

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

**Module - 10**  
**Verification of High-level Synthesis**  
**Lecture - 34**  
**Phase-wise Verification of High-level Synthesis**

Welcome, everyone. In today's class, we are going to discuss Phase-wise Verification of High-Level Synthesis.

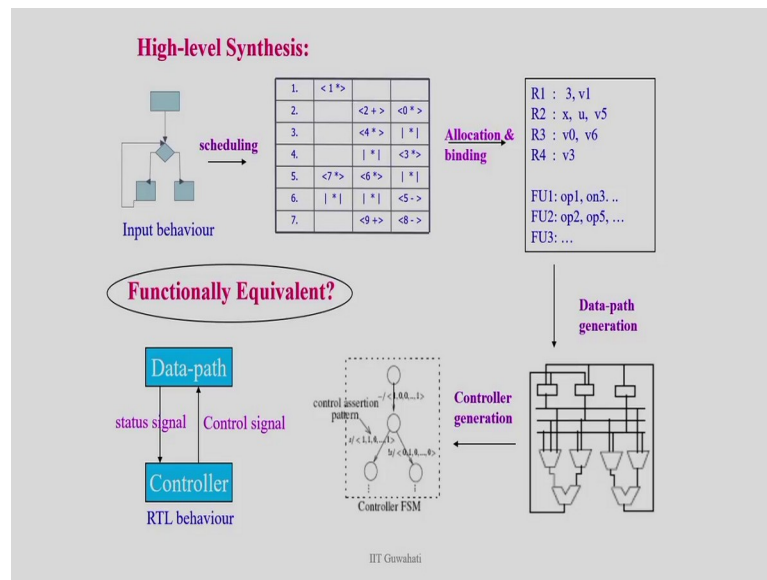
(Refer Slide Time: 00:57)

**References**

- R. Chouksey and C. Karfa, "Verification of Scheduling of Conditional Behaviors in High-level Synthesis," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, pp. 1-14, 2020.
- R. Chouksey, C. Karfa and P. Bhaduri, "Translation Validation of Code Motion Transformations Involving Loops," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 7, pp. 1378-1382, July 2019.
- K. Banerjee, C. Karfa, D. Sarkar and C. Mandal, "Verification of Code Motion Techniques Using Value Propagation," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 33, no. 8, pp. 1180-1193, Aug 2014.
- C. Karfa, D. Sarkar, C. Mandal and P. Kumar, "An Equivalence-Checking Method for Scheduling Verification in High-Level Synthesis," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 3, pp. 556-569, Mar 2008.
- C. Karfa, D. Sarkar, C. Mandal and C. Reade, "Hand-in-hand Verification of High-Level Synthesis," in Proceedings of the 17th ACM Great Lakes Symposium on VLSI, pp. 429-434, Mar 2007.

I am going to cover primarily our work in this area. So, although we have some recent works on this verification of high-level synthesis, I primarily will cover the basic theory that was covered in our earlier work, and then we will briefly talk about these recent works, what are the things, what is the research directions in the recent times, ok.

(Refer Slide Time: 01:15)



So, as we mentioned in the last two classes that high-level synthesis is a complex transformation phase, right, so which converts an input C code into an equivalent into an RTL code, right. And this high-level synthesis went through various processes like scheduling, allocation, binding, and data path control generations and we get finally, RTL.

Now, the point here is that the semantics, the way the C behaves, and the RTL behave are completely different, right. And since there are a lot of optimizations happening in every step and there is always a chance that this high-level synthesis tool introduces some bug in the RTL and that bug may not be so easily revealed, right.

So, maybe if you run for some basic inputs, you will not able to reveal that bug. And there may be some corner cases which is not implemented properly in the tool, you might encounter that bug and as a result your generator RTL is not functionally equivalent to the input C code.

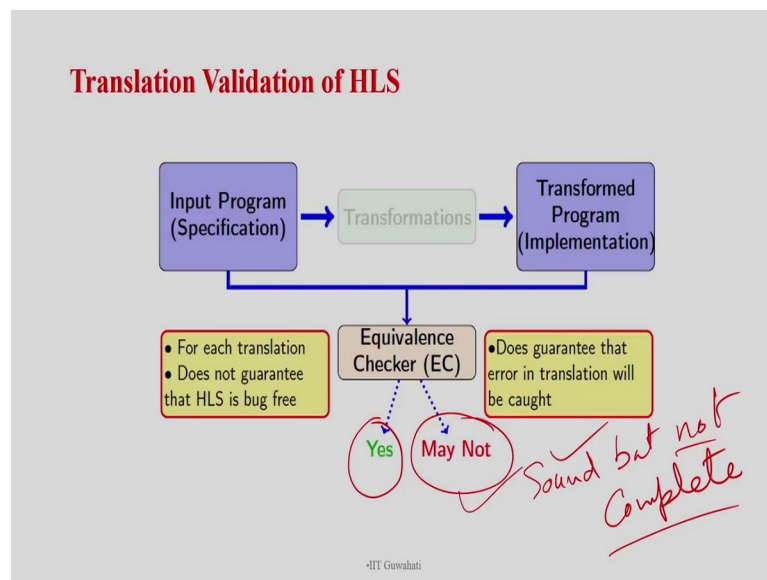
So, therefore, verifying whether the high-level synthesis process is correct or not is very important. And in the last two classes, we primarily talked about that how we can verify this high-level synthesis process using simulation, right. So, in the first class, we have seen how we write a C test bench, and then how the HLS tool automatically adopts them for RTL verification, right.

And then in the next class, we have shown that this RTL simulation is very slow. So, we present our recent work on this RTL to C reverse engineering and we have shown that using that technique we can actually verify or simulate the RTL very fast using a C simulator and that actually gives you performance benefit.

But doing all those things, whatever you do finally, what do you guarantee? You guarantee that the output is correct for the number of test case runs, right. But the test case number of test case runs cannot be exhaustive. Because if there are say 2 inputs, let us say there are 2 inputs of 32 bits, so you would there are possible inputs say  $2^{32} * 2$ ,  $2^{33}$ , right. So, there are so many inputs are there.

And it is very difficult to run that particular behavior for all possible inputs, right. So, that is why this simulation phase verification cannot give you a 100 percent guarantee that this (Refer Time: 03:40) is correct, right. So, the alternate approach that I have already talked about is basically formal verification, and in formal verification, the primary technique that is applied for high-level synthesis is the translation validation.

(Refer Slide Time: 03:47)



So, what translation validation does? It does not prove the correctness of the tool for all possible cases. But it says that for the given input, I have given an input program and it generate a RTL, and it formally proves that for this input, and this corresponding out which functionally equivalent, right. And once it is proved that, it is guarantee that for all possible input, these two behaviors are equivalent.

The example that I have given there are 2 inputs of 32 bits, if I just formally prove that these 2 inputs this RTL and the C V is equivalent then it does not matter how many test case you run or not, they are actually is always equivalent, right. So, that is the benefit. So, you can actually see the translation validation is also doing the kind of doing a kind of whatever we actually achieved by simulation-based verification.

