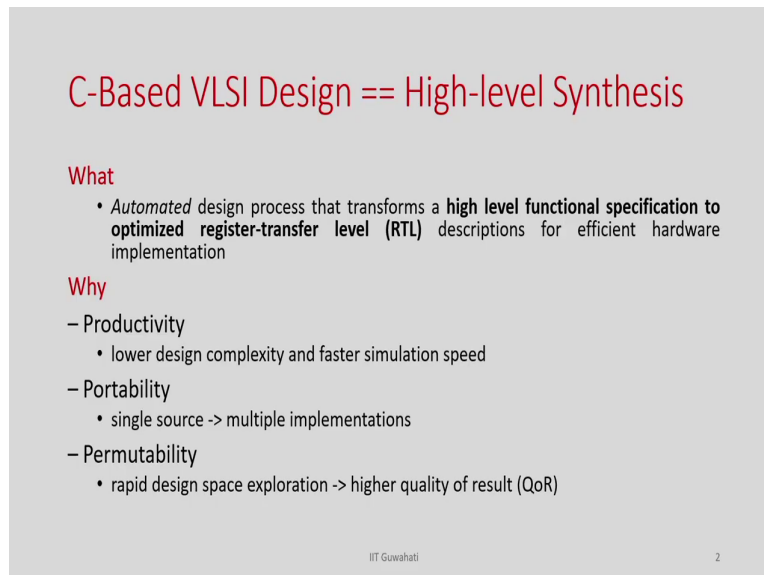


**C-Based VLSI Design**  
**Dr. Chandan Karfa**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Guwahati**

**Module - 01**  
**Introduction to C - based VLSI Design**  
**Lecture - 03**  
**C- Based VLSI Design: Problem formulation**

Welcome students, in this class we are going to learn about this in more detail in high-level synthesis.

(Refer Slide Time: 00:58)



**C-Based VLSI Design == High-level Synthesis**

**What**

- Automated design process that transforms a **high level functional specification to optimized register-transfer level (RTL)** descriptions for efficient hardware implementation

**Why**

- Productivity
  - lower design complexity and faster simulation speed
- Portability
  - single source -> multiple implementations
- Permutability
  - rapid design space exploration -> higher quality of result (QoR)

IIT Guwahati 2

So, in the previous class, we try to see we take an example and so, how high-level synthesis can convert C code into an equivalent register transfer level code. So, we try to formulate the problem in each step right what are the problems are we going to solve in every step, and what are the challenges I am going to face in each stage that I am going to talk about also, I am going to talk about giving you a very high-level idea, how the common C constructs get mapped into the equivalent things in RTL.

So, just to start with that we have already discussed that C based VLSI design means we will start the VLSI design flow start from C code, and then we will convert it into RTL and from RTL we will do this gate-level transformation by logic synthesis tool and from gate level to transistor-level by physical synthesis tool and so on.

And then finally, we will get that cheap IC after fabrication, but we will primarily focus on this high-level synthesis because that is something that is completely new in the context of EDA design flow right. So, the logic synthesis and physical synthesis parts are kind of matured. So, the primary focus of this course will be on this C to RTL conversion by high-level synthesis.

So, we have already seen that high-level synthesis is something convert and equivalent to C code into an RTL or register transfer level code and it has become a unique benefit like productivity, portability, and permutability right. So, productivity in the sense that, it lowered the design complexity because you do not have to develop anything on RTL, you can write just specifications in C and you can completely get your IC quickly because you do not have to develop your RTL and also see if you.

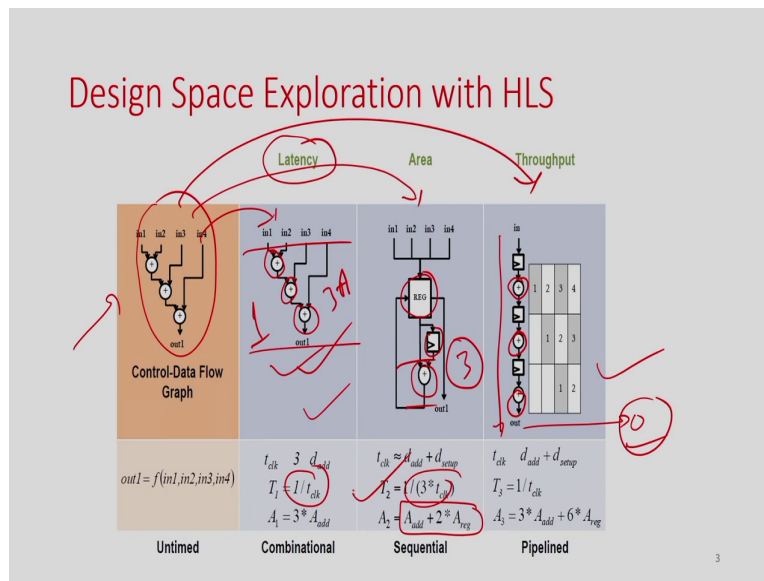
So, checking the specification correctly or not will be done at the C level which is much faster than the RTL simulation. So, you have this low design time as well as low design complexity and it is a faster simulation so, faster verification ok. Portability is something so, we have a specification written in C, you can actually map it to FPGA you can map it to ASIC because the high-level synthesis tool you can just set the parameter that I want to map it to FPGA or I want to map it to ASIC and it will generate the RTL according to that right.

Even at the FPGA level you can have different vendors right you can have Xilinx we can have Intel Altera and so on even for those FPGAs have different components. So, you even you can actually target for specific target boards right. So, this is something that gives you the portability right.

From the same specification which is written in high-level C++, you can actually make it to different targets right; that is the big biggest benefit of high-level synthesis. And then the permutability which is basically the design of space exploration.

So, from the same specification which is basically he is untimed there is no objective right whether it is something for power area or timing nothing it is just a specific its untimed C specification, and I can actually explore different kinds of design from the same specification.

(Refer Slide Time: 04:04)



Here is a very nice example that supposes I have a given very simple example of this right which is doing this the addition is basically in 1 plus in 2 plus in 2 in 3 plus in 4. Now, suppose I want to target it for reducing the latency right so; that means, I do not I want to do it very fast right and the corresponding data will be this, data path will be this because there are no intermediate registers. So, the latency will be 1 right, but it is the area is 3 adders right.

So, I can generate this RTL using high-level synthesis just specifying certain pragmas or certain parameters in the tool or suppose I want to optimize for an area. So, I can take these, and I can convert this here. So, I can see here that I can actually take one adder and I am storing the intermediate results and it is basically it will take 3 cycles to complete these 3 addition operations because I have one adder right.

