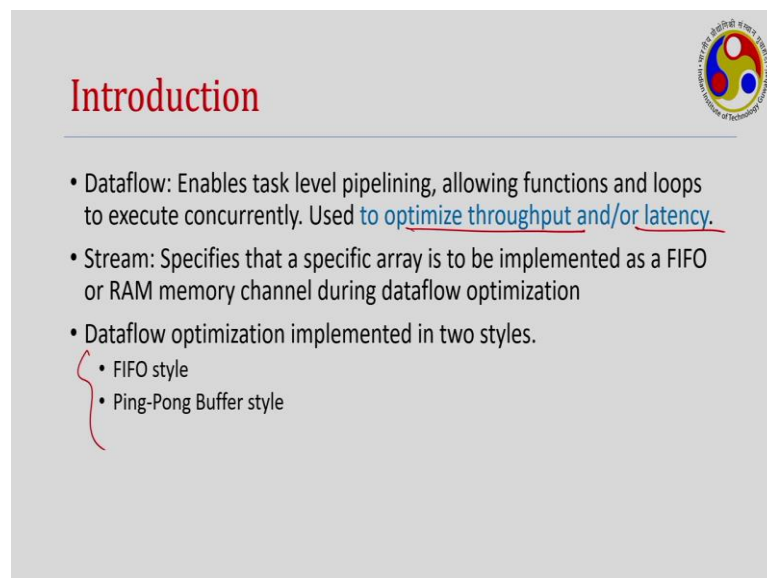


C-Based VLSI Design
Dr. Chandan Karfa
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Module - 08
Hardware Efficient C Coding
Lecture - 28
Dataflow Optimization in High-level Synthesis

Welcome students. In today's class we will learn about the Dataflow Optimization in High Level Synthesis.

(Refer Slide Time: 00:50)

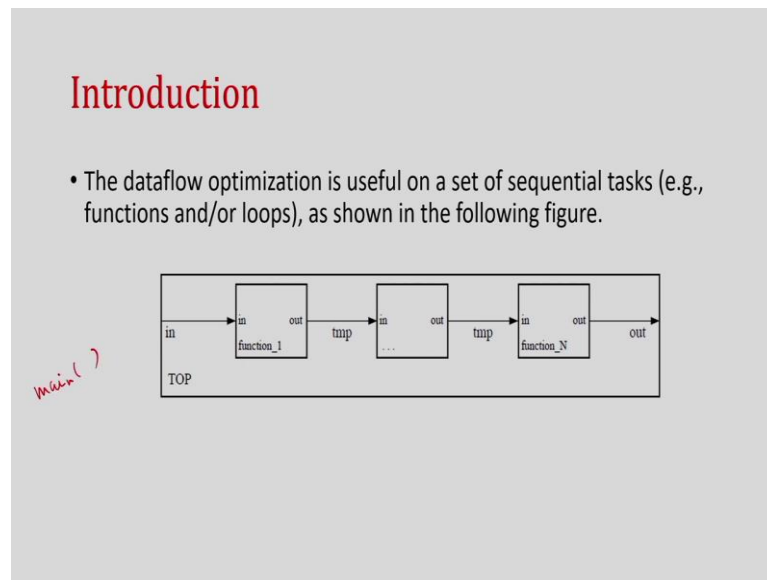


Introduction

- Dataflow: Enables task level pipelining, allowing functions and loops to execute concurrently. Used to optimize throughput and/or latency.
- Stream: Specifies that a specific array is to be implemented as a FIFO or RAM memory channel during dataflow optimization
- Dataflow optimization implemented in two styles.
 - FIFO style
 - Ping-Pong Buffer style

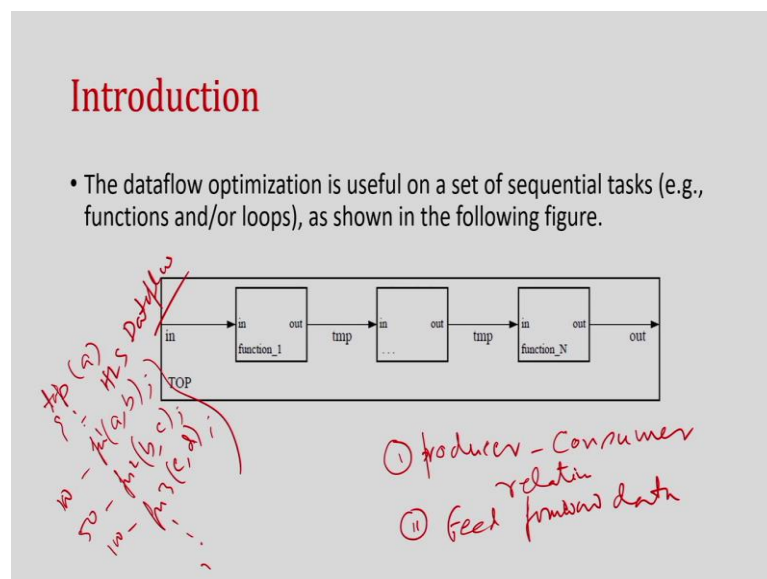
So, data flow is an very important optimizations in high level synthesis in which there are multiple functions which can run in parallel in hardware ok.

(Refer Slide Time: 01:04)



So, let us try to understand what is that. So, let us take an example here. Say suppose you have written a program where you have this say main. So, in usually in high level synthesis main is the test bench.

(Refer Slide Time: 01:19)



So, let say this is my top level function which is say taking a as an input and say its calling a function which sending function 1, sending a and function 1 is producing b ok then you are calling function 2 with b and you are producing c right. And then you are

calling function 3 where you are sending c and producing d ok and there are many other part in the code as well.

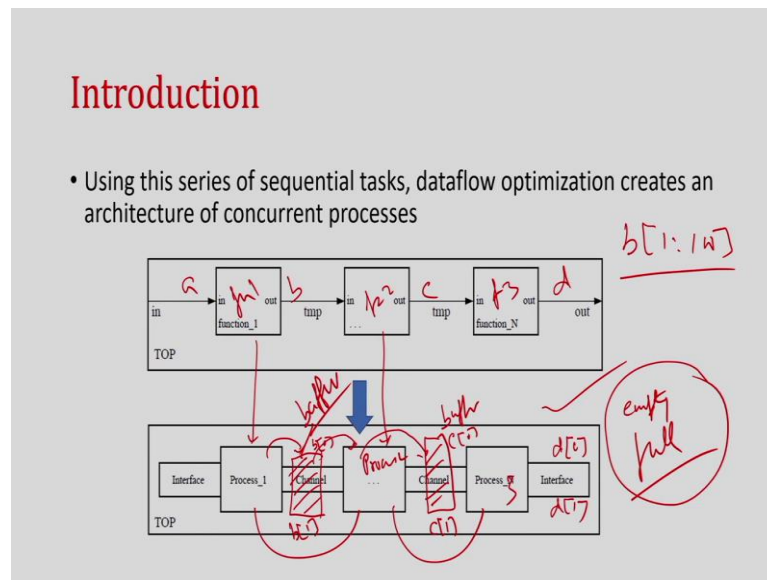
So, in this case whenever there is a function 1 which is consuming something producing something and then function 2 which is consuming the data which is produced by function 1 and producing something which is c which is consumed by function 3 and producing d right. So, this is a there is a producer consumer relation right.

So, this is very important term called producer consumer relation. This is very important and the second thing is a feed forward data right because there is no back edge right. So, it is basically in one way traffic. So, function 1 to function 2, function 2 to function 3.

So, if this kind of behavior you have in some application right. So, then if you think about its execution in c, so, it will complete the execution of function 1 and let say this a, b, c, d all are kind of array and that there are a big loops is happening there right in general its happening same. And then say function 1, say it is taking say 100 cycle ok, then if the function 1 will be completed then I am going to go to function 2 and then I am going to execute function 3 right.

So, let us say function 2 takes a 50 cycle and then function 3 is taking say 100 cycle again. So, what all execution time will be? 250 cycles, this is addition of this because it will be execute one by one sequence value. Since there is a producer consumer relations can I run this particular functions in parallel right? So, that is something is the data flow and this data flow optimizations actually make sure is going to happen that let us try to understand why how ok.

(Refer Slide Time: 03:38)



So, as you understand that this function 1 this is my function 1 this is function 2 and this is function 3 the way I have explained right. So, a is coming here b is getting produced c is coming here and d is getting output right, this is the flow. So, what I understand that since this function 2 consuming the data b or say array b which is produced by f 1. So, b a function 1 has to wait right until this b is getting produced that is why the sequential letter is coming.

So, now what about, but it may be that this since it is running in loop and then this I do not need to wait for the complete production of the whole data a. Say b this b may be a array of say 100 size 100 right. So, there are 100 data to be produced by function 1, but this function 2 may not need to wait for all 100 data to be produced right.

So, in general say a b 0 is produced right. So, first data is produced and this function 2 can take b 0 and process that data. So, if there are some function written in function 2 which is take b and produce say b 0 and produce c 0, right. So, the point I am try to say is that whenever some data is getting produced in function 1 immediately function 2 can start processing that data right and that is what will enable.

So, if we can enable that process then what is going to happen? I do not need to put this function to wait till the complete b is getting produced right. So, this is so, this data flow optimization creates some hardware which makes sure that now this function 1 2 and 3 can run in parallel; obviously, this they follow the producer consumer relations.

So, and basically whenever there is an intermediate data get produced immediately this function 2 can start working on that right that is what is the data flow optimization is all about and let us see how we can enable that things in hardware right. So, basically here we have to make this run function 1, function 2 and function 3 independently without with some handshaking to each other right with some synchronization.

So, it is not. So, as a result what the basic idea is that you add a channel between the function 1 and function 2. So, you implement this function 1 as a module and you implement function 2 as a module right because function gets mapped to some different module in hardware and you add a channel. Channel is nothing but some memory right.

So, I come into that the kind of type of channel we can support here. So, I am going to just put some memory here between process 1 and so, this function 1 map to process 1 and function 2 map to process 2 ok. So, now, I am going to do is that I am going to add a memory in between right. So, buffer, buffer is some memory it can be RAM, FIFO we are come to I will come to that point.

So, what is going to happen here is now process 1 which is equivalent to function 1 produce the data. Say its produce the b in the order like b 0, b 1, b 2, b 3 and so on. So, whenever b 0 gets produced b 0 will come here and immediately this process understand there are some data came and there will be some way to check whether some data is available or not.

Whenever it found that b 0 is consumed so, it can now process b 0 and produce c 0 and it will also be stored in some buffer here right and then say next clock this process b is now writing b 1 to this memory and in the next clock this process to understand b 1 is now available.

So, it will now take this b 1 process it and produce c 1 and its getting written here and similarly process 3 now which know that this c 0 is available in earlier cycle it now can take c 0 and you can produce d 0 right and next clock it understand there is a c 1 and it process d 1.

So, what is going to happen in this way? So, if you just put some intermediate buffer between these two process and such that whenever some this process this all this process works independently and whenever it produce some data it will do not send that data

directly to the next module rather it just put into the buffer in between and there is a handshake method.

So, there are two way you have this process two process can handshake each other is there is a there will be a two signal called empty and full right So, if it is full that means, the process the producer process understand that key this buffer is already full and there is no space to write. So, it has to wait right.