In simulation-based verification also we cannot guarantee the correctness of the complete tool, right. What we just say, for this input we generate the RTL and these two are equivalent, right. They are actually functionally correct. So, this translation is equivalent, right. This translation process is correct. So, that is what the simulation process is also achieved and even the translation validation process is also achieved.

It does not guarantee that this tool for all possible input, will produce correct output, right. It just says that for this input this tool produces the correct output, right. That is the difference. So, one more very important point in this context of translation validation or formal verifications, here you have actually effectively compared two programs, right. And once you compare two programs that problem is actually undecidable, right.

So, undecidable in the sense, that you cannot have a method that is basically complete, right. Complete in the sense, that whenever I give two programs, it will say yes or no. Yes, means they are equivalent, no means they are not equivalent. But, the method that we can generate it can be sound, right. So, the method can be sound, but not complete.

What does it indicate? A sound means whenever I say this program and the output program are equivalent that definitely equivalent. So, I can actually strongly say yes. I do not have any false positive cases. In the sense, that I tell that these two methods are actually equivalent, but we can actually have some input for these two programs might generate different outputs so that never happens.

So, the formal methods are always sound, but not complete because the problem inherently is undecidable. So, we cannot have a complete method. I cannot have all scenarios where I can say yes or no. So, that is why these methods usually the equivalence checkers say yes and may not.

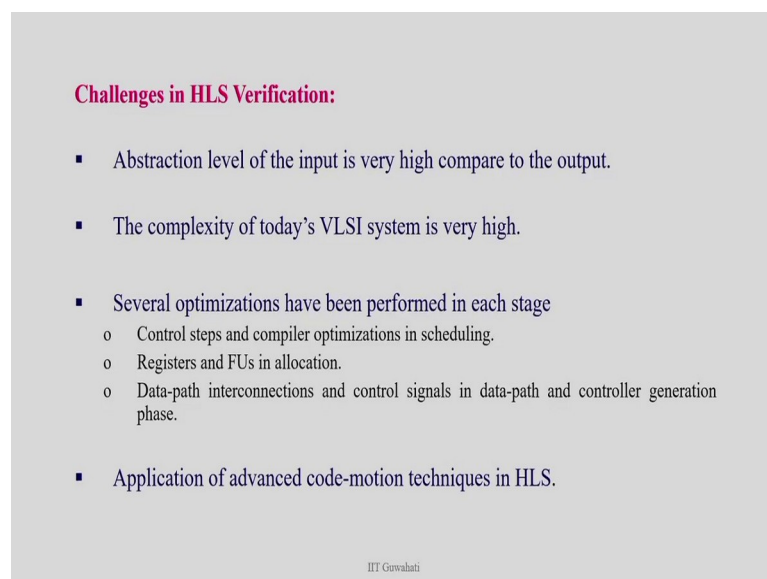
So, whenever say he cannot prove the equivalence, it says it may not be equivalent. So, there may be some scenario, where it actually says I know whether they are equivalent or

not, but they can be equivalent, right. So, that is why it may not, it is not, right. So, it may not.

And, but on the other hand, it is always sound in the sense that if it says that these two programs are equivalent, so they are indeed equivalent. So, you do not have any counter-example for that, right. So, all these translation validation methods is basically sound, but not complete, ok. So, in this work in this particular presentation, I am going to talk about how we can actually have an equivalence checker for high-level synthesis, ok.

And specifically in this particular class, I mean in this particular discussion I want to emphasize that since the high-level synthesis process is a very complex process, we cannot have an end to an equivalence checker easily, right. We will, in the next class, we will talk about those methods and their limitations, but in general the current research practice is basically verifying the phases of high-level synthesis, right. So, that I am going to talk about in today's class.

(Refer Slide Time: 07:41)



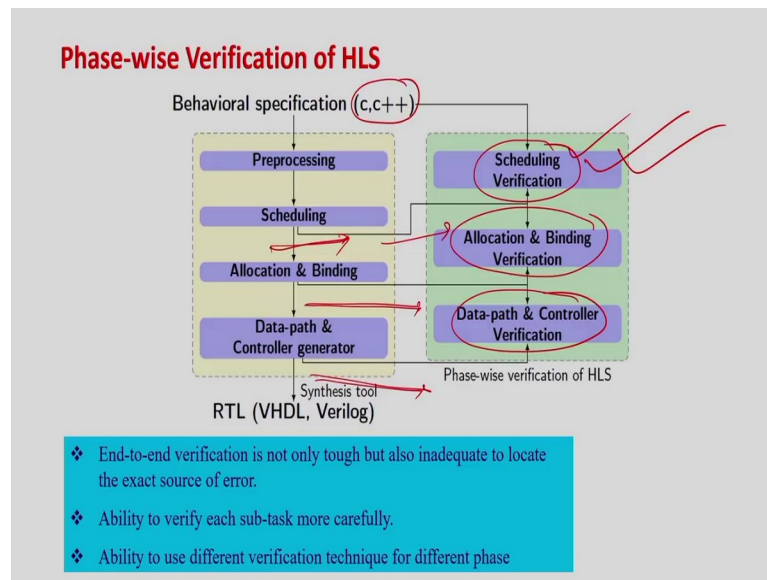
**Challenges in HLS Verification:**

- Abstraction level of the input is very high compare to the output.
- The complexity of today's VLSI system is very high.
- Several optimizations have been performed in each stage
  - Control steps and compiler optimizations in scheduling.
  - Registers and FUs in allocation.
  - Data-path interconnections and control signals in data-path and controller generation phase.
- Application of advanced code-motion techniques in HLS.

IIT Guwahati

So, as I mentioned just line this high-level synthesis is a very complex process. So, end-to-end verification is very difficult because you can understand that C and RTL do not execute in a similar manner, right. So, showing their formal equivalence, their execution semantics is not, it is difficult. It is not saying impossible, it is a difficult task. So, the thought it is actually now basically has a phase-wise verification, right.

(Refer Slide Time: 08:05)



So, you can actually assume that the high-level synthesis processes have these steps, preprocessing, scheduling, allocation binding, and data-path control generation phase. So, usually, in the current, if you see in the literature, most of the time it is actually verified in 3 phases.

The first phase is the scheduling verifications, where we actually take the actual input and the output after scheduling, right, and we verify, whether they are equivalent or not, right. So, in this preprocessing, in the scheduling phase, it is not only the ascending time step we just say how it is getting scheduled, but also the various optimizations compiler optimizations get applied. So, we verify both.

In the next phase, we take the schedule output and the output after register allocation and binding, and functional allocation and binding, and we compare them and that is called allocation and binding verification. And in the next phase, we take the output after this allocation binding, and the final RTL, and we just verify that, and we just verify whether the data path and controller generation phase is correct or not, right. So, this is how the phase-wise verification happens.