So, the area will be one adder, but there will be two registers also. So, the area will be one adder two registers, but the clock will be very faster right. So, earlier it is 1 by t clock. Now, it is a 1 by 3 clock it is a three-time faster clock right because you are not the total delay of the combinational circuit is 3 adders here, here it is only one adder right.

So, that is the advantage but its area is less, but it is slow it will take 3 clocks right, and then if you want to improve the throughput; that means, you want to get your output fast you will get a pipeline circuit like this, where you just still you are adding 3 adders, but it is in pipeline mode so, that after one the pipeline stages are filled you can get output in every clock right.

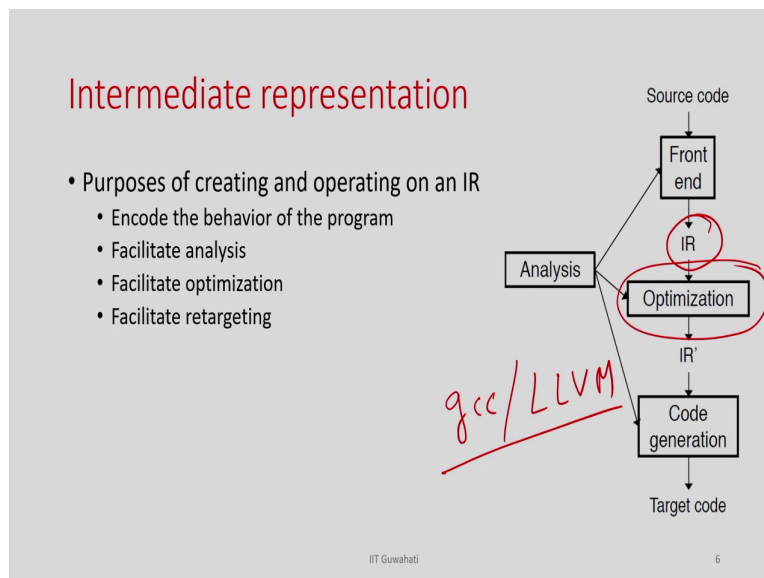
So, if you just suppose you want to do it for a streaming data right you are continuously giving the new set of data, you will get the output in every clock here, but here it is not like that right here you will get it, but the latency is 1, but here the delay the clock is very slow, but here the clock is still faster and you will get output in one clock.

So, the point he tries to make is that you can actually all get all these 3 kinds of data paths just by setting certain parameters in the high-level synthesis tool right. So, that is the kind of you can actually explore the design space using high-level synthesis easily right. So, that is the biggest advantage we have already talked about.

So, let us now move on to the high-level synthesis we have seen that high-level synthesis consists of several sub-steps right. It IS a complex transformation process. So, it goes through several sub-steps like pre-processing scheduling allocation binding and data path and controller generation, and then finally, you get the RTL right.

So, what I am going to do is that I am going to try to see what are the exact problem I am going to solve here in every sub-step and what are the things involved there right. So, I am going to discuss this in today's class.

(Refer Slide Time: 06:49)



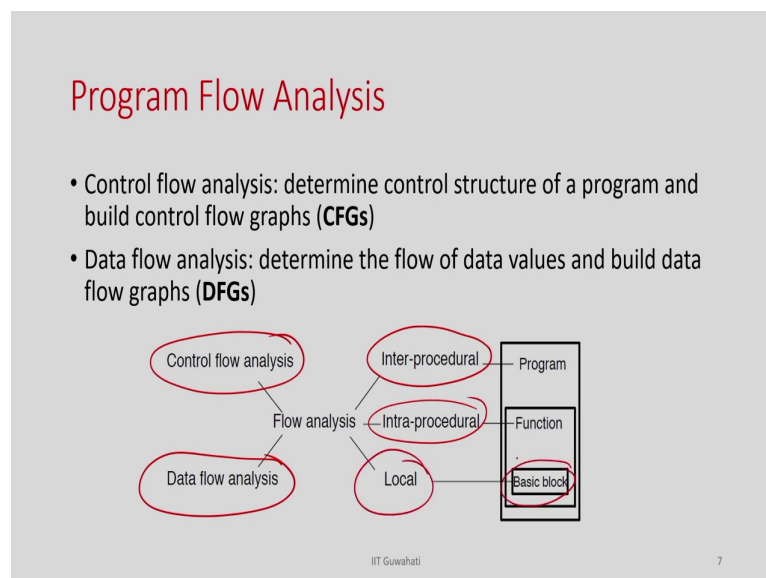
So, as I mentioned in the pre-processing step is something you have a C code, you convert into some intermediate representation. So, the task is like you actually identify you break the big expression into 3 address form, you identify the basic blocks, you identify the control

flow, you identify the data flow, and then you actually represent that that input CV we are in that IR form right so in terms of a control and data flow graph right this is what is happening here.

And if you look into this compiler because this is a C code you need a parser right you need a compiler that basically can parse a C code. So, and we have very common we already know that GCC LLVM. So, we can use any of such parsers because I do not have to write my own code it is something it is a C code, I can utilize the GCC or LLVM parser which actually parses this input C code.

It will just do the syntax analysis and semantic analysis and it give me the IR and then most importantly the IR level we can apply the various kind of optimizations which we will cover in some subsequent classes so that we can actually optimize certain things in the input code itself if there are some redundancies. And then finally, that IR is something we will represent in some form which you understood by the high-level synthesis tool ok.

(Refer Slide Time: 08:06)



As I mentioned there is some dataflow analysis is involved here. So, there is information like a control flow analysis to be done. So, what is the control flow we have to identify the back edge, we have to identify the loops for that we have to do the dominator analysis right? So, all those things are part of this compiler right, and also have to identify the data flow, and the dependency among the operations within basic blocks and across the basic blocks right.

You have to do this analysis for as I mentioned it is basically inter-procedural if there are multiple functions right you have to do it for the hierarchy of functions or within the function also right inter-procedural also.

Within the function what are the dependencies within the function you have some local dependencies which are called basic blocks right? So, we have already understood that. So, all analysis involved in the pre-processing step, and at the output level what we understood is that we actually get the basic blocks right.

(Refer Slide Time: 08:59)

### Basic Block

- **Basic block:** a sequence of consecutive intermediate language statements in which flow of control can only enter at the beginning and leave at the end.
- Identify Basic blocks
- Identify Control flow
- Usually C compilers like GCC or LLVM are integrated into HLS flow as front end

The diagram shows three basic blocks:

- B1:** (1) p = 0, (2) i = 1. It has a "Branch Target" label.
- B2:** (3) t1 = 4 \* i, (4) t2 = a[t1], (5) t3 = 4 \* i, (6) t4 = b[t3], (7) t5 = t2 \* t4, (8) t6 = p + t5, (9) p = t6, (10) t7 = i + 1, (11) i = t7, (12) if i <= 20 goto B3. It has a "Next Inst." label.
- B3:** (13) j = ...