So, now, this process cannot go further until it finds some space to write the data in the intermediate channel or buffer right. Similarly, the empty is important for the consumer process.

So, consumer will always check whether this buffer has data or not. If the empty equal to 1; that means, this buffer is empty right, so, there is no data inside. So, this process has to wait and this situation might occur because this process 1 may be very heavy right. So, producing b 0 might takes a 5 cycle whereas, this process 2 can produce take consume b 0 and produce c 0 maybe in 1 cycle.

So, it is not that this all process will be equally take same number of cycles, but they might they may be basically not so, I would say balanced as a result. There may be some scenario can come in between when the this consumer process has to wait for the data and similarly if the consumer process is very heavy process or there are lot of computations are there probably producer process has to wait in between until the consumer consumes some next data right.

So, this is how there is no kind of overflow or underflow can happen or there is some inconsistency can happen. And the most important point is that just adding this buffer in between we will enable this running this process independently and then run in parallel right. So, that is the biggest advantage right.

So, by doing this what we can do? We can actually achieve we can improve the throughputs; that means, we can produce output more frequently and also probably the latency because now the latency you can understand that because this all processor are now running in parallel, the overall time to compute the data will be much less right. So, that is the what is the overall basic idea of the data flow optimization.

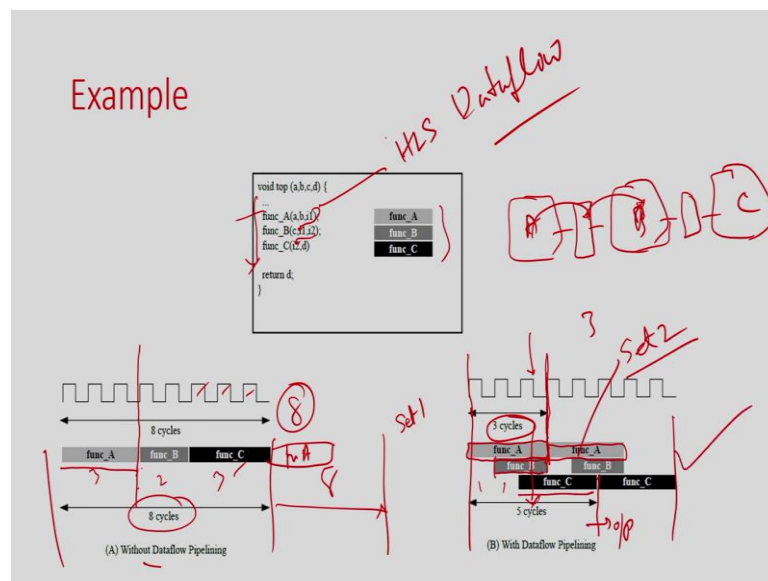
And given such behavior whenever there is a; so, if you just have this kind of producer consumer functions and if you just specify that HLS data flow, so, then the high level synthesis tool like Vivado HLS can understand this. So, the I mean probably I have to implement this as a data flow and you can analyze. So, whether this particular function can be implemented using the data flow. And if yes what it does? It basically automatically generate this kind of hardware with all these handshaking signals right.

So, there will be some handshaking signal between these two process using this buffer between these two process and all. So, that there is no rest condition happen or deadlock happen or say starvation happens. So, everything will be controlled by this synchronization signal.

So, the RTL that will be produced by the high level synthesis tool is in this form and it is the execution of this is completely different from the actual c code because in c code it is going to execute in sequentially whereas, here it will actually happen in parallel right.

So, now that we understand what is data flow and how this high level synthesis enable these things right. So, the buffer that I talked about between two process can be implemented using FIFO and ping-pong right. So, there are two type of way it can be done one is the First In First Out another is the ping pong buffer. I am going to explain this how this things happen with examples ok.

(Refer Slide Time: 11:47)



So, let us first take an very simple example where I try to produce when I try to convince you that how this data flow will improve the throughput and latency ok. So, the same example that I have taken arbitrarily. So, there are say three function A, B, C and then they are actually having a relation produce and consumer.

You can see here that it produced i1 and i1 getting consumed here, it produce i2, i2 is getting consumed here. And you remember that it is not necessary that this function B can only take input from this function A, it can have some other inputs also like C; C is another input here right. So, this can happen.

So, now I have this. Say let say this function 1 takes a 3 cycle right. So, this is 3 cycle; 1, 2, 3, function take 3 cycle and function 2 takes 2 cycle right. So, 3, 2, 3 ok; so that means, it is total 8 cycles. So, if you just execute this without data flow it will actually implement in 8 cycles.

So, it will create a module 1, module 2, module 3 and it will be since there is no data flow there is no synchronization between them. So, it will have a module 1 and module 2 will wait until this module 1 completes and then it will be scheduled like that right. So, first 3 clock module 1, next 2, 2 clocks module 2 and the next 3 clock module 3. So, it will take overall 8 cycles ok.

Now, let us see I have applied data flow here right. So, I just add the pragma HLS data flow. So, now, see how the things will change. So, whenever there is a data flow I already show that it will create three modules right. So, there will be three modules and there will be intermediate buffers right. So, there will be intermediate buffers through which there will be synchronized and run them parallel.

So, now I can run this three module A, B, C in parallel right. So, what is going to happen? So, I will start the process function 1 right and now because so, after some time say after 1 clock it will produce some data and immediately I can now start function B because that is what I have (Refer Time: 13:57) first clock. It will write something here and then that in the next clock B can actually take this data and start this and that is what is happening here.

So, you see in the next here the B start only when a function a completes, right. So, when the function a completes then only function b starts. Here function is B A will going to

complete here, but after 1 clock only because some intermediate data is already available in the intermediate channel function B can start execution right. So, that is the important thing is happening here. And similarly after the next clock comes B will produce something and function C can start his execution now ok.

So, with this you can understand that now after 2 clocks in from 3rd clock onwards all these 3 functions are running on parallel right. So, they are running on different different data, but they are running in parallel and I can see that after 5 cycle because this function 3 will take 3 cycle and two more cycle here, after 5 cycle my first output will be produced. So, it is throughput become in every 5 cycle every 5th cycle here throughputs is every 8th cycle right.

So, the next output will come after 8 more cycle right. So, they are in the non data flow version of the hardware, it will take the throughput will be 8 right, 1 1 in 8th cycle here it is one in 5th cycle. Most importantly since function 1 complete his execution here, now if you give the next set of data in hardware it is not that you are only running for one data right continuous data is coming.

So, it is a kind of stream of data is coming. One set of input comes you process it you get the output then you give the next set of input right. Since function A is completed here I can now the start the execution of the function A again the for the next set of data right. This is for the first set of data right. So, there is set 1 and this is for set 2, right. So, this is the 1st set of data and this is 2nd set of data.

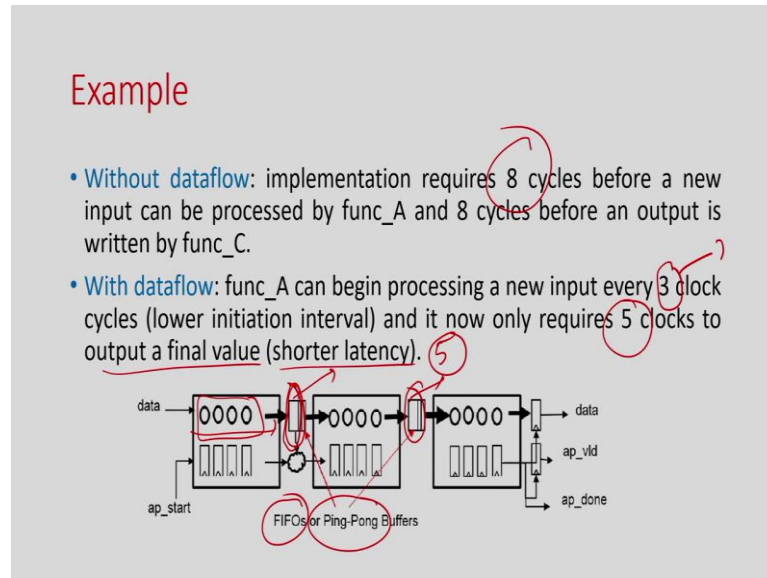
So, now, I can start the whole process for the 2nd set of data after 3rd cycle itself. Here I cannot start it here because this function B will be complete and that is how it is getting scheduled. So, I can only start function A again here. So that means, even here there is no even this for different set of inputs the function also not running right.

So, here immediately after function is completed because now I can start function A on for the next set of data and this is how the overall latency become. You can understand that the latency become, so, basically this latency become 3 right because earlier it was 8 because I can now I can give input in every 3rd cycle right.

So, this is the initiation interval or the kind of latency of the whole system that earlier I can give the next set of data only after 8 cycles, now I am giving here after empty 3 cycle

right. So, that is the advantage of having this. So, you can understand that this is much more efficient design because here because in hardware all modules in running pair parallel and I am actually taking advantage of that using the data flow transfers ok.

(Refer Slide Time: 17:10)



So, that is all about this data flow. Let me move on. So, as I mentioned that this with data flow 8 cycles here it is taking 5 cycles for throughputs. And so, basically sorry. So, the latency I mean I am little bit wrong there. So, that it takes 5 clock to produce the final output. So, latency is 5 and throughput is it actually I can start the process in every input. So, it is basically every 3 right.

So, I mean I hope you understand that. And this is how this good things is happening that you are actually adding some channels here right and which is basically for storing the intermediate difference ok. So, let me try to now move on to this how this exactly it happened right how this channel works. And as I mentioned that there are two kind of channels is possible, ping pong and FIFO ok.

(Refer Slide Time: 17:56)

Dataflow Architecture

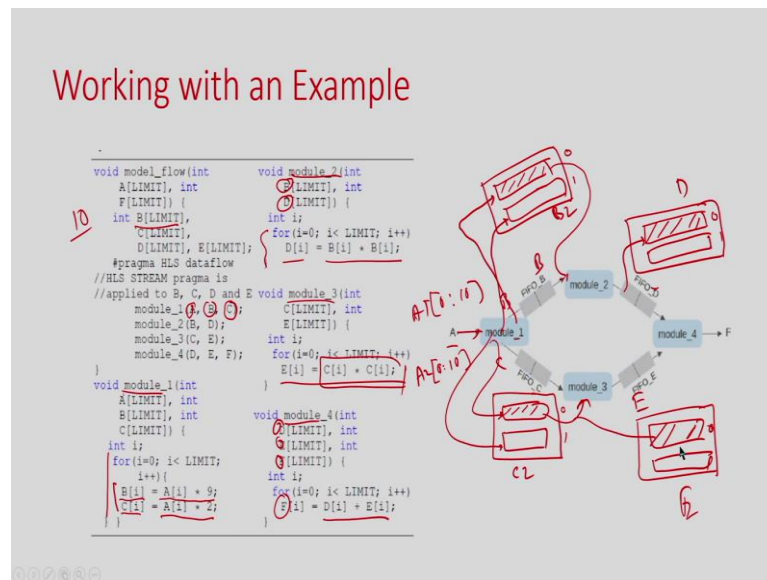
- HLS creates individual channels that model the dataflow to store the results of each task in the dataflow region. These channels can be FIFO or ping-pong (PIPO) buffers
- FULL/EMPTY signals represent a handshaking interface
- By having individual FIFOs and/or ping-pong buffers, each task can execute in parallel
- This allows for better interleaving of task execution than a normal pipelined implementation
- Additional cost: additional FIFO or block RAM registers for the PIPO

So, let me start with this ping pong first. So, as I mentioned that this data flow architecture is something. It basically adds some FIFO or ping pong buffer in between and there are some handshaking interface. So, empty and full and this by this handshaking interface now each task can run in parallel right and then it is also actually allow thus better interleaving of the task.