And we will see in the literature, and if you see in the problem is once this scheduling is done after that the behavior control structure does not modify, right. So, whatever the scheduling decide after that allocation and binding just map those variable to register,

operations to function register, and so on, and the data path generates the corresponding data path, but the control structure remains the same.

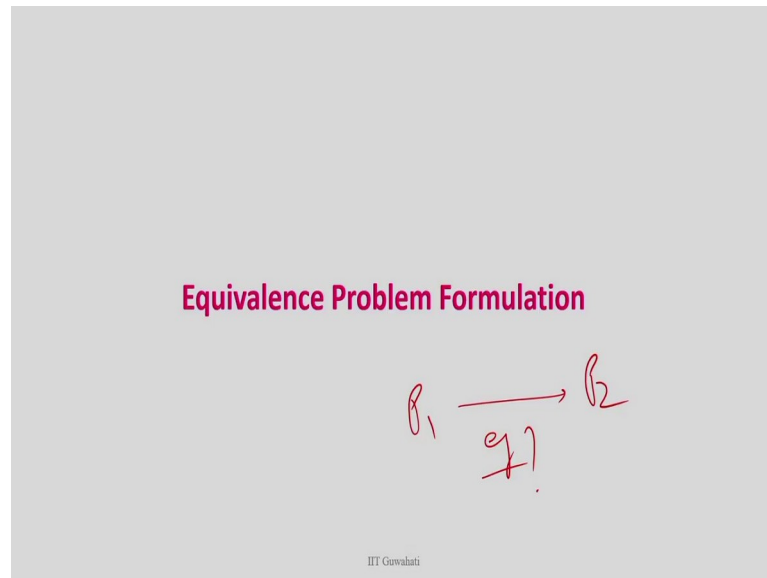
So, in general, on the other hand, in the scheduling phase, we apply various compiler optimizations, right. We apply code motion, we apply constant propagation, loop transformation, and so many things and the control structure can get modified during scheduling. So, as a result, the most challenging part of these 3 verifications is the scheduling phase. So, if you search in the literature if you find that 90 percent of work is actually on the scheduling verification because that part is most challenging compared to the other part.

So, today, what I am going to talk about. So, how we can verify first I will talk about how we can check the equivalence of the two programs, right. I will try to formulate that particular problem first and then I will talk about; so, this is the generic program formulation because I have two programs, how we can check the equivalence?

And then I am going to talk about how can I apply or adapt that basic the generic equivalence checking method for these 3 phases, right; what are the challenges are there and what are the things we can do to tackle those challenges, right. This is how I am going to go ahead.

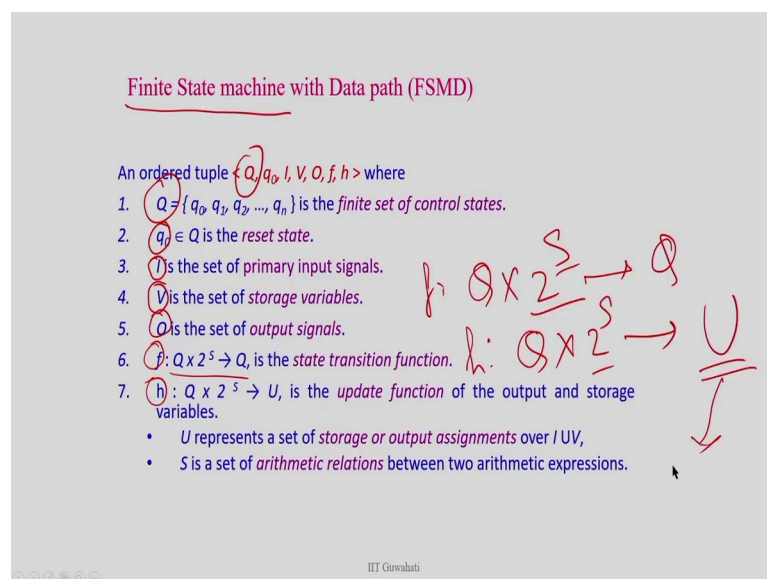
So, if you look into this it is a C program, and the scheduling is also kind of you can think of this is actually a C program, only it is a time C. We know in each time step what are the operations happening and the actual input is untimed. But they are actually still behavioral codes. There is no RTL, there is no hardware, right. So, this is still the same behavior. So, we can actually assume these are the two programs, ok. So, there is a behavior high-level behavioral program, ok.

(Refer Slide Time: 11:02)



So, now, you have given two programs, right. So, what is your problem? You have two programs, program 1, and you transform to program 2, and you want to check equivalence, right that is your problem. So, for any formal methods, we need to model the program, right. So, the first problem is how to model this, because unless you model it formally you cannot check the properties or whatever the way you want to check the equivalence, for that using the mathematical or formal representation, right.

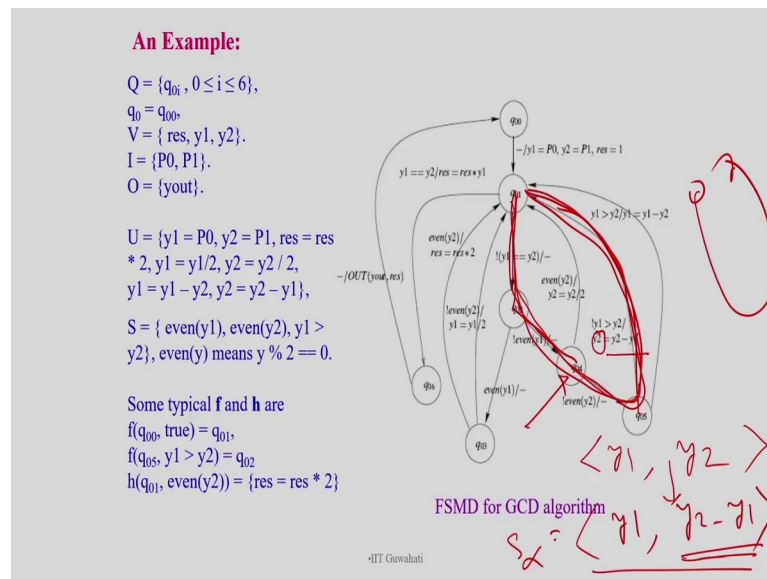
(Refer Slide Time: 11:31)





So, the way it is commonly modeled, the programs, for verification of a scheduling is finite state machine with data path. So, you all are aware of the finite state machine it is basically FSM, and here we have the data path integrated into the control structure. So, FSM we can think of it is basically a control structure, and the data path is the data flow or the data or data flow of the program. So, this is a very convenient way and it is very widely used model to model a program, right.

(Refer Slide Time: 12:05)



So, let us try to understand before going to the formal definition what is this FSMD. So, if you have a program, right. So, how we can represent it? This is a FSM representation with GCD algorithm, ok. So, how it is happening? So, it has set of states, right, and among states there are transitions, and interestingly within the transition we have, so this is a state, this is a state, we have a transition here, and in the transition, we have a condition first.