Control flow is indicated by red arrows: from B1 to B2, from B2 to B3, and a branch from B2 back to B1. Handwritten red annotations include "DAG" and a large circle around the B2 block.

IIT Guwahati
8

So, we will represent our behaviour as a basis set of basic blocks and their control flow right. So, within the basic block what do we have? We have a sequence of 3 address operations where we have actually each operation consist of single operations and it is a there is no control flow right. It is this code that is going to execute at the top to the bottom sequentially right and this is the control flow right.

So, after that, you will go here and once say this particular condition does not hold you will go to this basic block and so on right. So, the pre-processing steps do all those things in the context of high-level synthesis since all the high-level synthesis tools just have integrated one of the existing C compilers say either GCC or LLVM.

So, we will expect that our input is already converted into those 3 addresses in terms of basic block representation ok. And in this particular course, I am not going to go into detail about this because this is a very conventional compiler you will get in any compiler course.

So, I will assume that you have certain knowledge about that and I will expect that whatever the input for my high-level synthesis is after the pre-processing right where you will get the code in this form and what I am going to do, I am going to take one basic block at a time and then I am going to do the scheduling and then we try to do the allocation binding where we try to identify the minimum number of registers and function units which can be shared across basic block also right that we will discuss in detail in subsequent classes ok.

So, now let us move on to the next step is something scheduled. So, we already know that scheduling is something where we assign a timestamp to an operation. So, what is my input? We have this input; we have this basic block information.

So, we have these operations we construct a kind of data flow graph DAG the data dependency graph. So, which basically represents the dependency of operations and how the operations are dependent on each other is given to me.

(Refer Slide Time: 10:59)

### Scheduling Problem Formulation

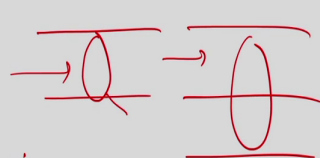
**Input:**

- Sequence Graph  $G = (V, E)$ ,  $|V| = n$
- Delay of each node.  $D = \{d_i, i = 0, 1, \dots, n\}$
- Resource or Timing Constraints (optional)

**Output:**

- The start time of each node  $T = \{t_i, i = 0, 1, 2, \dots, n\}$
- Latency: number of cycles to execute the entire schedule. Difference of start time of source node and sink node; latency =  $t_n - t_0$

The start time of an operation is at least as large as the start time of each of its direct predecessor plus its execution delay

$$t_i \geq t_j + d_j \quad \forall i, j : (v_j, v_i) \in E$$


IIT Guwahati

So, we will learn that it will be represented by a graph called sequence graph ok and then we have also the delay of the nodes. So, we have not discussed in a previous class that these

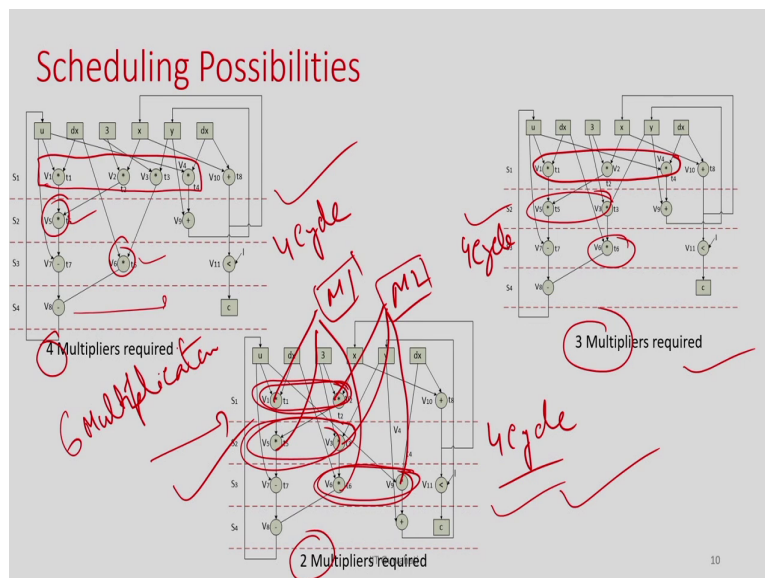
operations may not be always single-cycle some operations may be multiple cycles, some operations may be single cycles, and so, on. So, that information is also there.

So, a single cycle means, it will complete this execution in one cycle and if it is a two-cycle operation it will take two-cycle to complete the execution. So, now, that so, this is the called delay. So, for each node, I know the what is the delay value, and then also I have certain kinds of constraints. So, whether how many resources I can use or how much time I can utilize. So, that information also is there, and what scheduling does do? It is basically as I identify where the operations to start.

So, the start time of the operation  $t_i$  for node  $v_i$ . So, for each operation, it will say when this operation starts. So, if it is a single cycle, it will complete the execution within the same cycle if it is multi-cycle, it will start here, but it will complete in I mean after I mean. So, if it is a 2 cycle so it will take 2 cycles I mean 2 clocks, and all right.

So, this is something that schedule does. So, given this data dependency graph of a basic block and the delay information it just says when this operation will start right. So, this is what the stacks of the scheduling are, but what are the complexities here let us try to understand this ok. Say suppose I say I just give you a graph and some sequence graph or data dependency graph and say you schedule it in 4 cycles right. So, I just give you this.

(Refer Slide Time: 12:46)





So, here is an example. So, if you take that example that DIFU example you can actually schedule this behaviour in 4 cycles that we know right and now we will just see how many. So, I will just show you some possibilities ok. So, this is one of the schedules where you see the main idea is that your schedule does not violate the data dependency which I forgot to mention here.

So, you have to just find out the start time, but you make sure that the operation data dependency must satisfy right. So, this is something that must make it is not that we just assign timestamp, but you make sure this data dependency is not violated and whatever the timing or resource constraint given to us that is also not violated ok.

So, now given behaviour and say suppose I give you that you use only 4 cycles to schedule it right. So, you can have many scheduling possibilities. So, I have shown 3 scheduling possibilities here you see here. So, all are actually satisfying the data dependency, here all these data dependencies are satisfied.

So, that satisfying data dependency in the sense, if there is an operation, is dependent on the operation. So, the previous operation must be executed then only the dependent operation can execute right otherwise the data will not be available right. So, data dependency will be violated.

So, you see here I have 4 3 solutions here which all take 4 cycles ok, but here you see how many multipliers happening here? 4 right; that means, there are 4 multipliers that are going to execute in timestamp 1. So, that means I need at least 4 multipliers to do this here only 1 multiplier.

So, there are 6 multiplication operations, there are 6 multiplication operations in this behaviour and I identify because of this schedule since there are 4 operational schedules is 1 I need at least 4 multipliers because these 4 will run in parallel. So, what does it means? 6 multiplication operation need 4 multipliers ok.