I can understand that now this all every clock all task is running right. So, all task is running in parallel. But this kind of advantage coming with additional call that means, you need this FIFO and ping pong implementation. So, you need more memory in between because you need to put a large channel there.

So, the size of this FIFO depends on the what is that maximum data can be written here and you can have some calculation to identify what is the minimum size of this FIFO needed to communicate these two, but that some analysis needed, but in general it can be the maximum size of the kind of number of data it gets produced. So, this way this is the extra additional cost which does not need when you run it in sequential manner ok, but with this additional cost the benefits is huge ok.

(Refer Slide Time: 19:05)



So, what I am going to do it now? I am going to take an example, this example and I am going to talk about how this ping pong and then this FIFO style get things get implemented with this example right and how that works actually ok. So, let us take the example here. I have taken a little bit bigger example. So, I have four modules; module 1, module 2, module 3 and module 4 and the data flow implementation is like this right. So, what is happening here, you let us see.

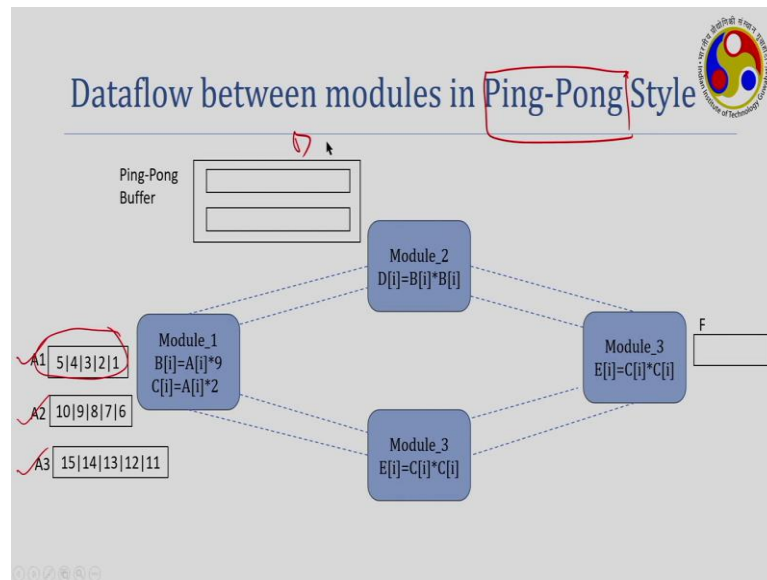
So, module 1 is taking inputs A and B right. So, sorry it is taking input A and it is producing two data B and C. If you see here I am just taking a very simple. So, there is a loop inside the module 1 and it is basically calculating B array B and C which is using this equation right. So, whatever the A i, I am just multiplying with 9 and just multiplying with 2 2 produce C i. So, there are some all dummy operations just to represent things.

And now this B goes to module 2. So, B it is a module 2 now consume B produce D and again there is a loop. So, in the loop I am just doing B star right, B i star to produce D i. So, it is basically making a square of B i to produce D i and then it is getting written into D i. Module 3 take this C which is produced by module 1. So, it this produce B and C, C is getting consumed here and it is producing E using this equation.

Again it is doing C star right. So, whatever the C is coming C i is coming it will do a multiplication of that and it producing E and finally, module 4 is going to consume this D and E and its producing F, it just doing D i plus E i to produce F i. You can see these

are there are loops right. So, all basically module have loops and they are actually taking the input and this is the data flow. So, in this case it is possible to apply data flow because there is no it is a forward flow and this is a producer consumer relations, ok. So, now, let us say I am going to implement this as a ping pong buffer ok.

(Refer Slide Time: 21:11)



So, in the ping pong style, so, the idea here is very interesting that between these two module you just put two buffers ok of the size of B and C right. So, for example, here in this particular ping pong buffer I am going to the size will be of say the limit. So, the limit is say 10 ok. So, that means so, this ping pong buffer has a array of size 10 right.

So, what I am going to do here? I am going to put two buffers ping and pong right. So, it is kind of even and odd, you can understand that even and odd. So, the basic idea is that it will consume one set of A module 1 right, it will produce a B and if the complete B will be stored in this the first one. So, this is a even one and this is the odd one ok and similarly in this particular ping pong there will be two RAM right.

So, basically ping pong is implemented by a RAM and it will give two array ok of size. So, since the limit is 10. So, sub size 10 right, it is ideally. So, here I need two copy of the buffer. So, the which is ping and another is pong kind of ok. So, this B is completely. So, one set of A is taken. Say A 1 is taken which is size 0 to 10 and it will be written here ok. It will be process A star 9 and it will be written here. So, this will be written.

And similarly from A, it will produce the complete array C and it is going to be written here ok and at that moment. So, it will take 10 cycles say to complete this. So, for 10 cycle, the other module 1 start right. So, here it is not that this the sum data is already written. So, I can start it. So, here the style is different I am going to explain that.

B will wait until the complete the even buffer is getting filled right. So, one this one set of data is written here. So, one set of data is now available and it is in the say even buffer ok. So, now for the next after 10 clock I assume that limit is 10. So, after 10 clock now B and C will start accessing this buffer and start working on it right.

So, what is going to happen? From 11th clock onwards whatever the value B and C has first set of B and C, now this module 2 module 2 will consume B and module C is going to consume this C and its going to be written in this buffer. Again here again there will be a two copy. So, this E will be written in the even buffer now and here also the D will be written in the even buffer now ok.

So, there will be two copy of the buffer which is very important and again this is say I am going to write this things data into the D into this buffer. At the same time from the 11th clock, now this module 1 will take the next set of data right. So, first set of data is already processed now it will take the next set of data and it start writing the data in the next buffer right.

So, it will now start writing the things in the next odd buffer. Now, it will take the next set of data and it will going to write in the next buffer. So, this is say C2 and this is also say B 2. You can understand that from after 11th clock module 1, 2 and 3 is now running in parallel ok.

So, after 11 clock module 1 is working on the next set of data and module B and C is working on the first set of data and it is actually still and they are actually taking the data from the even RAM and this guy is writing the odd RAM.

So, there is no conflict right. So, there is no kind of problem or handshaking needed because they are actually writing to different data. Only thing is that now this module 4 cannot work because there is no data. So, unless the whole RAM is full it will not start. So, there is handshaking needed to say that the first the my even buffer is now full right. So, only then only I am going to start this module 4.

So, from 10 to 11 to 20, now this module 1, 2 and 3 is running in parallel where this module 1 is writing to the odd buffer and module 2 and 3 is reading from the even buffers and producing the data input writing into the even buffer in D and E right. So, after C11 to 20 clocks from 21 clock what is going to happen? So, now, after this 11 to 20 all these data will be read right. So, this buffer will become free now.

(Refer Slide Time: 25:40)

Working with an Example

```

void model_flow(int A[LIMIT], int F[LIMIT]) {
    int B[LIMIT], C[LIMIT], D[LIMIT], E[LIMIT];
    #pragma HLS dataflow
    //applied to B, C, D and E
    module_1(A, B, C);
    module_2(B, D);
    module_3(C, E);
    module_4(D, E, F);
}

void module_1(int A[LIMIT], int B[LIMIT], int C[LIMIT]) {
    int i;
    for(i=0; i< LIMIT; i++)
        B[i] = A[i] * 9;
        C[i] = A[i] * 2;
}

void module_2(int B[LIMIT], int D[LIMIT]) {
    int i;
    for(i=0; i< LIMIT; i++)
        D[i] = B[i] * B[i];
}

void module_3(int C[LIMIT], int E[LIMIT]) {
    int i;
    for(i=0; i< LIMIT; i++)
        E[i] = C[i] * C[i];
}

void module_4(int D[LIMIT], int E[LIMIT], int F[LIMIT]) {
    int i;
    for(i=0; i< LIMIT; i++)
        F[i] = D[i] + E[i];
}

```

And this buffer will be full because this the a this module 1 has written to the odd buffer. Similarly, after twentieth cycle this buffer become available and this buffer will be full. So, now, what is going to happen? Now, from the 21 clock to 30th clock, now this module 1 is going to take A 3, the next set of data and it is again going to write it in the odd buffer ok because the odd buffer is free.

So, this is the writing to this buffer will be alternate. For the first set of data in even next set of data it is in odd again is going to write in the even a next set of data will be in odd and this is how things will work right. So, similarly now this module 2 because earlier it has taken from even data now it is going to take it from odd buffer right, the next set of data. So, it is going to take it from the odd buffer.

Similarly, so, here for the third set of data module 1 is writing again to the odd buffer, this is writing to the odd buffer and this module is again consuming from the odd buffers and module 4 now can start taking data from this even buffer right. And since in the next time in the next clock.

So, this is going to be consumed from here and so, next time for the next set of data module 2 is going to write the D into the next odd buffer right because now this buffer is free. So, it is again alternate way it is going to write the data into the odd buffer module 2 and 3.

So, now from 21 to 30 this cycle you can understand this module 1, 2, 3, 4 all are running in parallel right, but they are all running in different set of data and since they are writing in either we have. So, the reading and writing from this channel is happening either from. So, if it the writing is going to happen in the even channel reading is happening from the odd channel right. So, there is no problem.

So, they can actually run in parallel and they can fully it the full even they can actually complete the things right. So, you can understand that after certain period of time all these module will start running in parallel and they do not have to wait for others because this complete data they can read and then it can actually run in parallel.

Only thing is that if this module 1 is a slower because it is happening. So, two operations are here is the only operation. So, this module 2 might complete the processing of this odd data earlier and it has to wait. Because this even channel is not full yet. Until that is get full it cannot start the next iteration that is where the synchronization happen, but the ping pong style.

So, this is how so, this all model, so, there is no interleaving happening. You can understand here that that it is not that the for first whenever this the A 0 is written B 0 is written it start take the B 0 and do that. So, it will actually wait for the completion writing of the or even buffer then only it will start.