We have a condition, and slash the operations, right. So, this is how we represent the FSMD. Say, it is a set of states, a state of transitions, each transition has a condition and some set of operations that is the data path, right. And if there is no condition, if the condition will be basically true, right. So, by default, if there is no condition associate, it is basically true, always true condition, right.

So, now if you go to the formal definition, so the FSMD has a set of states Q. There is a start state, obviously, we will start from the actual state, obviously, any program has a set

of inputs. So, I have  $I$  is the set of inputs. We have some temporary variable in the program which is the  $V$ , set of storage variables. We have set of outputs  $O$  and we have a transition functions and the operations are associated with the transitions.

So, if you look at the high-level this transition is basically from a state with some condition,  $S$  is some says are set of possible conditions. So, we can take any combination of these conditions, and you go to the next state, right. This is my transition function. And the update function is basically any state under the same condition because it is the same condition  $I$  am happening, we will do some update operation. The update is basically arithmetic expressions, right. So, that is all.

So, we can easily model a given C program using this model we can understand because it is just nothing but the control flow. And whenever there is a back edge you can understand there will be some loops happening here, right. So, this means there is some loops happening here, right. So, we can understand that things. So, that is all. This is how we represent the programs. So, this is our model.

So, once we have the model, we define certain terminology on this model program model. And the first thing that we have to define a path, right. So, what is the path? The path is a sequence of states, right. The one important characteristics of the path that all the nodes of the path are basically distinct. Only the last node can repeat as the first node or any nodes, right.

For example, say suppose this is a path definitely, this is a path. This is also a path. So, only the last node of this path is repeating, it can be one of that node. But you cannot have a this is not a path. So, if you have a 3 times, this is not a path, right. So, path is something it a distinct sequence of state. So, at the last state can repeat. That is all, ok.

So, once you define a path, what we have to define? So, the how do we characterize the path, right, what are the information there in the path, and that is the characterization of the path. And if you consider a path any path say this is my path, so there are two things is there are set of conditions are there and some operation is happening, right. So, the way I am going to model it, I am not going to keep the operations.

What I am going to see? I am going to see what is the condition of this path, right. What should be the condition that should satisfy at the start of this start state then only the path

should execute, and if I execute the path what will be the final value of the variables. So, I have set of variables and outputs, so what I am going to see? At the end of this path, what is the final value of the variables, right? So, for example, in this path there is no operation is happening, but say let us consider this path. So, this is a path, right.

So, let us say consider this path. So, in this path, the  $y_2$  is getting updated, right. So, the final value of  $y_2 = y_2 - y_1$ , right. So, basically, I can assume at the start of the symbolic value of  $y_1$  is  $y_1$  and  $y_2$  symbolic value is  $y_2$ . And once we execute this path, I can see only  $y_2$  is getting updated with  $y_2 = y_2 - y_1$ . So, I will say  $y_1$  remains  $y_1$  and  $y_2 = y_2 - y_1$ .

So, you can see here, I am not giving the value in terms of summing values, right. So, what is I do in the simulation? Here I am not running it, I am actually representing the expression symbolically. It is a symbolic value. So, this expression hold for any possible value of  $y_2$ ,  $y_1$ , right. So, that is why this formal verification does not check for individual value.

And if I say this is a path and there is another path in the transform behavior whose the value of  $y_2$  remains same in both path and in both cases it is a  $y_2 - y_1$ , it means that for all possible input these two paths the value of the  $y_2$  will be the same, right. I do not have to check for all possible values of  $y_2$  and  $y_1$  because this is the expression remains the same. So, here we actually check symbolically, right. That is the advantage.

(Refer Slide Time: 17:02)

**Path and Transformation along a Path:**

- **Path  $\alpha$  from  $q_i$  to  $q_j$ :**  $(q_i = q_1 \ c_1 \ q_2 \ c_2 \ \dots \ c_{n-1} \ q_n = q_j)$   
a finite transition sequence . where all  $q_i, 1 \leq i \leq n-1$ , are distinct , only  $q_n$  may be equal to  $q_1$ .
- **Condition of execution of a path  $\alpha$  ( $R_\alpha$ )** Logical expression over the set  $I \cup V$  such that if  $R_\alpha$  is satisfied by the data state at the beginning of  $\alpha$ , then  $\alpha$  is traversed.
- **Data transformation over a path  $\alpha$  ( $r_\alpha$ ):**  $r_\alpha = \langle S_\alpha, O_\alpha \rangle, S_\alpha = \langle e_j \rangle$ , an ordered tuple of algebraic expressions over variables in  $V$  s.t.  $e_j$  represents the value of variable  $v_j$  after execution of the path  $\alpha$ .  $O_\alpha$  is the output list of the path.

-IIT Guwahati

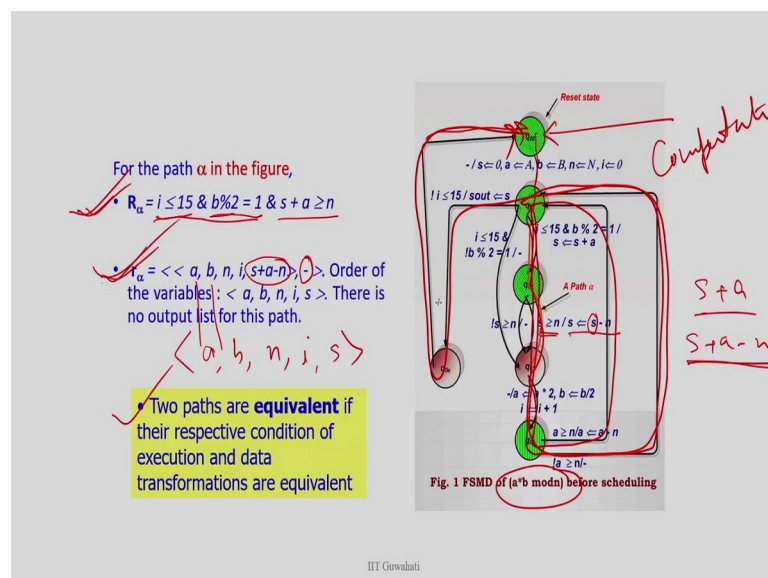
So, the way I am going to define a path, the path has two parameters. The condition of execution of the path and the data transformation of path. So, we represent this as a  $R_\alpha$ , right and this is a small  $r_\alpha$ . So, you can represent anyway. So, this is that terminology I am going to use it here.

So,  $\alpha$  is the path and this is the condition of the path. This is the transformation of the path. In the transformation there are two things are there, one is the  $S_\alpha$ .  $S_\alpha$  as I mentioned, the value of the it is a basically tuple of the variables and this represent the final value of the variables at the end of the path.

The example I have taken here is a tuple this is my  $S_\alpha$ , right. It has two variables  $y_1$  and  $y_2$ . So, if I traverse the path  $y_1$  remains  $y_1$  and  $y_2 = y_2 - y_1$ , right. This is my  $S_\alpha$ . And it might have some output. So, we will have to also specify the set of outputs as well, what are the outputs is happening in that particular path, right.

