember like we essentially described about how to perform point doubling and we also saw how we can convert the point doubling into projective domain ok. So, you can see so, leave it to as an exercise to kind of check for doubling which is which is relatively easy. And, you know like these are the corresponding computations which we perform ok.

So, now what we would like to do as I said is you have to schedule these operations ok. So, these are the fundamental blocks which you have to schedule ok. There is a restriction, restriction is that you have got 1 multiplier at hand because, if you remember in the arithmetic unit there is only one Karatsuba multiplier. So, how many clock cycles would you required to implement this. So, you see that there are 4 clock cycles which have been expended and apparently you cannot do lesser than that ok. So, so now, we will try to understand you know like what or how or what are the time steps of performing this.

(Refer Slide Time: 26:01)

## Projective Point Doubling Sequencing

**Table 6.2: Parallel LD Point Doubling on the ECCP**

Clock	Operation 1 (C0)	Operation 2 (C1)
1	$RC_0 = RA_1^2 - RC_0^2$	$RB_0 = BC_1^2$
2	$RB_1 = RA_1 \cdot RB_0$	
3	$BC_2 = (RA_1^2 + RB_1) \cdot (RC_0 + RB_0^2 + RB_0)$	$RA_1 = (RA_1^2 + RB_0)$
4	$RB_2 = RB_0 \cdot RC_1 + RC_2$	

**Table 6.3: Inputs and Outputs of the Register File for Point Doubling**

Clock	A0	A1	A2	A3	C0	C1
1	$RA_1$	$RC_0$	-	-	$RC_0$	$RB_0$
2	-	$RB_0$	$RB_0$	-	$RB_0$	-
3	$RA_1$	$RB_0$	$RB_0$	$RC_0$	$RC_2$	$RA_1$
4	$RB_0$	$RC_1$	-	$RC_2$	$RB_2$	-

**Point Doubling in Projective Coordinates**

$$Z_4 = X_1^2 \cdot Z_1^2 : X_4 = X_1^2 + b \cdot Z_1^2$$

$$Y_4 = b \cdot Z_1^2 \cdot Z_4 + X_1 \cdot (a \cdot Z_4 + b \cdot Z_1^2)$$

So this is the description of the point a projective point doubling sequence ok. So, you see that these are the 4 clock cycles which are required and this is my corresponding computation which is being performed ok. So, if you observe carefully you will see that this is the parallel architecture for performing the doubling or point doubling ok. So, in particular you see that as I said that in the arithmetic unit there are 2 output block C 0 and C 1. So, you can essentially kind of compute C 0 and C 1 in a parallel manner ok.

So, you see that in C 0 here I use the multiplier ok. So, now RA 1 and RC 1 essentially are inputs which are received from the register file ok. And RA 1 is nothing, but you know like your X 1 and RC 1 is essentially you know your corresponding Z 1. So therefore, right when you are computing this RC 1 which is your output. So therefore, what happens is that if you see that the input to this arithmetic unit is A 0.

So, A 0 essentially will have RA 1 and therefore right so, this will come here, this will come here as an input. And, the other input is A 1 ok, but what I choose in this as an input to this multiplexer is determine by the select line and by the select line which are coming from my control unit ok. So, what I choose is this line. So, what basically what I do is I choose this I choose this corresponding inputs that is my A 0 square ok. So that means, I compute RA 1 square in this way and for this multiplexer or through this multiplexer I choose this line that is A 1 square ok. So, I choose A 1 square here and I choose A 0 square here as inputs ok.

So, now this gets passed into this multiplier and therefore, these multiplier multipliers RA 1 square and RC 1 square as a result ok. In parallel you also calculate RC 1 to the power of 4. So, you see that already in my circuit there are some squarings which have been provided. So therefore, what I can do is I can take for example, A 1 which has got RC 1 now. So, this RC 1 comes over here and this is my RC 1 to the power of 4 because, it passes through 2 squaring circuits ok.

So, now this essentially is my you know like is also you know like something which I also; so, so the C 0 port is used to transfer the output of the multiplier whereas, the C 1 port is used to pass this corresponding result which is A 1 to the power of 4 ok. So, this A 1 to the power of 4 is now pass to this C 1 which means you also have this data ready in the same clock cycle ok. Likewise if you see right so, the now you have basically computed in the first clock cycle this component and you also have calculated in this Z 1 to the power of 4 in the same clock cycle ok. In the second clock cycle what you do is you multiply you know like RB 3 which is your Z to the power of 4 with RB 4. And what is RB 4? RB 4 for was used to store the constant b ok.

So therefore, right you basically get b and you multiply b with Z 1 to the power of 4 and here you essentially compute that, that corresponding result ok. So, likewise you know you also need to calculate Y 4 and you see that the computation of Y 4 has got few multiplication like it has got more than one multiplications which are which are present. So therefore, right for this computation we require more than 1 clock cycles ok.

So, what we do now is that we first you know like in 1 clock cycle we compute this part which is your RC 1 plus RB 1 square plus RB 3 ok, which essentially is computing this part. Remember A A is equal to 1 in this design and the other part RA 1 right is nothing, but you know like is computing you are RA 1 to the power of 4 plus RB 3. So, that essentially computes your this plus ok, this addition gets computed.

So, now in the set in the final clock cycle that is the fourth clock cycle you basically add this component and you essentially get the corresponding result ok. So, you see that you need 4 clock cycles to do this computation. And, if you understand right how this computation has been taking place you also know how the control signals should be generated in time steps and so, that we can perform the doublings operation ok.

So, let me stop here and in the next class right we will be continuing about and see how we can perform the addition operation ok.