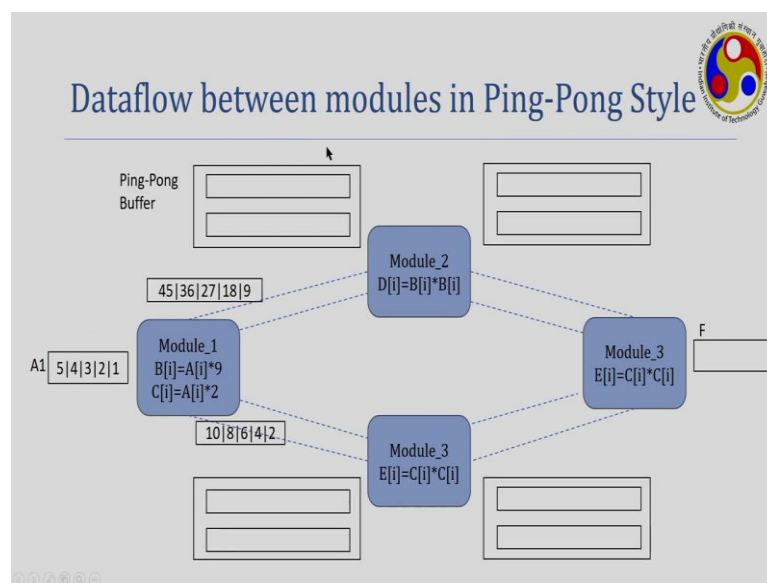
So, this is by managing this even odd buffer or ping pong buffer here. I am actually implementing this whole process and the advantage of having this is that here it does not need that your reading and writing sequence should be same. For example, this may be say from limit to 0, right. So, limit to 0 say, it is different direction ok

So, the order of the writing can be different because it is not that there is no once this guy is writing to this buffer he is reading from this other buffer. So, it does not matter in which order they are writing. So, for ping pong style when the producing order and the consuming order is different it is you have to implement the things in ping pong style

right, but if the this if the producing and consuming style is so, order is same then I can implement this as a FIFO I am going to explain that.

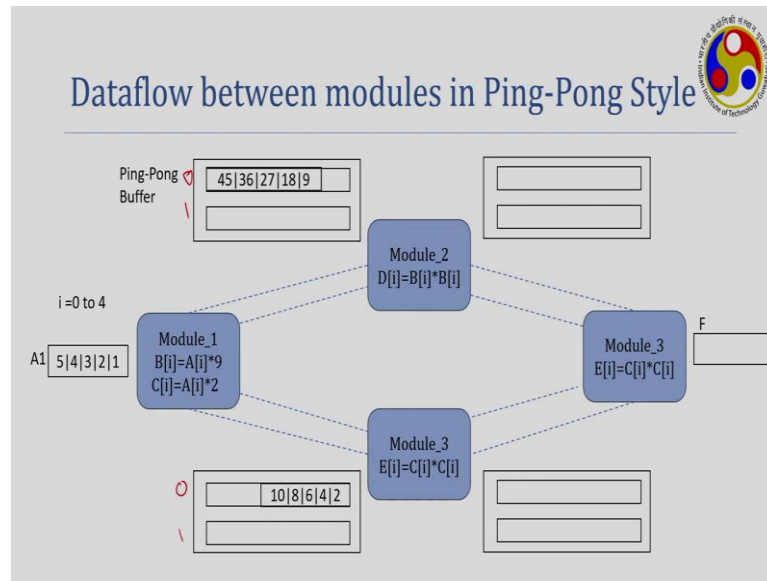
So, let me complete this execution it is animation here. Say let say I have three set of data and first set of data is A 1 then say next set of data is A 2 and third set of data is A 3 and I am assuming the limit is 5 in this example. So, what is going to happen? For the first 5 clocks this A 1 will be consumed and it will be it will be process and it will be written into this buffer B and C here ok.

(Refer Slide Time: 30:03)



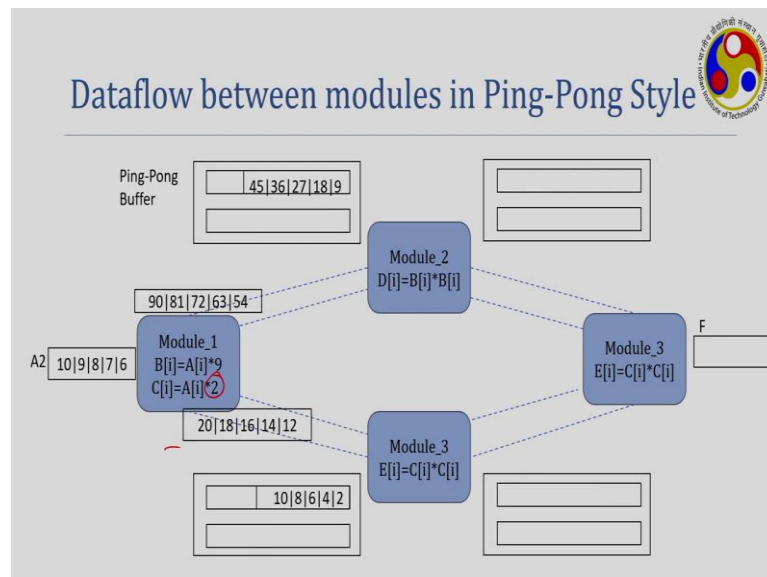
So, what is going to happen? For the 1st clock it will consume this and its multiplying with 9 and its it will be produce this data.

(Refer Slide Time: 30:11)



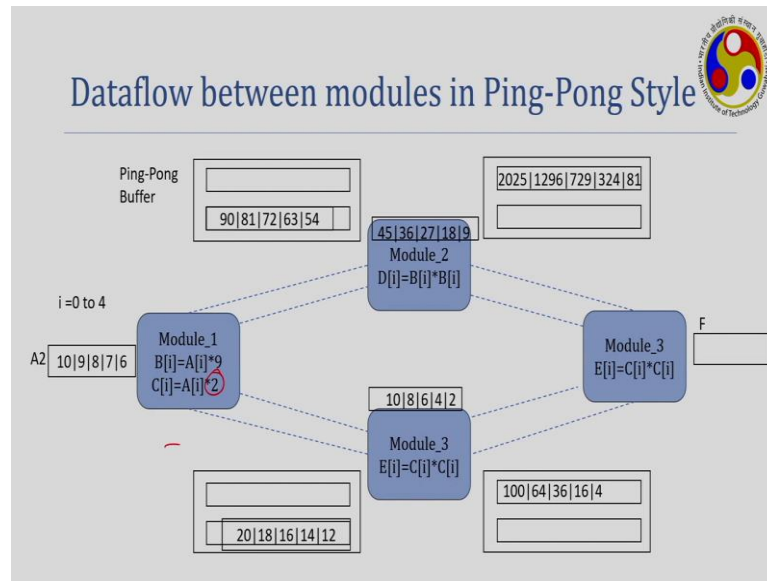
And this data will be written in the even buffer and this is basically say 5 into 2 is 10, 4 into 2 is 8. So, the first set of data will be multiplied by 2 and it will be written into this buffer right. So, this is your even buffer this is odd buffer this is even buffer odd buffer. So, this is for the 1st clocks 5 clocks. Next 5 clocks is going to happen.

(Refer Slide Time: 30:33)



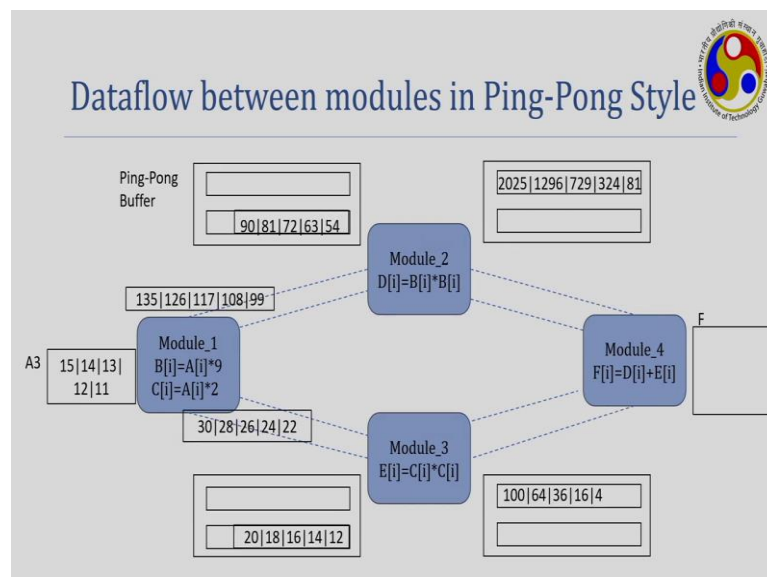
It will now take A 2 it will produce it will multiply the element with 9 and it will produce this data multiply the element with 2 and it will produce this data.

(Refer Slide Time: 30:45)



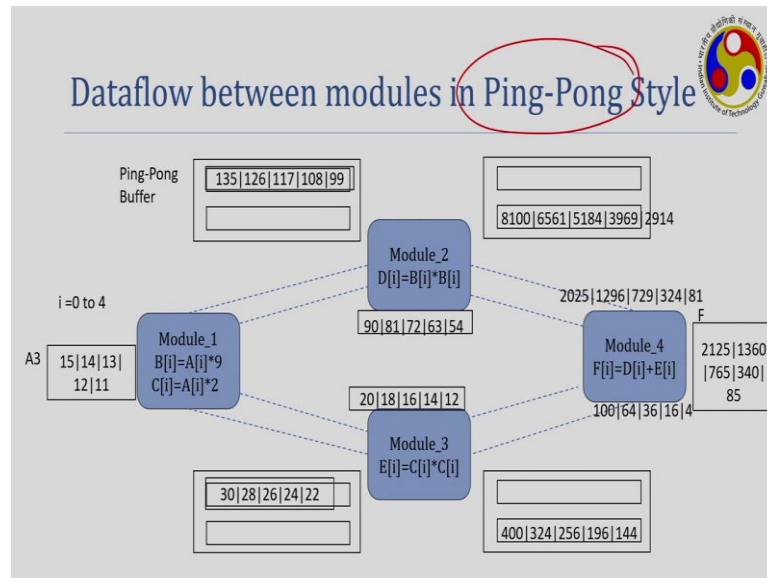
And this will be written into this buffer odd buffer right you can understand that. And similarly at the same time you see here module 2 is now consuming from the even buffer this data and it will do a square and this data will be written into this in this even buffer. So, this all this now the execution of module 2, 3 and 1 is parallel right, but they are writing reading and writing is happening from two different channels right.

(Refer Slide Time: 31:15)



So, the next clock as I mentioned now you are going to this particular module 1 going to take the A 3 and it will multiply again with 9 and it will produce this data.

(Refer Slide Time: 31:21)



And this will be written into the odd channel and the even channel the even channel will be read by this module and this other channel. So, this is the odd channel. It will be written where read by module 2 and module 3 and they will produce the data and it will be written into the buffers. And similarly you can see here this module 4 is now consuming data from here and it is producing the data. So, it is all happening in parallel.

You can see here I am running the animation again. As I mentioned after some time the producing. So, you see here module 1 is producing to these two buffer, module 2 and 3 is producing to these two buffer and at the same time module 4 is also producing to the buffer you can I am running the parallel things again.

So, this is going to write in the odd this is consuming and running in the even and this is reading consuming from odd and writing into the even right. So, the even and odd things is managing right. So, ping and pong. So, this is what is the ping pong style and if the access pattern is different the only way you can actually realize this using ping pong buffer and there are two copy of the buffer is needed ok. Now, I am going to take the same example and fortunately in this example the order of axis is same.