So, this is how I just represent. So, in a high-level no need to go into detail. So, for every path, I have a condition, and I have the final value of the variables once we traverse the path, that is a data transformation how the variables are get transformed, right. That is all.

(Refer Slide Time: 18:17)



Let us takes one simple example first. So, suppose this is the path I am going to consider, modularistic example. This is the mod  $n$  example, ok. So, here we can see here the what is

the condition. So, at this point the condition is  $(i \leq 15 \ \&\& \ b \% 2 = 1)$ , right. So, let me write it here. So, it is the I have written here  $(i \leq 15 \ \&\& \ b \% 2 = 1)$  right. This is the condition for this transition.

And then followed by this transition is happening and here it is  $(s \geq n)$ , right. But you see here that,  $s$  get modified here to  $s = s + a$ . At this point  $s$  is not  $s$  it is basically  $s + a$  whatever the value is updated. So, condition of this path will be the  $R_{\alpha} = (i \leq 15 \ \&\& \ b \% 2 = 1 \ \&\& \ s + a \geq n)$  because this  $s$  is not  $s$  because this  $(s = s + a)$ , right. This is the condition of this path.

And now, if you think about the transformation, say suppose the order of the variable is  $a, b, n, i$  and  $s$ , right. So, you have  $\langle a, b, n, i, s \rangle$ . So, that is the initial symbolic value. And we can see here  $s$  first modified to  $s + a$ , right. And then again it is  $s = s - n$ . So,  $s + a - n$ , right. That is the final value that I am going to get. So, that is what is  $s$  and the other variables remain as it is, right. So, they have done modified. So, that is the data transformations. And there is no output here. So, output is blank, right.

(Refer Slide Time: 19:50)

- **Computation of  $R_{\alpha}$  and  $r_{\alpha}$ :** By forward or backward substitution method.

Backward substitution method : If a predicate  $c(y)$  is true after execution of  $y \leftarrow g(y)$ , then the predicate  $c(g(y))$  must have been true before the execution of the statement.

Forward substitution method based on symbolic simulation

Figure 2. A typical path, its condition of execution and its simple data transformation

So, basically, you can understand here actually I am actually doing a rewriting and that method is called forward rewriting, right. So, either you can go for forward substitution or backward substitution. But, the basic idea is very same that you start at the point and whenever say I want to get the final value of  $s$  become  $s + a$  and then I move here and I found  $s = s - n$ , but this  $s = s + a$ . So, I will replace this  $s$  by  $s + a$  and then I will get this

expression. So, I am going to rewrite this forward way or backward, both is possible. But this is how I am going to get the final expression of paths, conditions and data transformations.

And now if I have two paths, I want to check the equivalence because our final problem is to check the equivalence of two programs, right. So, how can I define the equivalence of two paths? If their conditions are equivalent and their data transformation is equivalent, right. So, if both the paths have the same condition, and all the variables are actually modified identically, I will say these two programs are equivalent, right. So, that is very simple. So, this is how I am going to declare the equivalence of paths, ok.

(Refer Slide Time: 20:48)

Computation on the FSMDs:

A finite walk from the reset state back to itself without having any intermediary occurrence of reset state.

- Essentially a computation is an execution path of the behaviour.
- Two computations  $c_1$  and  $c_2$  are equivalent if
$$R_{c_1} = R_{c_2} \ \& \ r_{c_1} = r_{c_2}.$$

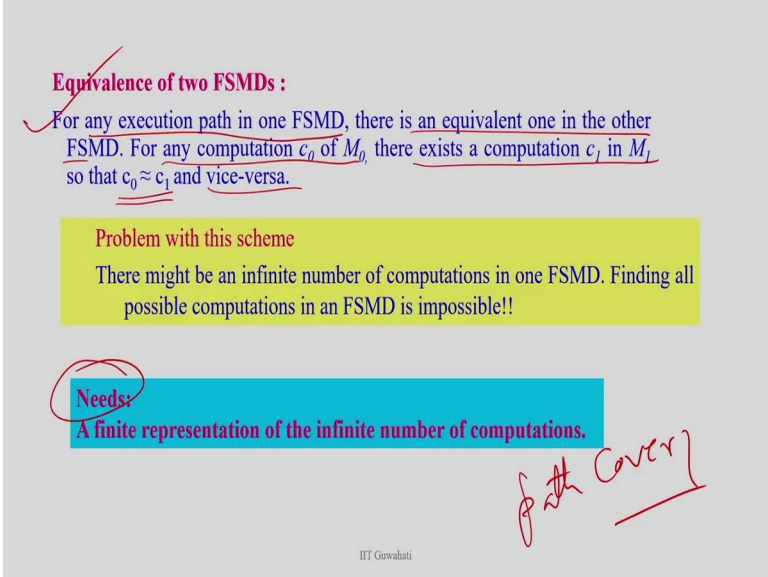
IIT Guwahati

So, next, I will define in the FSMD, the term called computations, right. So, once you give some input, suppose you have FSMD, right. So, if you let me go back to the same example. Suppose, here I am giving some value of the inputs, right, say here I give some inputs.

And once you give the inputs, what will happen? This program will execute, right. So, it will execute say this way, then it may be go this way, then it say come here. So, it will be some trace, right. So, then say suppose it come here this way and then finally, ends here, right. So, this is one execution trace, right.

If you give a input, your loop might execute 2 times or maybe 4 times, 100 times, whatever it is, and finally, you actually go to the n state, right. Here n state actually we always put it to the reset state. So, I will come back to the reset state. So, this is one computation, right. So, given one input whatever the execution trace of the behavior that is one computations, ok. So, this is what a computations.

(Refer Slide Time: 21:55)



**Equivalence of two FSMs :**

For any execution path in one FSM, there is an equivalent one in the other FSM. For any computation  $c_0$  of  $M_0$ , there exists a computation  $c_1$  in  $M_1$  so that  $c_0 \approx c_1$  and vice-versa.

**Problem with this scheme**

There might be an infinite number of computations in one FSM. Finding all possible computations in an FSM is impossible!!

**Needs:**  
A finite representation of the infinite number of computations.

*path cover*

IIT Guwahati

So, how we can now define the equivalence of two FSMs? Because finally, I have two FSMs; that means, I have program 1, I model it as a FSM 1 and I have program 2, I model it as a FSM 2. And I want to check equivalence of these two model, right. So, now, we should define the equivalence of two FSMs.

How we can define the equivalence of two FSMs? It is pretty obvious that how do we check the two programs are equivalent. It is basically for all input, all possible input both programs should give the same output, right. So, if you give input 1 to program 1 and program 2, you will get output 1 and output 2, and this output 1 must match with output 2, right. So, this is for input 1.

Then, you give input 2 to both the program you check the outputs, you check if they are equivalent, and you give the input 3 input 4. That is all, right. So, the equivalence definition cannot change. So, by definition for all possible input both the programs produce the same output, right. So, that is the definition.