And if you look into these 4 multipliers only one will be used here, one will be used here and none of them will be used here so; that means, in this particular clock 4 multipliers will be utilized and the rest of the time most of the multiplier will remain idle. So, this is the one solution.

So, remember these are the all-valid solution because it does not violate the data dependencies ok and now see this is another solution here there are 3 multipliers scheduled here, and still it is 4 cycles right this is also 4 cycles, this is also 4 cycles. So, here because 3 multiplier is in parallel here 2 and it is one here.

So, it will take at most 3 basically 3 multipliers to execute this right. So, both are 4 cycles, both have 6 multiplications, it needs 4 multipliers, this needs 3 multipliers. So, let me take another solution here. Here 2 multiplier is scheduled here, 2 multipliers are scheduled here, and 2 multipliers are scheduled here. So, I need two multipliers right this also takes 4 cycles and all 3 are the valid solution because it satisfies the scheduling constant.

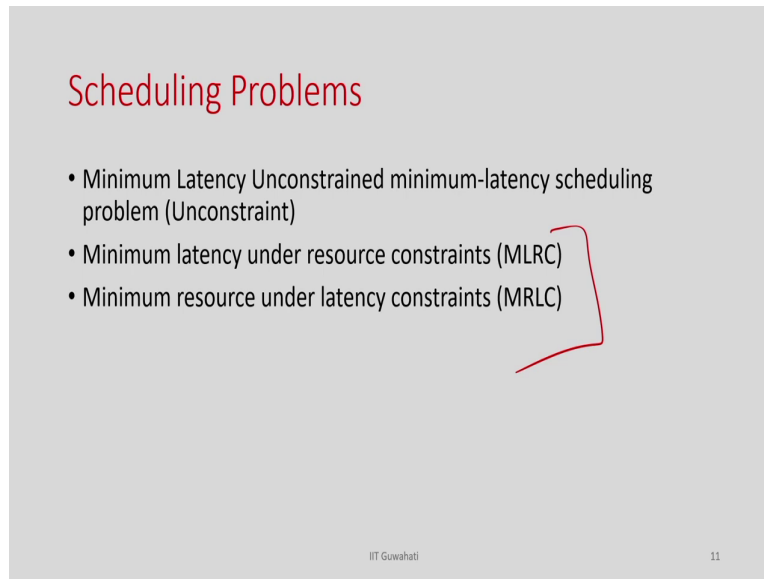
So, this gives you a very classic example that given a timestamp also you can have many scheduling possibilities, and non-necessary that all possible possibilities will give you the same resource right. I have a very nice example here that I have taken and behaviour where there are 6 multiplication operations, I show you 3 schedules, and all take 4 cycles 4 clock cycles, but this is the best solution in terms of resources because it can do it in 2 multipliers right.

So, basically, the point here is that in the scheduling given some even the time limit that time constant says 4, there are many solutions right you have to develop some algorithm that will actually even to identify this solution right this is something the better solution other than right. So, that is; that means, your scheduling problem is not only identifying a valid solution but an optimal solution right or a good solution right. So, that is something that is going to happen during scheduling.

I do not this example does not give you another angle of this problem right. So, which is basically here I told you these 4. So, I just mentioned to you that it is 4, but if I do not mention that right. So, if I say I told you that I give you 2 multipliers, you identify what is the minimum time required right so; that means, you might come maybe do it in 4 cycles, you may do it in 5 cycles, you may do it in 6 cycles, but you have to identify their solution that 4 is the best solution right.

So, given 2 multipliers you might have many solutions which might take 4 cycles, 5 cycles, 6 cycles, 7 cycles, or 8 cycles, but you have to identify that for given a certain resource bound, this is the best time I can obtain right. So, that is another angle of the problem.

(Refer Slide Time: 17:55)



**Scheduling Problems**

- Minimum Latency Unconstrained minimum-latency scheduling problem (Unconstraint)
- Minimum latency under resource constraints (MLRC)
- Minimum resource under latency constraints (MRLC)

IIT Guwahati 11

So, in that way, there are two kinds of problems we can look into in this scheduling which are called minimizing latency under resource constraint or minimizing resource under latency constraint. So, these are two conventional two dual problems because if you just try to use the minimum resources you have to number of clocks will increase right because their operation has to be scheduled in more time right. If your number of resources is less you need more time and if your resource is more, you need less time right.

So, these are the two conflicting requirements. So, usually in the scheduling, the two problems that we want to solve is that I tell you that this is my resource constant you identify the minimum latency you can achieve what is the best latency you can achieve using this resource right this is one kind of problem we are going to solve.

Now, another kind of problem is that I have given you a latency bound just like the example I have shown here that lets you schedule this in 4 cycles, but you identify the minimum resource. So, this is another angle of the problem. So, both are important problems and we are going to discuss the kind of efficient algorithm to do that right. So, that is about the scheduling.

So, let us move on to the next phase which is allocation and binding. And we have already understood high level that allocation means you have such a set of operations you have to map that operation into a function unit because in hardware your operation is executed using some function unit right.

So, the operation to be mapped to the function units and the variables of your program map to be registers or RAM memories right. So, you are going to map the memories. So, this problem is here you have a set of operations you can utilize the same number of resources which is ok.

See if there are 6 multiplier multiplication operations examples here you can always utilize 6 multiplier function unit multiplication units, but it is not good right. So, I can show you that in this particular example, I can actually utilize 2 multipliers because at most my requirement of max multiplication in every state is 2 utmost 2. So, I need two multipliers right.

And I can put this multiplier into multiplier 1, this into multiplier 2, this into multiplier 1, this into multiplier 1, this into multiplier 2, and this multiplier. So, this will not create any problem right this is what is the resource sharing which is very important in the context of hardware because I do not want to arbitrarily use a random number of hardware because that will increase my area, increase my power, increase my time anything it will all increase right. So, I have to always look for using a minimum number of resources right.

So, as I mentioned in this example perfectly that although there are 6 multiplication operations, 2 multiplier is sufficient and I can actually share one multiplication function unit for this 3-multiplier operation and I can use another multiplier function unit to share these 3 multiplication operations and that is what is important and that is what is being done in the allocation and binding phase.

So, from this schedule, I identify that 2 multiplier is sufficient. So, that is my allocation. So, I allocate 2 multipliers for this problem and then I bind these 3 operations this  $v_1 v_5$ , and  $v_6$  into multiplier 1. So, this is the binding of this operation to the multiplier and I am binding  $v_2$ ,  $v_6$ , and  $v_9$  into multiplier 2. So, that is the binding of the operations to the function unit.