(Refer Slide Time: 32:37)

Working with an Example

```
void model_flow(int A[LIMIT], int F[LIMIT]) {
    int B[LIMIT], C[LIMIT], D[LIMIT];
    #pragma HLS dataflow
    //applied to B, C, D and E
    module_1(A, B, C);
    module_2(B, D);
    module_3(C, E);
    module_4(D, E, F);
}

void module_1(int A[LIMIT], int B[LIMIT], int C[LIMIT]) {
    int i;
    for(i=0; i< LIMIT; i++)
        B[i] = A[i] * 9;
    C[i] = A[i] * 2;
}

void module_2(int B[LIMIT], int D[LIMIT]) {
    int i;
    for(i=0; i< LIMIT; i++)
        D[i] = B[i] * B[i];
}

void module_3(int C[LIMIT], int E[LIMIT]) {
    int i;
    for(i=0; i< LIMIT; i++)
        E[i] = C[i] * C[i];
}

void module_4(int D[LIMIT], int E[LIMIT], int F[LIMIT]) {
    int i;
    for(i=0; i< LIMIT; i++)
        F[i] = D[i] + E[i];
}

//Handwritten notes:
// 0, 1, 2, 3 }
// 0, 1, 2, 3 }
```

```
graph LR
    A --> M1[module_1]
    M1 -- FIFO_B --> M2[module_2]
    M1 -- FIFO_C --> M3[module_3]
    M2 -- FIFO_D --> M4[module_4]
    M3 -- FIFO_E --> M4
    M4 --> F
```

You can understand that this will be produce B 0, B 1, B 2, B 3 and write that as a B 0, then 1, then 2 then 3 and here consumption is also 0, 1, 2, and 3 right. So, here the production of the producing order of B and consuming order of B is same. Similarly for C because the loops are all same size right. So, the production and consuming order of C and D n is also all same right.

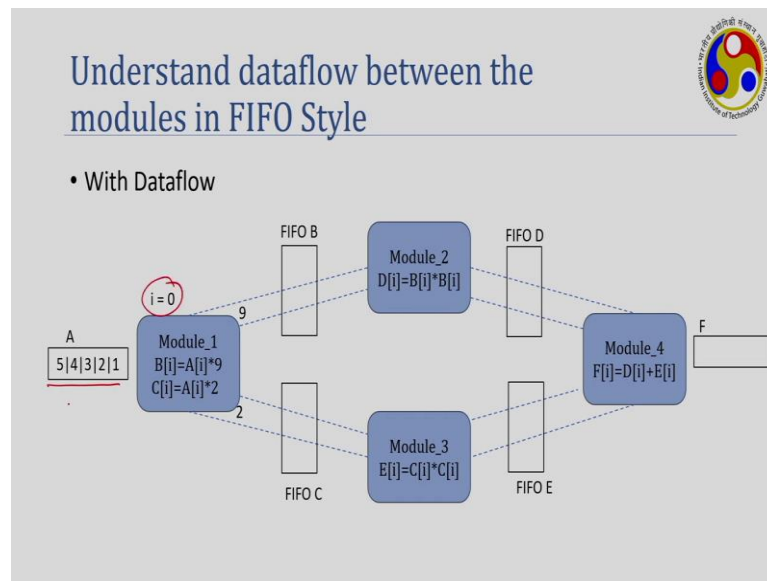
So, in this case I can also implement the things with FIFO ok and to do that with data flow you have to specify the HLS stream called pragma called stream HLS stream. So, if you say that I want to make it a streaming application then it will not implement that with a ping pong.

Ping pong will be the default option, but if this producer consumer the access that producing order and consuming order is actually same I can actually make the FIFO and which is FIFO is something in which the array this buffer size will be not double right.

So, in ping pong there are two buffers is needed, in FIFO I need only one copy and most importantly this modules can actually do not have to wait for the entire data to be produced. So, whenever one data is produced because that the consumption will be the same order immediately that can be consumed. So, the much inner level this coarse grained parallelism can be happened right.

So, this now I can actually mixing the iteration of the modules also I can parallelize. Earlier it was although they are running in parallel, but they are running on two different set of data right, but here even I can for the one set of data I can parallelize the 4 module and I can run all 4 module running for the same set of data even. Obviously, I can I mean which will be parallelized for different set of data, but even within for one set of data also different iterations of these modules can run right.

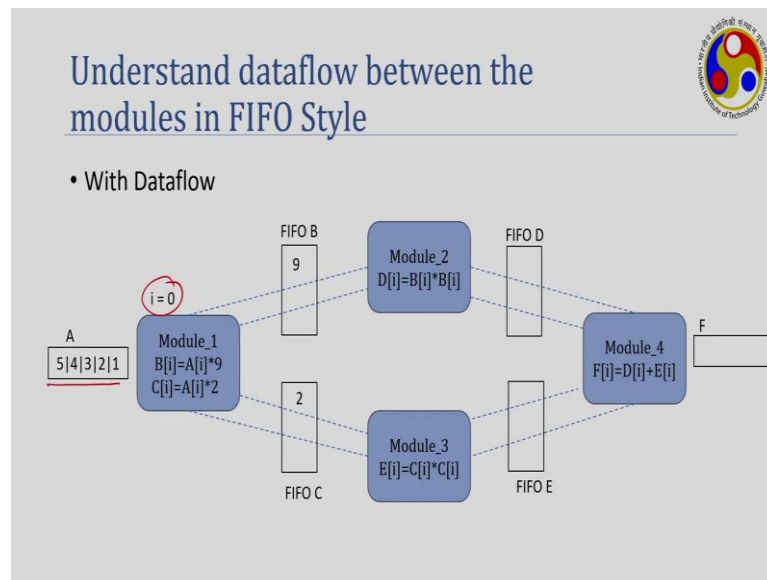
(Refer Slide Time: 34:28)



So, I am going to explain this with the same example. So, let us take the same example and say initially I have this data array 5 size 5 the same example. So, this is module 1, 2, 3 and 4 and this is the FIFO. So, FIFO transfer first in first out; that means, whatever I am going to giving the input and the same order it will come out right.

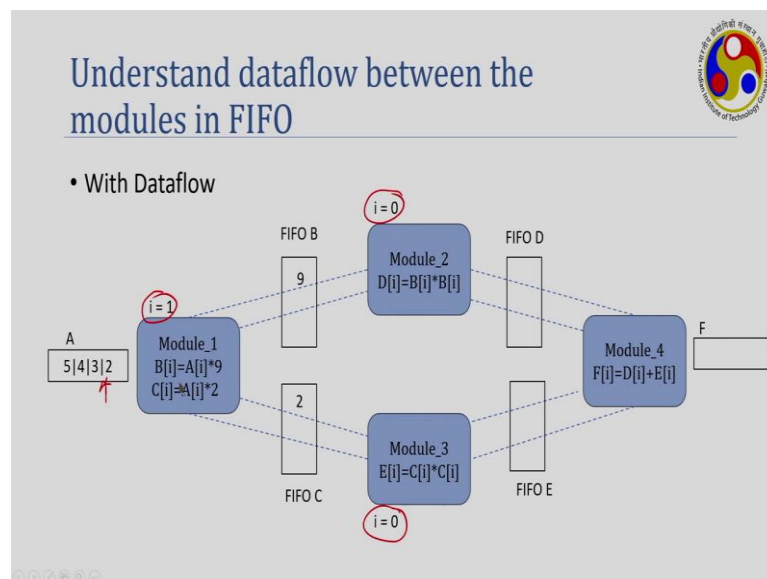
So, it is kind of a queue right. So, whatever the data you are writing it will be consumed in that same order ok. So, now let us see. So, whenever my I equal to 0, so, what is going to happen? That B 0 and C 0 will be produced, right.

(Refer Slide Time: 34:59)



So, since it is 1, I am going to multiply with 9. So, it will be 9 and it is 1, so, it will be 2. So, this is my B 0 and this is C 0, it is produced, 1st clock is done. The next clock now this FIFO will say [FL] it is not empty. So, B module 2 and 3 you will understand now my FIFO B and FIFO C is not empty. So, they can consume B 0 right.

(Refer Slide Time: 35:23)

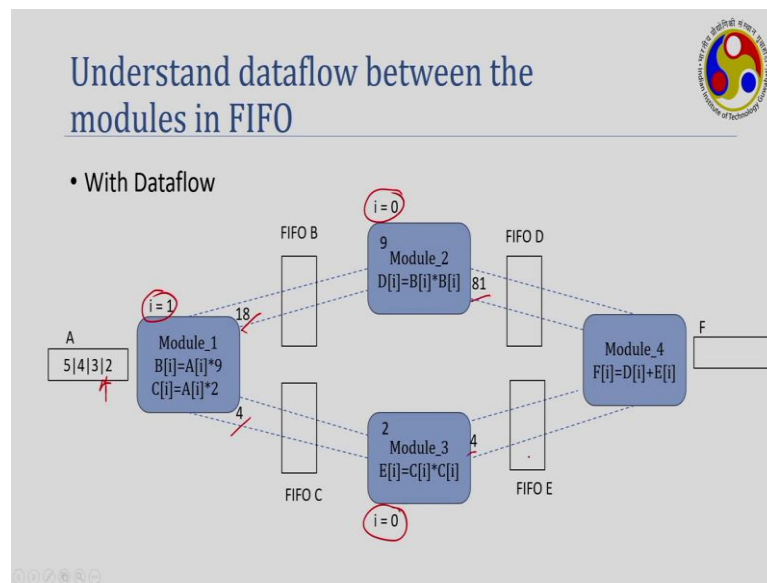


So, what is going to happen now? So, they can now start their iteration 0 right. So, they can take now and module 1 now actually run the iteration 1 because 1st iteration is

already done, now it will run the say next iteration, iteration 1 and module 2 and 3 is now C can run in iteration 0.

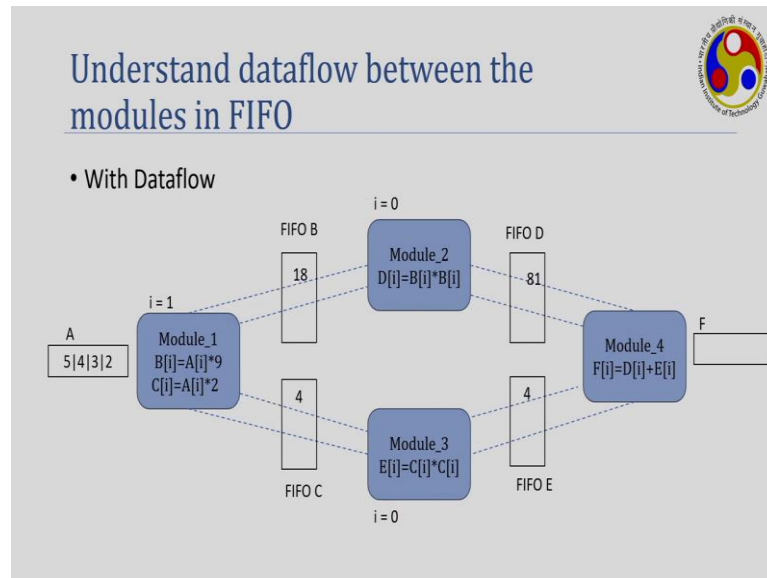
So, this was not happening in ping pong. Ping pong it was the complete execution of module 1 is done then only module 2 and module 2 are starting right. So, it has to wait for first n cycle. Now, here after immediately after 1st clock in the 2nd clock this module 2 and 3 can run their 1st iteration whereas, the module 1 will run for the next set next data on 2 ok.