And I say that for one input whatever the execution trace is basically is a computation, right. Because now I have to represent that whole thing, input-output characterization in terms of the FSM, the program models, right. So, as I mentioned that for a input whatever the execution trace is the computations. So, I can say now this whole the actual equivalence definition in this way that for all computation of program 1 or FSM 1, I have equivalent computations in the program 2, right and vice-versa.

That means it is basically represent the same thing because for one input execution trace is the computation, and I say that for all input both program produce same output. So, I am saying that for a computations, for all computation of program 1, we have equivalent computation in other program, right. So, this is how I can define it, right. So, that is what is the definition of a equivalence of two FSMs, right.

So, for any execution path which is basically the computations there is an equivalent one in the other FSM. In other way, for any computations in FSM M 0 there exist a equivalent computations c 1 and I define the computation equivalence this way and vice-versa, right. So, that is how I can define the equivalence of two FSMs. This is very obvious because it is nothing but saying that for all possible input they are producing the same output, right.

Now, what is the problem with this? The main problem here is that your have loops in your program, and the problem is loops, very simple. And this is because of the loops and the loops bound may not be static, right you might have a program, where for( i = 0 to n), and n is an input to the program, and n can vary, n can be an integer, so you can have 32000 value, 3674. And it may be long, long int and there may be n number of loops there.

So, the number of computations is very large, it may be infinite also, but I can say it is very basically very large. So, identifying all the computations of a program and showing their equivalence in the other program is basically just saying that I am actually simulating the whole program for all possible inputs which is also not possible we have already discussed. So, this is also not a problem possibly, right.

So, I cannot have an equivalence checking method that actually says that I will identify all the computations of a program and then I will try to identify their equivalence in the



other program, right, because it is the number of computation is too large. Your program may not, it takes lot of time or may take an infinite number of time.

So, we need to show this. There is no other shortcut, right. You have to show that these two programs are actually equivalent only when if they are actually equivalent for all possible inputs, right. So, what is the solution? So, the need is whether I can represent this infinite number of or a large number of computations using some finite number of paths. That is the question, right.

So, can I have a finite representation of this large number of computations? So, that is the question. And the answer is yes. We have a term called path cover, ok. What is that? Let us try to understand. So, if we have this finite representation of this large number of computations I can actually take only this finite number of paths and their equivalence. That will satisfy my problem, right. Let us try to understand in more detail, right.

(Refer Slide Time: 26:30)

**Solutions:**

Based on **Floyd-Hoare's** method of program verification.

- Put cutpoints in FSM s.t. each loop is cut at least one cutpoint.
- Set of paths from one cutpoint to other without having any intermediate one is a path cover.

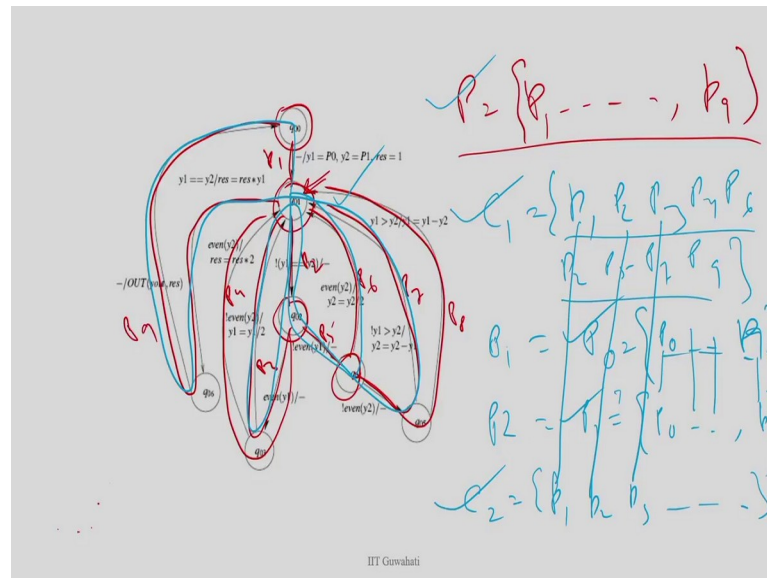
**Cutpoints:** Defined as follows in our work

- 1) Reset state.
- 2) State with more than one outward transitions.

The slide includes two hand-drawn state transition diagrams in blue ink. The left diagram shows a state with a self-loop and an outgoing transition. The right diagram shows a state with a self-loop and two outgoing transitions. A yellow box highlights the definition of cutpoints, with blue arrows pointing from the diagrams to the text. The text 'Floyd-Hoare's' is circled in red, and 'Cutpoints' is underlined in red. The footer '-IIT Guwahati' is visible at the bottom.

So, the solution is basically based on Floyd-Hoare's method of program verification, what it says is that you insert cut points or you define some of the state as cut points, ok.

(Refer Slide Time: 26:43)



So, what you can do? So, you can actually say this is the program I can decide, some of the nodes as cut points, say arbitrarily I decided these 3 are the cut points, right. But it is the Floyd-Hoare method of program verification says that you insert or designate enough number of nodes as cut points, such that all the loops of your program atleast cut by in 1 node point, right. So, that is very important. All loops must be cut by at least one cut point, ok.

So, if you can insert, so if I just have this program, say let us say I have all start state is always a cut point and I have say decided these 4 are the cut points. And I can see here there is a loop starting from this point, right. And there are many condition, which is at the loop and they are all coming back.

So, this is the loop point. So, the way I choose the cut points here say this cut point ensures that this loop is cut by at least one cut point, ok. So, once you do this theory says that set of paths between cut points without having any intermediate cut points is the path cover of the program, ok.

So, if I have these 4 cut points what are the paths are there? This is my path 1, this is my path 2, this is path 3, this is path 4, this is path 5, this is path 6, this is path 7, this is path 8 and this is path 9, ok. So, if I insert these 3, 4 cut points the number of paths in the path cover is 9 paths, right. So, suppose this is my p1, p2, this is p3, this is p4, this is p5, this

is p6, this is p7, this is p8, and this is p9, ok. So, this program has 9 paths. This is a finite representation, ok.

So, the Floyd-Hoare method says that you insert cut points in the program such that all loop must be cut by at least one cut point. So, the way I decided is if I decided 4 cut points and I identify there are a finite number of paths 9 number of paths are there in the path cover. But what it says the next is best important.

This is the definition of path cover is that it is a finite representation of all computations, right. What does it mean? That mean any computation of this program can be represented as a concatenation of path from this set, ok. Let us take an example where I want to take any computation, right. So, let us take say a computation starting from this node, ok, so which is going like this, say this is a computation.

So, I can represent this computation as p1, then I take p2, then I take p3, then I take p4, then p6, then I take again p2, then I take p5, then I take p7, then p9. So, this is a concatenation of these paths, right. So, this is one of the computations. And this computation I can represent using a concatenation of path from this and the concatenation consist of 1, 2, 3, 4, 5, 6, 7, 8, 9, 9 number of paths.