So, we understand that this allocation and binding. So, you identify the minimum number of resources for a given schedule and you understand that for different schedule the resource your resource recommend is different I have already explained in that example right. So, you identify for a given schedule what is the minimum resource you needed and then you bind them right.

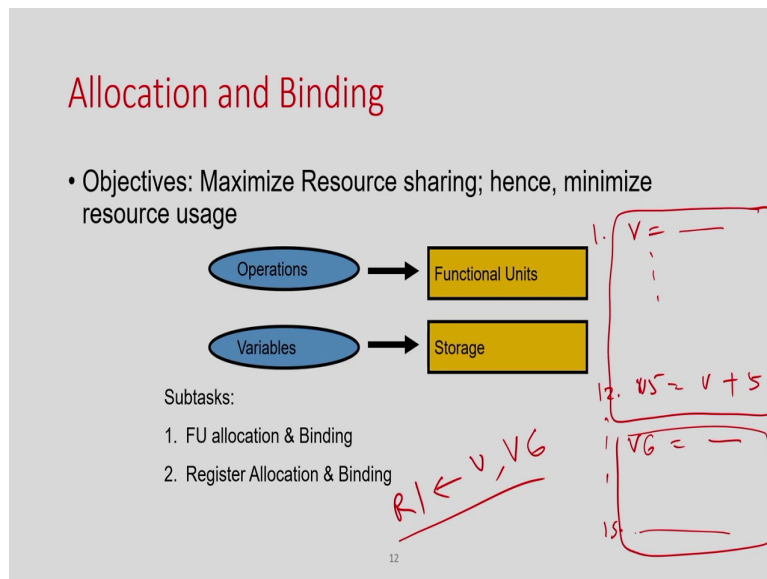
So, during bind you make sure that there is the operation that is running parallel cannot be bound into the same function unit because you can at a time, we can only execute one

operation in one function unit right. So, that is what is about this allocation and binding of operation to the functional units.

Similar procedure proceedings have to be done for variables to register to bind. So, you have many variables in your program and you want to map those variables into registers because in the hardware you do not have variables, you have to store them in some hardware unit sequential element which is registers and you have to do that.

Again if you have to say 10 variables you can always map them into 10 registers, but that is again not the good idea because I want to utilize the resource right, I do not want to overuse the resource and the basic idea here is that in C program whenever there is a variable, we never thought about their lifetime or when its getting used right. So, it is not that one variable is needed for the complete program.

(Refer Slide Time: 22:48)



So, you define a variable say here  $v$  equal to something and then say this is the last time where you are using say for  $v5 = v + 5$  equal to  $v$  plus something ok. So, this is saying this is in line number 1 and this is in line number 12 and say that complete program is 15 lines ok so; that means, I need to store the value of the variable during this time ok. So, I do not need to store this after the clock say 12 because after that the  $v$  value  $v$  never, we use right.

So, if there is another variable say  $v6$  is defined here and it is getting here. So, I can actually store this  $v6$  and  $v$  into a single register say  $R1$  ok this is the way I am going to identify how

many registers I need. So, if there are two variables that are not getting used at the same time or that means their life is not overlapping, I can put them into the same registers and that is how the shearing happens.

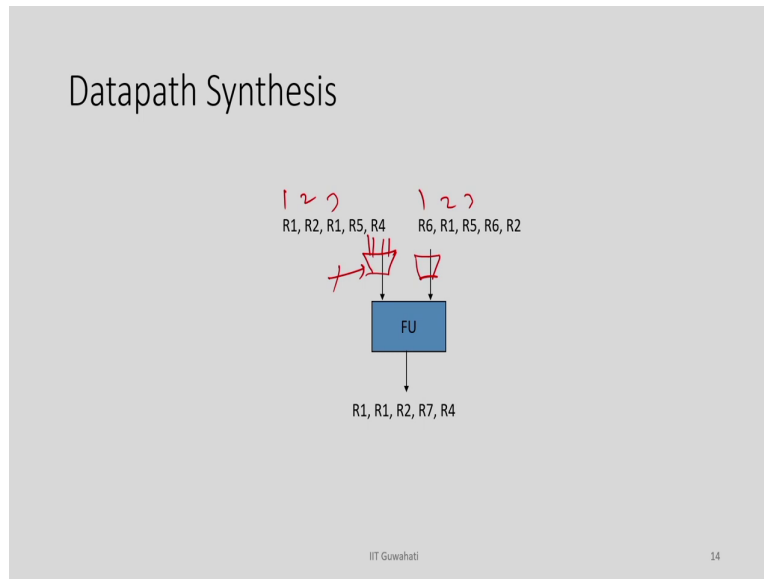
So, this is how you identify how many registers you need to be allocated for these variables and then you bind them this is what the binding I have done that  $v$  register  $v$  and  $v_6$  map to register 1. So, this is what register allocation and binding we will go into more detail here about how this buffering happens and what kind of algorithm to be done.

So, you have to always keep in remember that whatever I am talking about scheduling allocation binding is not a manual job right we need a software which going to do that. So, the most important part here is, how can I automate this process right. I identify this, but can I write a program that will automatically identify the lifetimes of the variables.

And then identify the variables which have a non-overlapping lifetime and map them into the same registers and you have to guarantee that this actually always results in the minimum number of registers right. You can have many such mapping, but can you identify what is the minimum number of registers needed right similarly for FUs.

So, these are the interesting problem that we are going to solve during this when this course I am going to talk about different algorithms that I am going to do these things ok. Once this allocation and binding happen, the next phase is kind of conventional, but important that you have to make the data path right.

(Refer Slide Time: 25:03)



You have you know the how many function units are there, how many registers are there what you have to do? You have to make connections and this connection is that because this FU is getting shared or it is basically performing multiple operations in multiple time steps their input may be different right.

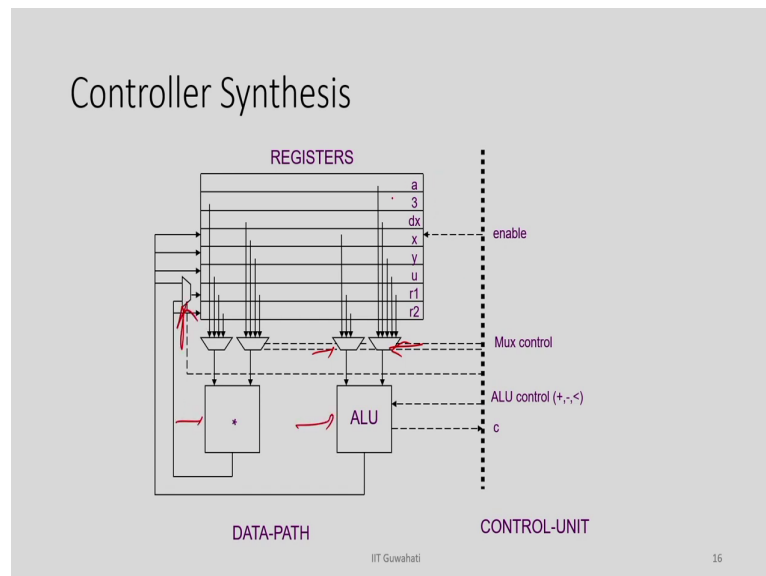
So, it may be that one FU's in the register. So, its input for this FUs is R 1 and R 6 in time step 1 and say R 2 and R 1 in timestamp 2 and R 1 and R 5 in timestamp 3 and so, on. So, these are the possible inputs coming to these registers to this function unit right. So, how to make this connection? You have a set of registers. So, you have to put some kind of multiplexer here so, that you can actually give these inputs and you can actually do a time-division multiplexing right.