(Refer Slide Time: 36:09)



So, you see what is happening here now animation very closely.

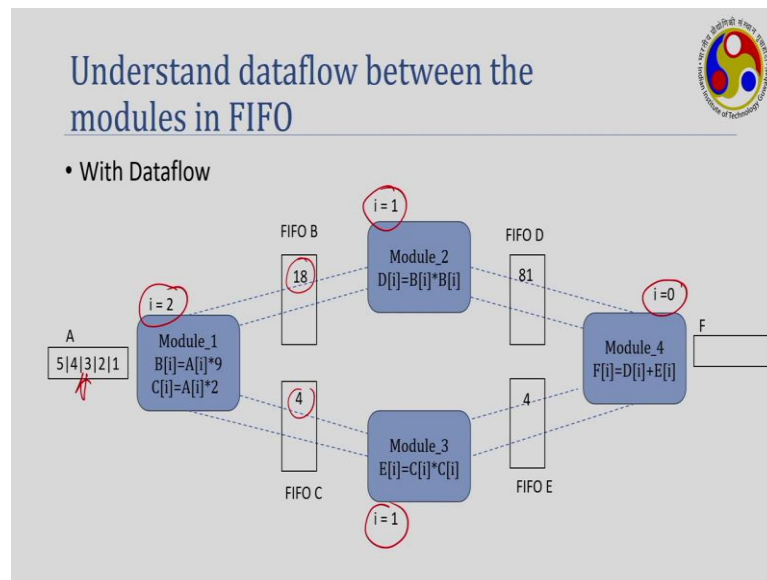
(Refer Slide Time: 36:39)



So, you can see here now 2 and 9 will be consumed by module 1, 3. They will produce 9 into 9, 81 and 2 into it is 4. So, and similarly this module now taking the A1 which is 2 and will produce 9 into 2 is 18 and 2 into 2 is 4. So, they will produce this and this will be written into this buffer.

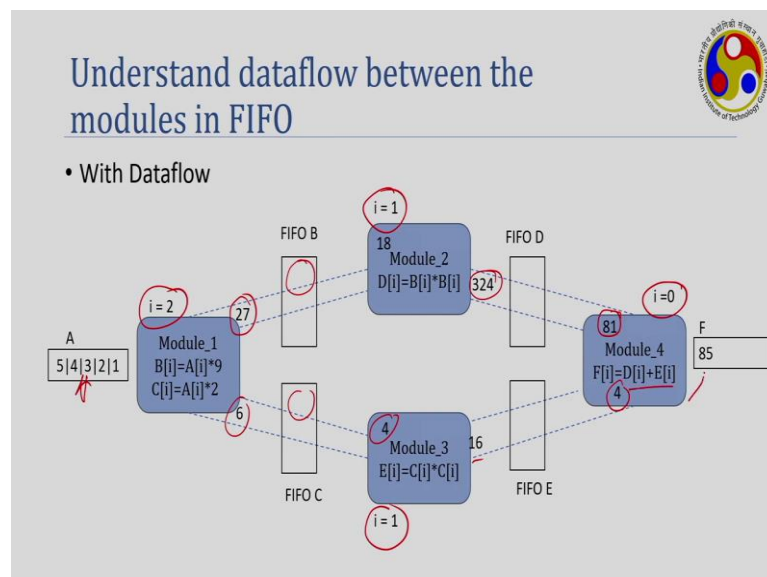
18 and 4 will be written into this buffer FIFO B and C and 81 will be written into FIFO D and 4 will be written into FIFO E right that is what is going to happen in right. So, you can see here. So, they got written here. So, now, after this clock, so, this happened in the 2nd clock. In the 3rd clock it is going to happen. Now, this module 4 is can also run is a iteration 0 right.

(Refer Slide Time: 36:52)



So, now, module 4 also can run in iteration 0, module 2 and 3 will run that iteration 1 and module 1 will run that iteration 2 right. So, it is now going to consume 3. It will consume this these two data and it will produce the D and D1 and E1. And this will take this two data and it will produce F0 right, it is actually a consuming D 0 and E 0 and it is producing this right.

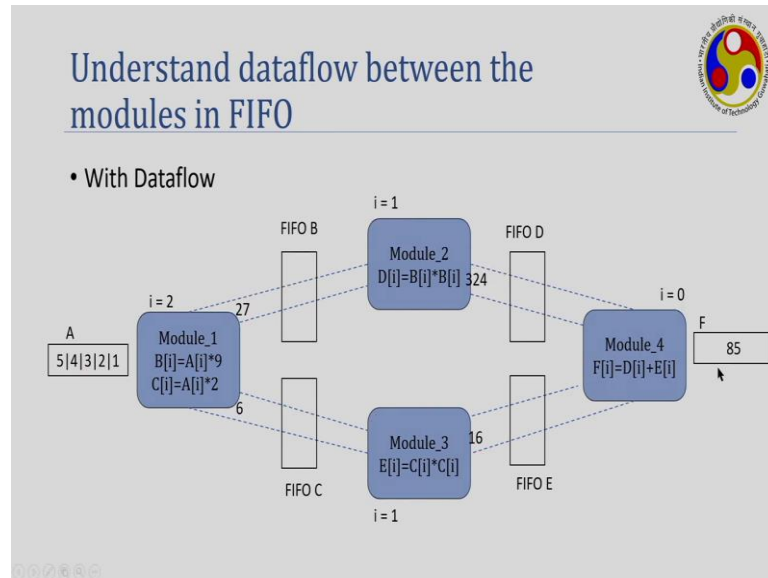
(Refer Slide Time: 37:19)



So, now you can see here after 3rd clock you can see here all, these things is happening parallel this is very interesting right. So, now, so, this is. So, it will take 3 right. So, it

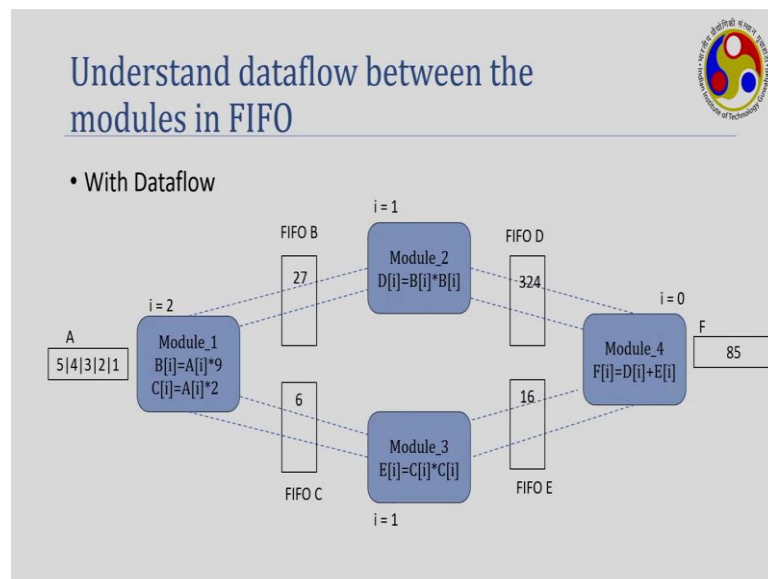
will take 3 it will produce 27 and 6 right and this will take this 18 and it will do a square of 18 square which is 324 and 4 square is 16 and that will be written here and here it was 18 and 4 81 and 4 was there it will consume and it will do a addition it will write into this right.

(Refer Slide Time: 37:47)



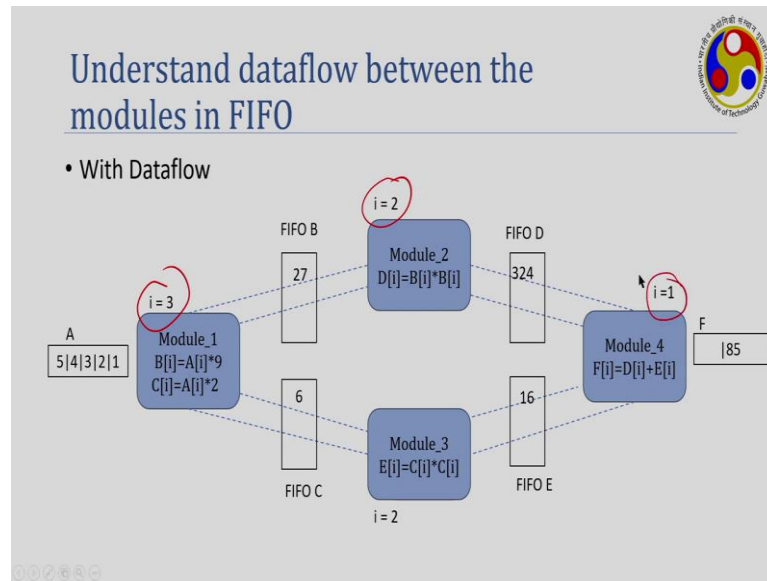
So, now you can see here all things are happening in parallel right.

(Refer Slide Time: 37:51)



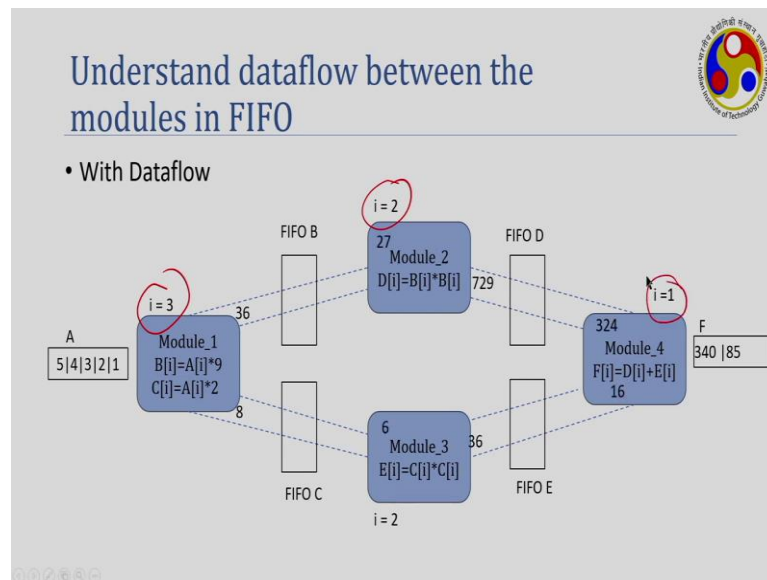
So, this now this will be produced here and it will produce here.