And I can we take any computations, I can understand now that I can actually represent that path as a concatenation of path from this set. So, that is the beauty of this path cover definition. So, it is now I have only 9 paths, but this program might have say 10000 computations, all 10000 computations can be represented as a concatenation of path from this 9 paths, using these 9 paths, that is a very important result, right.

And with this what we can do now, I can redefine my equivalence of definition the definition was that for every computation, I have an equivalent computation in another behavior. Now, I am going to say that for this program for every path in this path cover there is a path in the other program equivalent path in the other program, right.

So, I have program 1, say program 1 and the path cover is say p0. It has 9 paths say, right. So, p0 to p9. And I have program 2 where I have say p1 and it also consist of say p0 to p9, right. And I am saying that these two paths are equivalent, these two paths are equivalent, these two path are equivalent, and these two path are equivalent.

See if I show the equivalence of this path cover, right, so, given two programs if I show the equivalence of the path covers, what it will say? Then, these two programs are equivalent, right. So, I do not have to identify all computations and their equivalence, if I just identify the path covers and I can show that there are two path covers where the corresponding paths are equivalent.

And I can actually guarantee the equivalence of the program because even if I take these computations, right I can and say there is a corresponding computation in the other program is C2. And this C2 will be also represented as a concatenation of paths like similar like this p1, p2, p3, and so on. And since, these two paths are equivalent, these two paths are equivalent, I can actually show the equivalence of any two computations because these two computations are nothing but a concatenation of paths from these two sets.

(Refer Slide Time: 32:10)

Computation can be looked upon as concatenation of paths  
 $[\alpha_1 \alpha_2 \alpha_3 \dots \alpha_k]$  where  $\alpha_1$  emanates and  $\alpha_k$  terminates in the reset state of the FSM and  $\alpha_i$  terminates in the start node of  $\alpha_{i+1}$  for all  $i = 1$  to  $k-1$ .

**Finite Path Cover:**  
A set  $P = \{ p_0, p_1, p_2, \dots, p_k \}$  of FSM  $M$  is said to be a finite path cover of  $M$  iff any computation of  $M$  can be looked as a concatenation of paths from  $P$ .

**Theorem:** If there exist a finite path cover  $P_0 = \{ p_{00}, p_{01}, p_{02}, \dots, p_{0l} \}$  of  $M_0$ , for which there exists a set  $P_1 = \{ p_{10}, p_{11}, p_{12}, \dots, p_{1l} \}$  of  $M_1$  such that  $p_{0i} \cong p_{1i}$  for all  $0 \leq i \leq l$  and vice-versa.

IIT Guwahati

So, that gives a very important result here that now I do not have to identify all computations. What I am going to do? I am going to find a set of path cover of a program and then I will find that for, so for there exists a finite path cover of path in a program, I will find the path cover in the other program and I will just take the corresponding paths are equivalent. So, this theorem guarantees that these two programs are equivalent, right. So, this is something is important and this is how I am going to do, right.

(Refer Slide Time: 32:39)

**Outline of the basic equivalence checking method:**

Let  $M_0$  and  $M_1$  be the FSMs corresponding to the behaviours before and after any phase respectively.

**Steps:**

1. Find the path cover,  $P_0$ , in  $M_0$ .
2. For every path in  $P_0$ , find an equivalent path in  $M_1$ .
3. Repeat step 1 and 2 with FSMs interchanged.

IIT Guwahati

So, though I am going to do is now the program is not I am going to identify all computations. What I am going to do? I am going to find the path covers of program  $M_0$ , and then for every path in the path cover, I want to find an equivalent path in other program  $M_1$ . And so, this says that this  $M_0$  must contained in  $M_1$ . And then I will going to repeat the step by interchanging.

And now I am going to do for every path in FSMD 1 or  $M_1$ , I want to I have a equivalent path in other behavior which will actually say  $M_1$  is contained in  $M_0$ , and using these two I can say that  $M_0 \equiv M_1$ , right. So, that is what I do. So, that is how the overall basic equivalence checking works, right.

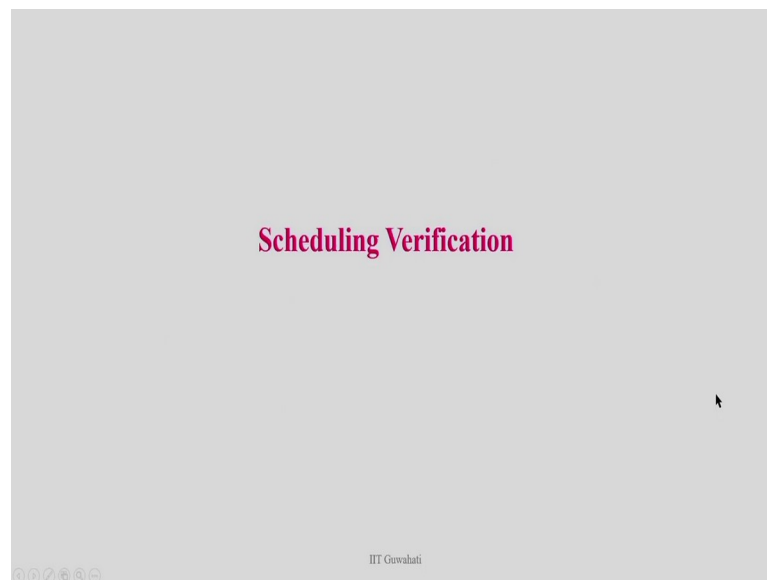
So, it is basically insert cut points in both programs, we identify the path covers, and we try to show the equivalence of paths from one for every path of this path cover there is an equivalent path there, and vice-versa if that happens then I can say these two be programs are equivalent, ok. So, the only question is that what are the cut points I should ensure, right.

So, as I mention that the requirement of the cut points is that it must cut all the loops, right. So, whenever there is a loop, right, so suppose there is a loop here. So, it is happening like this, right. So, I can see here that from there is a divergence of flow, right that this is a there will be a must two paths, right. One path goes inside the loop, another is going outside the loop.

So, if I consider any path where there is more than one outward transition, then this ensures that our loop will be must cut by at least one cut point, right. If you have a if-else then also there is a by divergence, but I take a superset. So, this particular way if I select the cut points that my start state or the reset state there is a cut point, and any state where there is an outward there are basically bifurcation of flow or divergence of flow.

That is a cut point and that will actually happen for if-else and loops. So, this set of cut points ensures that over loop must be cut by at least one cut point. And hence, the path between cut points will be a path that cover of that program, right. And I can actually use that path cover to show the equivalence of two programs, right. That is the basic theory, ok.

(Refer Slide Time: 34:55)



Now, what I am going to do is now I will move on, I will now talk about how I can use this basic theory to verify scheduling, ok.

(Refer Slide Time: 35:07)

**Scheduling Verification:**

**Challenges:**

1. In **path based scheduling**, consecutive path segments may be merged by the scheduler. Control structure of the input FSM is modified by the scheduler.
2. Several **code transformation techniques** like **speculation, reverse speculation, conditional speculation, early condition execution, common sub-expression elimination, renaming, dead code elimination, branch balancing, loop shifting** etc are used along with the scheduling to generate more optimized result.