It is not that all values will come in the same clock rather in every clock one of the values will come and I know which value will come in which clock accordingly I can give a control signal here which will make sure that that in the timestamp 1 only R 1. So, it comes into FU among all these inputs similarly in timestamp 1, R 6 only come into the FUs in timestamp 1 ok.

So, this is how the interconnection is made right. So, you have to identify for FU what are the possible values are coming to this and make you make connections and you also make sure you add proper control signals, and then in that particular control state you give that

particular value so, that R 1 should come right this is what is the data path and controller generation right.

(Refer Slide Time: 26:35)



So, the controller is once the data path connections are done. You identify what are the control signals you need for FUs in ALUs, for mux, and for register inputs. So, everywhere there are control signals and you have to take control of the data right. So, you have let us suppose 10 registers, but may not be all 10 registers get updated in a clock right. So, only say 3 of them get updated in a clock.

So, we have to make sure that in that particular state only in right enable for those three registers are 1 and for rest of the things are 0 otherwise you will store some garbage value will come into those registers right. Similarly, as I mentioned for the data path if there is a mux that has 4 inputs to FU you have to make sure that in clock 1 what are the values I should set so, that the corresponding correct register value will come into FU input right.

This is what is the controller synthesis that every state you identify what is the operation to be performed. And accordingly, you set the control values control signals to 1 or 0 properly so, that exactly the required operation going to be executed in that time and rest on rest things are not going to happen in that state right.

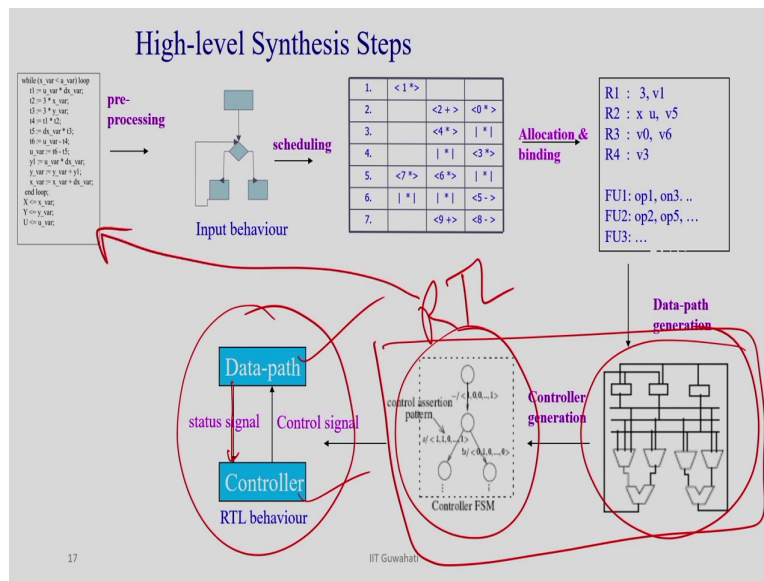
This is what is the data path and controller synthesis and this in this particular part what is important is how to identify these things automatically and what will be your interconnection



type, it can be bus base or mux base right. So, we will go into all detail, but this step you can consider is kind of more convenient because most of the decisions are already taken during the allocation and binding phase and scheduling phase.

So, this is where you have to make sure that whatever the decision has been taken in scheduling allocation and binding is going to happen properly in the hardware right. So, that is something is the interconnections ensured right that is going to happen in these steps ok.

(Refer Slide Time: 28:17)



So, with this, if you just follow all these steps finally, you will get an RTL that has a separable data path and a controller right. So, you have a data path it is going to execute the operations and you have a controller which is nothing, but an FSM finite state machine where every state assigns these 1 0 signals and gives that value to the data path as it is and this proper upper series going to execute in the data path right and in some state if there are some conditional operations happen.

So, the value of that conditional operation also returns back to the controller right. So, suppose you are doing a less than b. If a less than b then you are going to happen some operations if a less than b is false then you are going to do some operations right. So, that is information must come back to the controller so, that controller understands what operation to be done in the next state right.

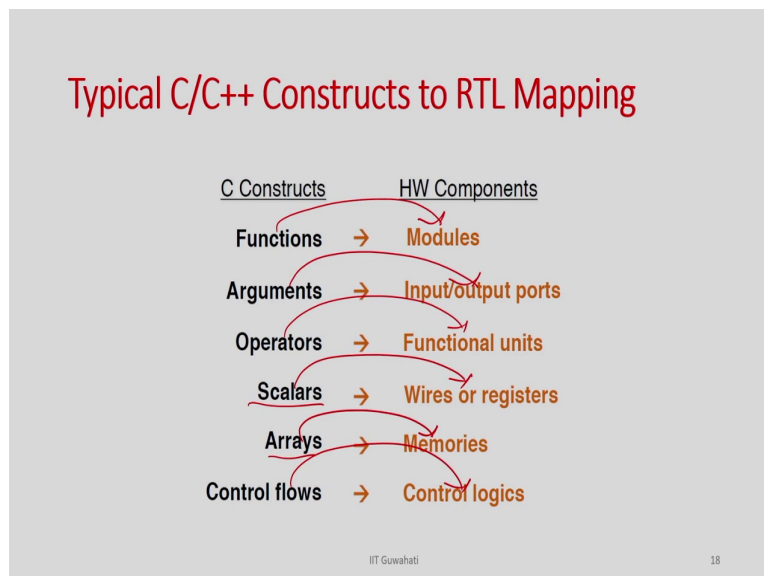
So, that is something that comes back to the controller. So, once you have done that. So, your data path will look like this, your controller will look like this and in combination, this will be the RTL which will execute the input behaviour that you have given to the high-level synthesis tool right.