(Refer Slide Time: 37:54)

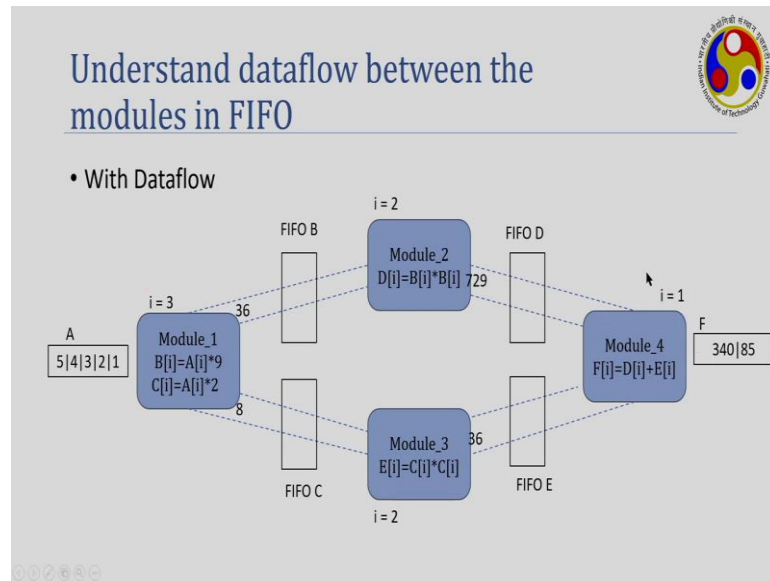


So, this way it will now run this A3, it will run the A2; i 2 and this is run the i 1 and this all running in parallel right.

(Refer Slide Time: 38:01)

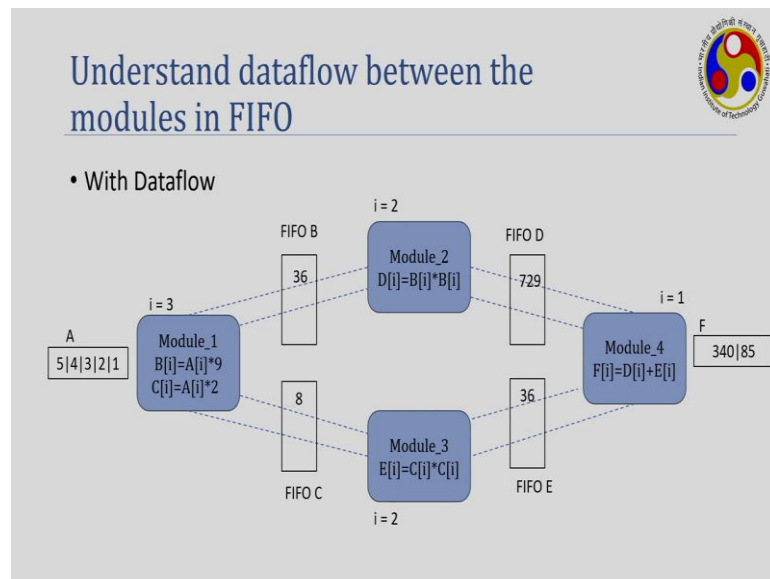


(Refer Slide Time: 38:02)

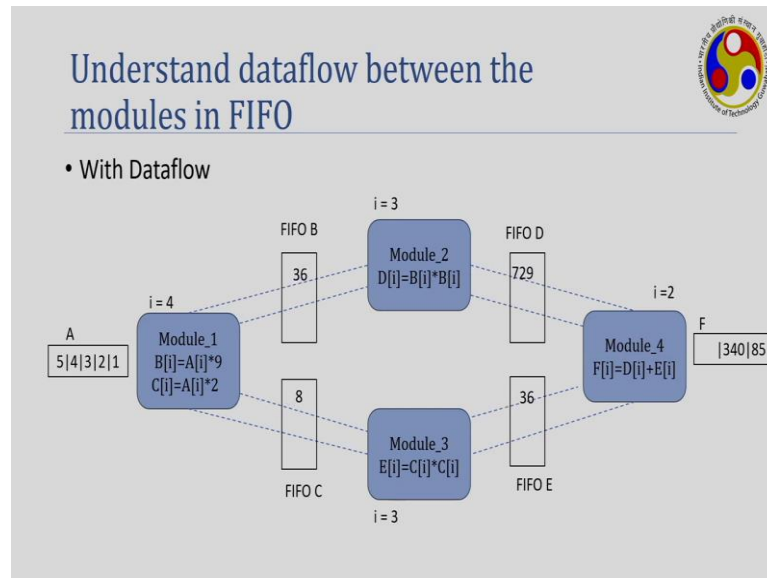


So, this is how things will move on right.

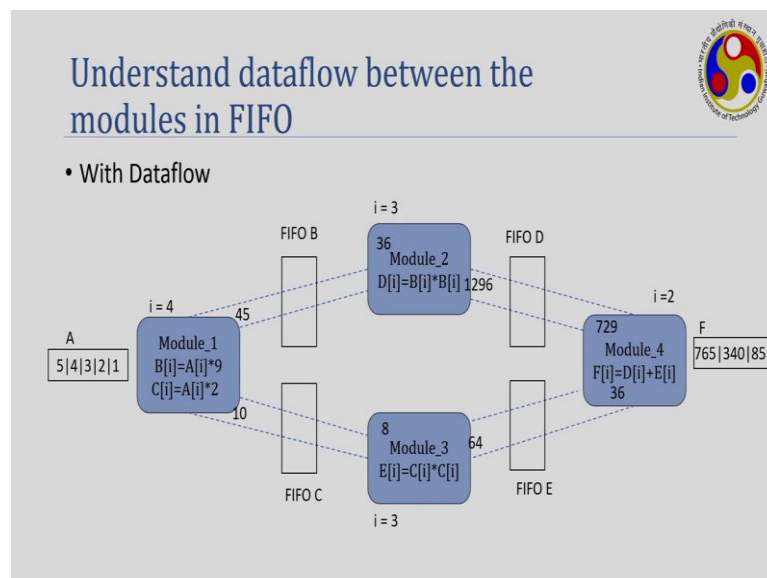
(Refer Slide Time: 38:05)



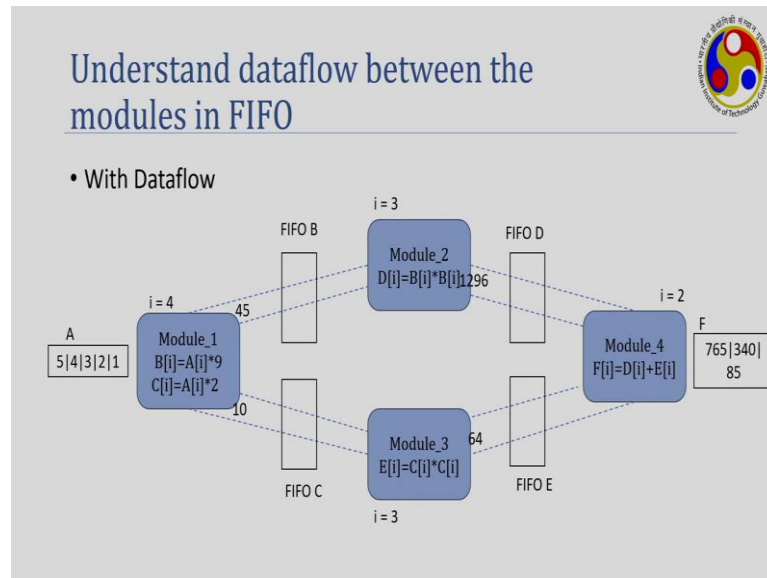
(Refer Slide Time: 38:07)



(Refer Slide Time: 38:09)

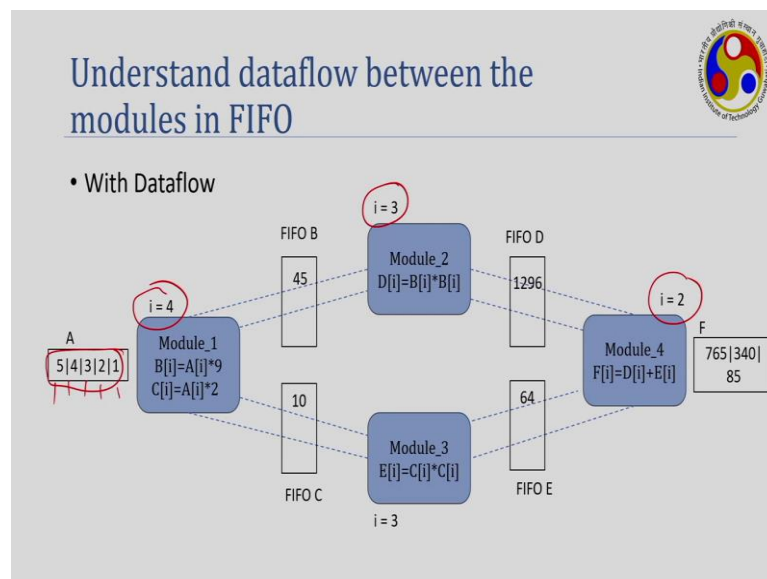


(Refer Slide Time: 38:09)

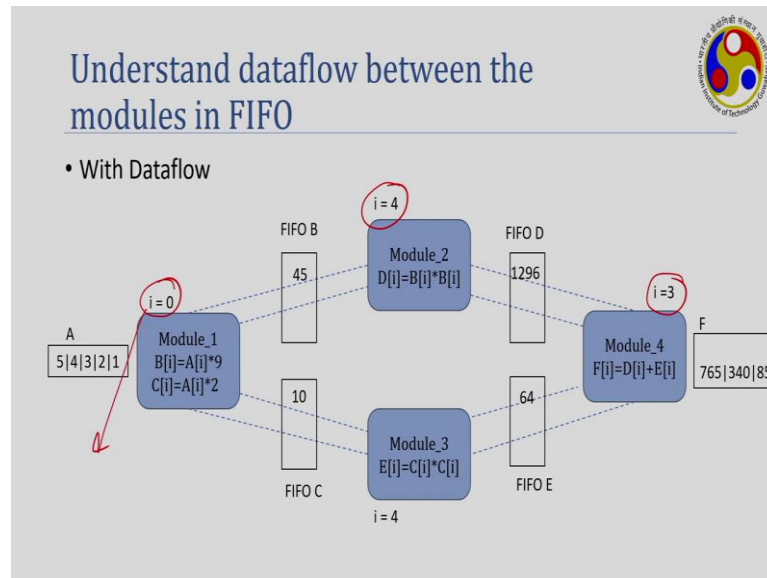


So, this way it will continue.