This is what is the automation automated conversion process. And as I mentioned in this course, I am going to look into the algorithms that are going to do the scheduling the algorithms that are going to perform this allocation binding, and the operation or the kind of algorithm that can actually automatically synthesize the data path and the controller ok. So, that will be a major portion of the course and which I am going to follow from the next week.

But in today's class, I am going to give you some interesting facts that in the C, C++ there are different constructs like you have functions, you have operations, you have variables, you have arrays have a control flow for how those things are going to be mapped into RTL in general by high-level synthesis what is the corresponding things in the hardware right.

That is something I will just give you a brief idea and obviously, again I am going to call take each of them in detail and how efficiently they going to map to the hardware I am going to discuss, but here I am just giving you some high-level idea that the common construct of C what is the equivalent things in hardware.

(Refer to Slide Time: 30:31)

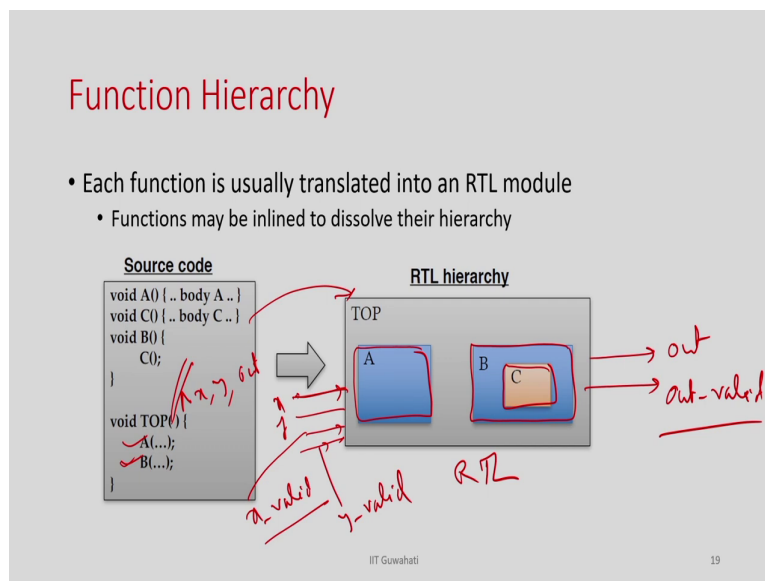


So, the first thing is the function. So, you just know that the function gets mapped to the module. I mean I am expecting that you have some familiarity with the Verilog or VHDL or HDL language if not please go through that otherwise you might face a little bit of struggle in the latter part of the course.

So, the function gets to the module in hardware, the hardware is the module is a kind of a component where you can perform specific tasks right. The argument of the function becomes the IO port for the module, and operations map to the functional unit that we have already discussed.

Scalar or the variables gets mapped to the wire or registers we will identify we will talk about which is wire and which is register, the arrays mapped into memories RAM and ROM and control flow is the control logic of the FSM ok this is the general mapping.

(Refer to Slide Time: 31:23)



So, I will just talk about them briefly before concluding today's class. So, as a function. So, function becomes a module right. So, you have the top-level function. So, the top-level function becomes the top module. So, this is the module that is the RTL I am going to generate, and whatever the input of this top-level function those will become the port of this right.

So, suppose this top has input say x and y and it is producing out. So, it will create x and y as input and it will produce out as the output. So, it will create two input ports for x and y and

one output port for y ok. Most importantly it actually generates the synchronization signal ok. So, it is an x valid and this is y valid right.

So, you have whenever this value you give valid equal to 1, then only whatever the data you are giving it is a valid data and I mean this RTL will take that right this will be useful for synchronization because if once you create this module and if you want to integrate into a bigger S o C, you need the synchronization.

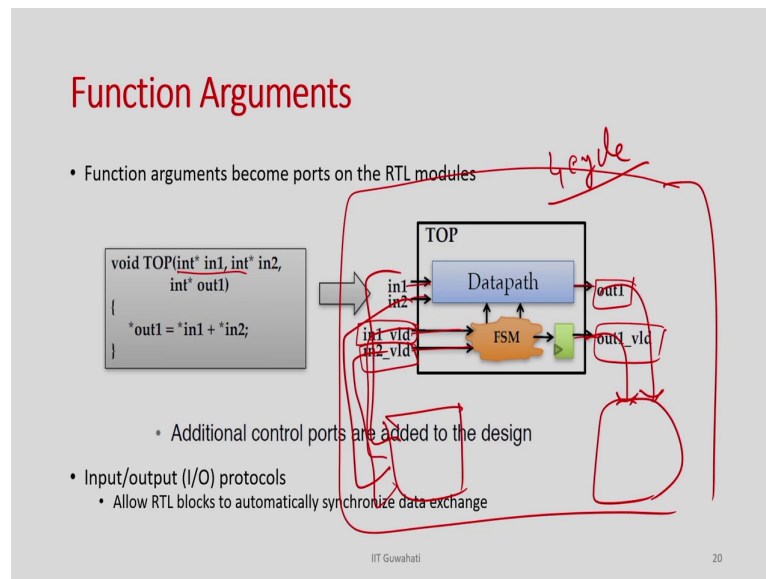
When this data is the valid data is coming and when this particular module is creating a valid output also because it may not give you always correct input right. So, if the latency of the circuit is say 4. So, every fourth cycle is going to give you a correct value right. So, this something has to the synchronization things has already maintained ok.

So, whenever there is a function, it creates a module. So, for example, in the top if you have a function called A and B, it will create a module A which will implement this function A, it will create another module it will implement B and then this function will be invoked by instantiation of the module in the top-level right.

See in the top-level I am going to instantiate this module A and module B and I am going to connect with the proper control signals so, that control signals and the value so, that operation A actually can perform the desired operations ok and it can actually maintain the hierarchy also right.

So, if this B again calls a function C, it will create a module and it will be instantiated within module B. So, in general, it will create hierarchical RTL modules. So, which will maintain the hierarchy structure of the original C code ok, but if sometimes the function is very small and it might be in line also ok. So, just to so, it might get inlined in between ok.

(Refer Slide Time: 33:44)



So, as I mentioned that the function arguments. So, the function argument becomes the port. So, here is the example that in 1 and in 2 become the port and out is the output port, but it also has the valid signal so, that this whole process can be integrated into a bigger system right.