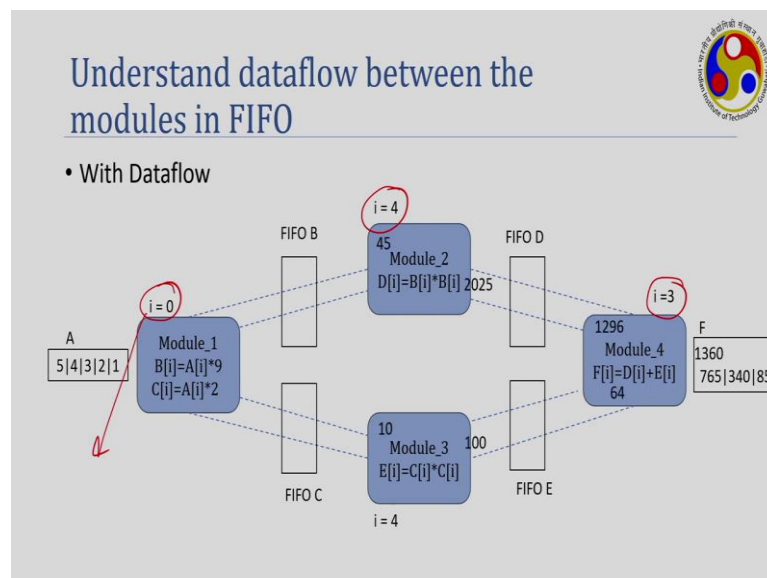
(Refer Slide Time: 38:11)



(Refer Slide Time: 38:12)



(Refer Slide Time: 38:14)

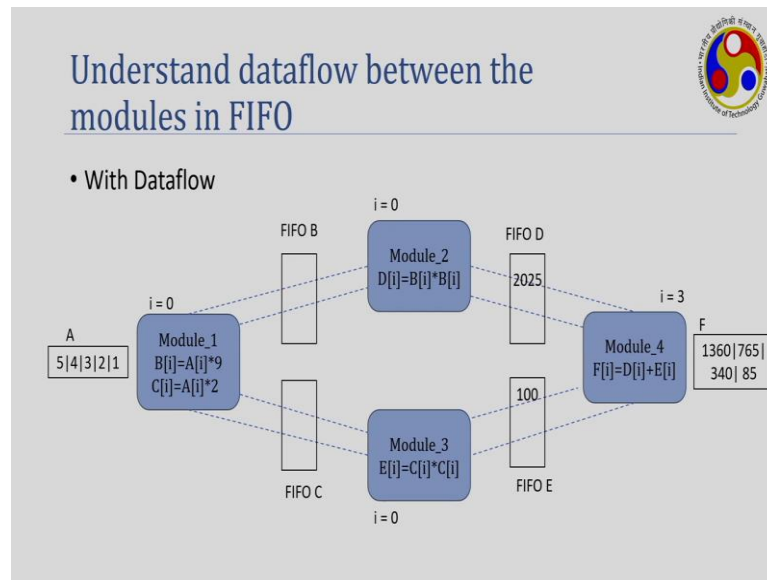


And finally, so, you can see here. So, when once this you can see here. So, whenever this is running the i equal to 4 that means the last iteration 5th iteration they are running i equal to 3 and this is running the i equal to 2. So, next this all set of data is actually produced right. This is A_0, A_1, A_2, A_3 and A_4 . So, after this cycle this will complete the execution for the first set of data.

They will have some still left and this will have two more iteration left here, one more iteration left here there is no iteration left right. Now, next is important. Now, you see.

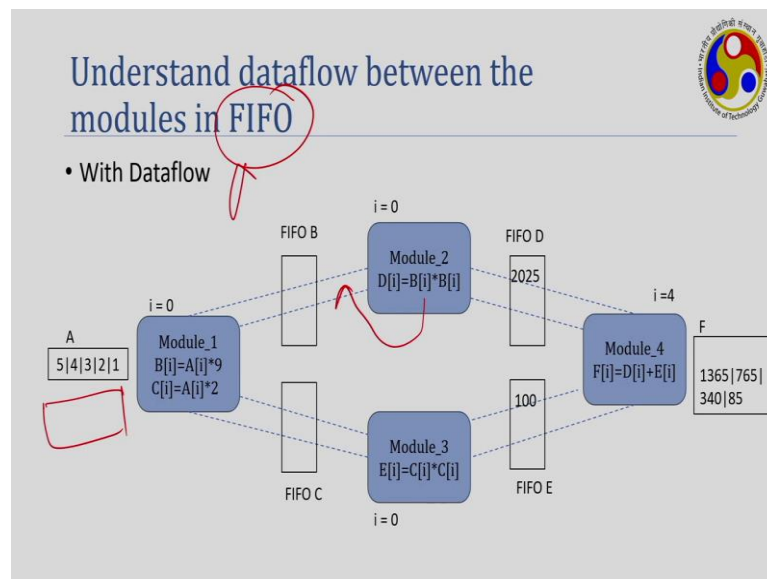
Now, for the it will now start the next set of data it will now start accessing the next set of data and this is actually working on the i equal to 4 for the first set of data. This is still working on the first set of data where i equal to 3. So, this i equal to 0 means, it will now consume the next set of start consuming the next set of data which is something say.

(Refer Slide Time: 39:09)

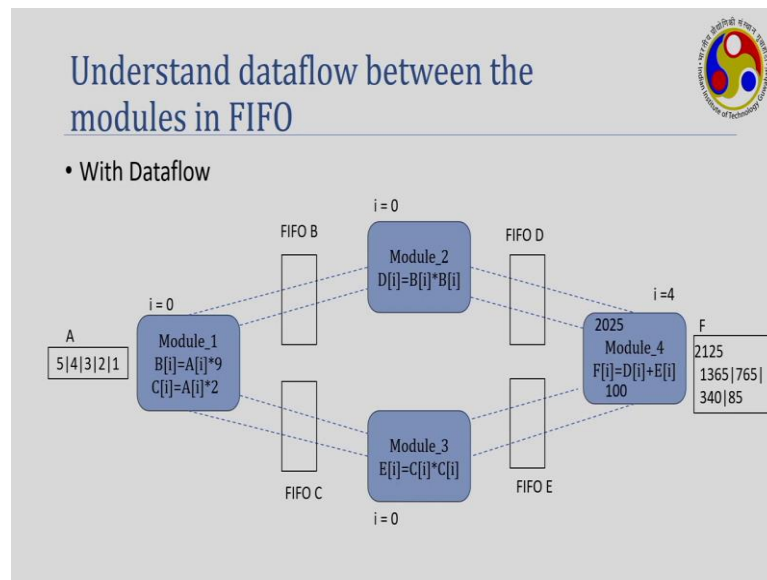


So, this will it will be completed and it will start executing this.

(Refer Slide Time: 39:12)



(Refer Slide Time: 39:15)



And if you do not have any data it will wait because there is no data, so, it will consume here. But in general so, I just give the animation for one set of data, but in practice you can actually whenever this become i equal to 0 you give the next set of data and it will start consuming this data it will start producing the next set of input.

So, this is how FIFO makes sure that this modules actually can interleave their iterations also right. They can parallelize their iterations as well and they eventually actually can produce then can actually run for the same set of data they can run parallelly even for the same set of data.

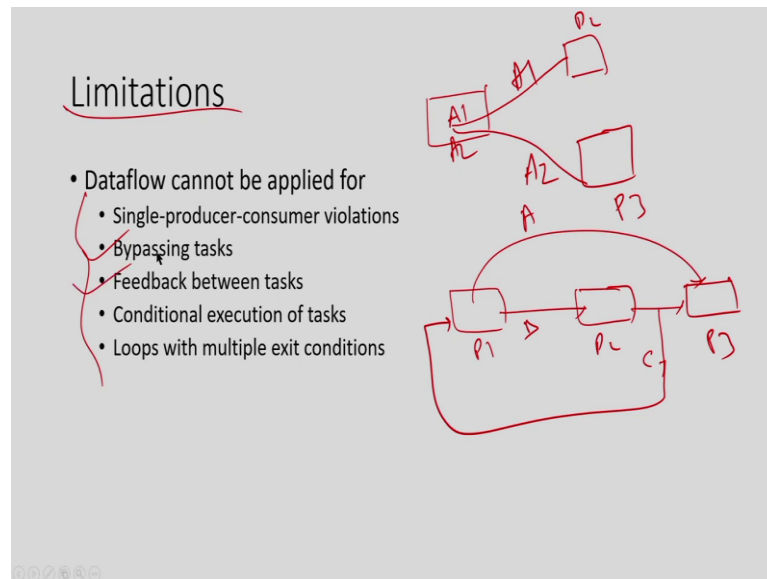
And finally, once the first set of data is over now this module you can start consuming the next set of data and since this is order is different, so, they are going to be written in other places. So, whenever this guy is going to consume, it will consume the previous set of data right. So, I do not actually create any problem ok.

So, this is how this FIFO realize the data flow and FIFO only can be applicable when the writing order and reading order is same that means, the producing and consuming order is same then only I can implement FIFO. If it is not if you ask the tool to apply FIFO it will say it is not possible to apply FIFO because the produce order and consume order is different.

So, then it will be implemented by ping pong style because in ping pong style there are two buffers and the access is random right. So, any order the producer can produce, consumer can actually consume in any order. So, you do not create any problem, but it need double buffer right. So, here I need only one set of buffer ok.

So, this way it will complete. So, I hope you understand what is data flow and how it actually improves the performance in the hardware and its actually true truly hardware and hardware parallelism right. So, although the actual C program is in sequential, but it actually extract the parallelism among these models and actually enables the hardware which actually is true truly parallel ok.

(Refer Slide Time: 41:13)



So, there are some limitations in some cases where you cannot actually apply this data flow. For example, if these are the case where you cannot actually apply, so, let me explain one by one say single producer consumer violation. So, the idea here is that if you produce say A here and if you send this A to say process 2 as well as process 3, you are sending A, so, then it unable to implement this. If you produce one data it should be consume only one consumer not by multiple consumer.

The best solution is that you copy this right. So, you make it A1 and A2, you put A1 here and you put A2 here then it will work right. So, the idea is that one data cannot be sent to two consumer ok. Bypassing task is something like this. So, if you have say this is a process 1 and there is a process 2 and there is a process 3 and it is producing say B and it

is consumed by here and it is a producing A and it is consumed by process 3. So, this is not allowed.

So, it is basically should be it forward flow, it should not bypass any of the process. So, if it is there it will it you cannot implement this data flow right. So, this is what is the bypass task. Feedback between task is not obviously, you cannot work on that so if there is a some the C is getting produced and it is consumed here right. So, then it cannot run in parallel because there is a complete dependency right. Until the these things is done and then only you can start the next iteration.

So, there is no point of implementing data flow. So, if there is a feedback you cannot implement data flow ok. And then conditional execution of task is obviously, you understand that if say some condition is true then only you do the data flow otherwise you do not do that that also is not supported.

In that case you cannot actually implement data flow. And you as a example I have taken there as a loop right and if the loop should be here you could not have a break exit in the loop right.

So, if there are multiple exit you cannot implement data flow because this data flow needs lot of synchronizations and once you have this multiple exit the synchronization becoming become too complex and the modern day high level synthesis tool still will not be able to handle such complex scenarios ok.

But if it is forward flow as I mentioned there is this conditions are not there then and then you can actually implement this using data flow and data flow gives you a efficient hardware ok.

So, with this I conclude today's class. In the next class, I will take another interesting topic that your actual code may not be in the way it is written that your data flow cannot be applied, but we can actually rewrite the code to make it efficient for data flow implementation because the data flow of implementation gives a huge performance benefit.

And you so, basically in modern day if you think about this machine learning algorithm, so, if you understand this machine learning algorithm or image processing algorithms

where there is a lot of loops and lot of data is getting produced and this get consumed by the next one right. So, the idea here is that can I rewrite my code somehow?

If I if you take the original code it is may not be there is no data flow, but I can actually rewrite the my code such that I can actually create the data flow instances and then I can actually run this modules in parallel right. So, then so, that is something a very interesting area of research because if you just can do this data flow implementation that is give you a huge benefit right.

So, in next class I am going to take two such examples and I am try to show that the original behavior is not supporting data flow, but this is how you can modify and then after the modifier the data flow will be supported and you get the maximum benefit ok. So, with this I conclude today's class.

Thank you.