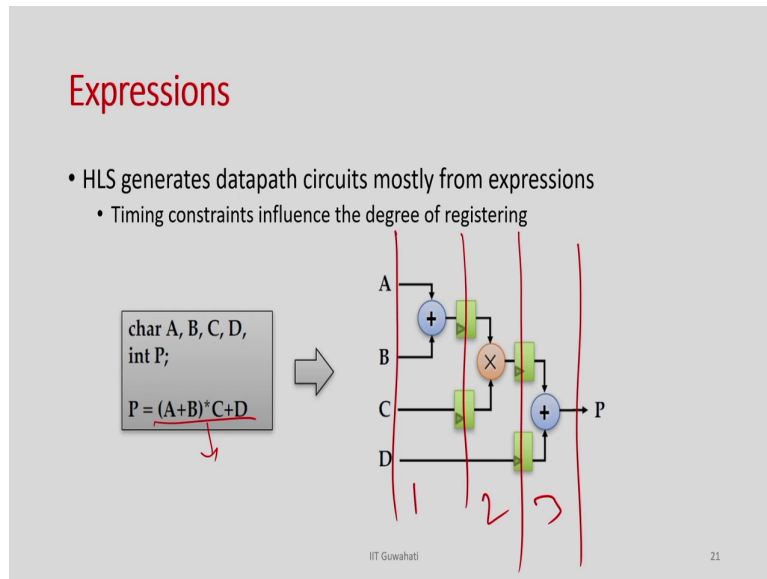
So, this particular valid signal is automatically generated by a high-level synthesis tool and just to give you the idea that if this particular take is taking say 4 cycles as I mentioned earlier also. So, it is not that you can give input in every clock right so, it will take input in every 4 clocks.

So, you have to so, whenever or it will wait for the valid signal whenever the valid signal on, then only it is going to take it and now if the valid is coming less than 4 cycles, it will ignore that input because it cannot take in the input once it is taken in next 4 cycles right.

So, this is what is the idea and whenever it produces a correct, I mean valid output it will just say this is the output is valid and you can integrate it into a bigger system which may have some different module that might take this as an input and say there may be some another module which might produce these inputs right.

So, this will actually be combined or you can actually integrate together so, that it will work perfectly right. So, that is what this argument becomes and usually, a high-level synthesis tool generates all these valid ports as well ok for each input.

(Refer Slide Time: 35:07)



And the expression I have already explained that it will become it will map to this function units and usually the expression gets split into 3 address form, that I have already discussed and then it gets scheduled right. So, it is not that the whole expression is going to execute in one clock based on the target clock period and you might try to execute this in multiple cycles that I had already I have discussed.

So, here is the example that you have. So, this is the first clock this is my clock 1, this is my clock 2 and this is clock 3. So, this operation is going to execute in 3 clocks and the intermediate data will be stored in registers ok.

This is a pipeline implementation, but you can have different implementations for that ok. So, the expressions get into the function unit and how much time it will take to execute an expression depends on the target clocks and the resource availability ok.

(Refer Slide Time: 36:00)

## Arrays

- By default, an array in C code is typically implemented by a memory block in the RTL
  - - Read & write array -> RAM; Constant array -> ROM

- An array can be partitioned and map to multiple RAMs
- Multiples arrays can be merged and map to one RAM
- An array can be partitioned into individual elements and map to registers

IIT Guwahati 22

An array is mapped into RAM and ROM if it is a read-write array it will map to RAM if it is a constant array; that means, only a read-only array is going to map to ROM ok. So, basically, this array maps to RAM, and the interesting is that that index becomes the address for this right for the RAM.

And one important thing to be noted in the hardware is that this array now once you map this into RAM has different other signals right. So, it has that this address in which address you are going to access, it is has some read enable signal whether you want to access this RAM or not and this is a write enable.

So, whenever you want to write something you have to give this write enable equal to 1 and you have to give this corresponding address and the value which 1 to write, and then the data will be (Refer Time: 36:50) get into RAM. And similarly, once you read something you give the address, you give the C equal to 1 write enable equal to 0 and the corresponding value will come into A out ok.

So, this is how this access is going to happen, but in the hardware, this RAM is not exactly like added. So, you can access any content at any time rather in the hardware this RAM will be accessed using ports right either it will be a single port or dual port. So, that creates a lot of bottlenecks and I will post a significant portion of this course time to talk about that that it is not that if you have to say 10 access to an array in an expression, you can do all this 10 access in the same clock because that does not support in hardware.

So, you have to come up with some strategies so, that you can actually perform that operation. So, either you do it in multiple clocks or you rewrite your code in such a way that this can be doing ok. So, we will talk about all these things in detail in this course.

(Refer to Slide Time: 37:51)

## Loops

- By default, loops are rolled
  - Each loop iteration corresponds to a “sequence” of states
  - This state sequence will be repeated multiple times based on the loop trip count

```
void TOP (...) {
...
for (i = 0; i < N; i++)
  b += a[i];
}
```

→

TOP

IIT Guwahati 23

So, loops you in by default it is a rolled implementation that you have this loop. So, it will create a one body of the loop and it will get iterated for n times right. So, this is called iterative implementation or rolled implementation. If one iteration take say 4 cycle and it says n times loop. So, it is it will take 4 into N clocks to complete that implementation ok.

(Refer Slide Time: 38:12)

## Loop Unrolling

- Loop unrolling to expose higher parallelism and achieve shorter latency
- Pros
  - Decrease loop overhead
  - Increase parallelism for scheduling
- Cons
  - Increase operation count, which
  - may negatively impact area, power, and timing

```
for (int i = 0; i < N; i++)
  A[i] = C[i] + D[i];
```

↓

```
A[0] = C[0] + D[0];
A[1] = C[1] + D[1];
A[2] = C[2] + D[2];
.....
```

IIT Guwahati 24



So, other than that loop can get unrolled. So, we basically unroll the loop and you remove the loop and you make it like this and you execute in the hardware or you can actually do a pipelining.

(Refer to Slide Time: 38:21)

## Loop Pipelining

- Loop pipelining is one of the most important optimizations for high-level synthesis
  - Allows a new iteration to begin processing before the previous iteration is complete
  - Key metric: **Initiation Interval (II)** in # cycles

ld - Load  
st - Store

for ( $i = 0; i < N; ++i$ )  
 $p[i] = x[i] * y[i];$

II = 1

cycles

$O(n)$

IIT Guwahati 25

And the pipelining is one of the most important optimizations in a high-level synthesis that you basically assume that you have an implementation of the loop and then you actually run different iterations of the loop in parallel right.

So, basically, you add some pipeline stages and then you actually start this iteration 0 here, and the next time you start iteration 1 and you have pipeline stages in between. So, basically different parts of the stage of the pipeline perform the operations for different sets of inputs different iterations right.

This is what the loop pipelining is very important optimization and as a result, you can actually execute the looping order of  $n$  times if the number of iterations of the loop is  $n$  ok which is very interesting and I am going to talk about in detail in the in this course, but in general, this loop has loop is very important because it takes a lot of time.

So, you have you must have an efficient implementation of the loops in the hardware right. So, again we are going to discuss this loop things in detail in a subsequent course ok. So, with this, I conclude today's class and from next week I am going to talk about the scheduling problems.

Thank you.