IIT Guwahati

So, what I am going to see now, what are the challenges are there in scheduling modification, right. So, we know that several optimizations applied like code transformations, loop shifting, conditional speculation, many optimizations are applied, any compiler optimization is can applied and also we have seen that the path based scheduling, right. So, path base scheduling actually changes the control structure of the path, ok. That is very important.

(Refer Slide Time: 35:29)

**As a result,**

- Operations may be moved across basic block boundaries (speculation, reverse speculation etc).
- The control structure may be modified (conditional speculation, path-based scheduling).
- Number of variables may be increased (renaming).
- Number of variables and operations may be decreased (dead code elimination).

**Objective:** To ensure that the output of the scheduler preserves the input behaviour irrespective of what scheduling technique used

IIT Guwahati

And this actually changes the control structure. So, I have enumerated here. So, as a result of this optimization, what are the things can happen? So, control structure of the operation the behavior can change, operations moves across basic block boundaries, it can go from if else to the next if-else, from loop to outside, outside to loop, from if-else to outside. So, anything can happen.

The number of variable can increase or decrease, right. So, this can also number of operations can increase or decrease. These are all can happen because of this scheduling, right. And the speed processing. And what is our objective? Despite all these things happen operation movements, control structure change, number of variable change, number of operation change, the output remain same.

That means the dependency does not get violated within the program, right. So, that is we have to ensure. We have to ensure that despite all these transformations, my final program, the schedule program is actually functionally equivalent to my input program, right. That is what we want to do. But there are so many complicacies introduced, right. So, we have to see how we have to tune my basic equivalence checking to handle all these scenarios that is where the scheduling verification works out then, right.

(Refer Slide Time: 36:39)

**Required modifications:**

**Situation 1:** Variable sets  $V_0$  of  $M_0$  and  $V_1$  of  $M_1$  are **not equal**.

**Reasons:** Due to code optimizations like renaming, dead-code elimination, speculation, etc.

**What to do?** Have to modify the definition of equivalence of two paths.

**Solution:** Impose restrictions:

1. Final values of the variables resides in  $V_0 \cap V_1$  only to be considered during checking the equivalence of data transformations of two paths.
2.  $R_\alpha$  and  $r_\alpha$  of a path  $\alpha$  should be defined over  $V_0 \cap V_1$  and I (**restricted to  $V_0 \cap V_1$** ).

Restriction on  $R_\alpha$  and  $r_\alpha$  denoted as  $R_\alpha|_{V_0 \cap V_1}$  and  $r_\alpha|_{V_0 \cap V_1}$ , respectively.

The first thing that we understood that the number of variables in both programs are not the same, right. That is obvious because the scheduling we introduce some variables, we



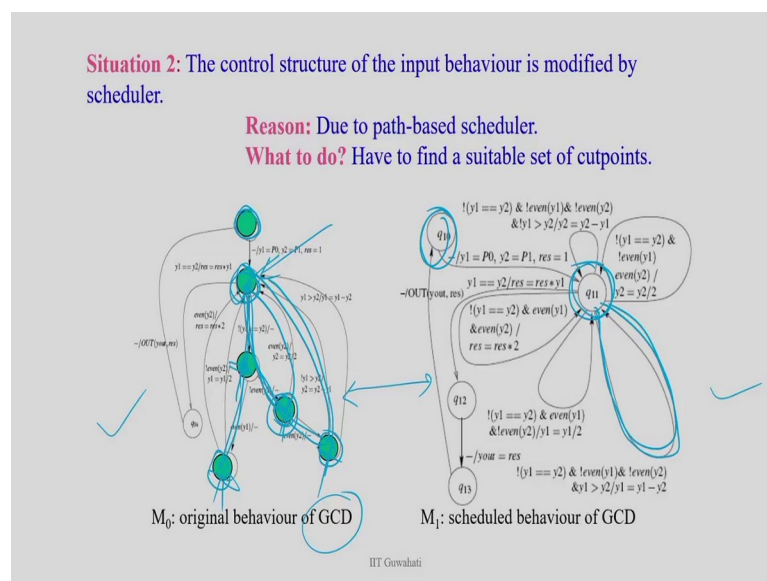
introduce, we do some constant propagation, copy propagation, and renaming, so the number of variables may not be the same, right.

So, how and, but when we check two program paths, we say that that data transformation are the same. And what does it mean by the data transformations are the same? That the corresponding variable must be transformed equally, right. But the since the number of variables can be different, what we usually do we will just check the intersection programs, right. So, what I am going to do?

Whenever I check two paths, I want to check the equivalence of the common variables, right. And there are things we have to consider whatever we have to do by  $V_0 - V_1$ , the variables resides here and  $V_1 - V_0$  in this set. So, this is the set  $V_0$ , this is the set  $V_1$ , right. So, I am going to check the equivalence for variables here. But if there are some variables here and there is some variables here, we have to say if it is defined that must be getting used later, right.

So, we have to take care of that, that particular definition get used later. So, these things are more complicated. I am not going to detail, but usually you cannot just ignore some definition, right. So, we have to make sure that, that definition get used later, till that point you have to consider that value and after that you can ignore, ok. So, that is the idea of handling this, these variables are not the same.

(Refer Slide Time: 38:05)



The important thing is that during path way scheduler your control structure may get modified. The GCD a example that I have taken earlier, so this is my input behavior and this is my output behavior you can see here that inside the loop all these paths are merged into single one. So, these paths merged into a single. So, this has 1 path, 2 paths, 3 paths, 4 paths, right. And that is merge to a single path in the original program.

Basically, these all states are get removed because of the scheduling. All operations are basically scheduled in a single state that is why these transformation happened, right. As a result all these paths become a single path, ok. So, now, the way I define the cut points, all these grid nodes are cut points because they have a divergence of flow there and you can see here I have only two paths. This is my start state this is one cut point and this is one cut point.

So, obviously, the number of paths will be different here, right. So, I cannot compare these paths, ok. So, how this is the problem; and because of this, the basic equivalence checking cannot handle these things, ok. So, how to handle this scenario, is very important, right.

And the point that you have to understand at this point here is this cut point says that you have to select cut points, so that all loops are cut. So, if you look into this behavior if I just select this one that is sufficient, right, but I have to choose more, right. And in some cases, if you choose this you may not able to show the equivalence also, ok. So, the point here is that the choice of cut points is not unique, right.

You can have many choices, you have to just make sure that your loops are getting cut and you can actually put redundant cut point as well. So, the choice of cut point is not actually unique. And it is not guaranteed that for every choice of cut point, you can actually show the equivalence, right. You can have only one choice of cut point for which equivalence can be shown.

For example, in this case, if you choose this and this as a cut point and this and this as a cut point, the equivalence can be shown. But if you choose any you just add one more cut point here, we are not able to do it, right. So, and, but there is no way you can actually say that if you choose only these two, then only you can show the equivalence. There may be some transformation where you have to choose this and this also, then only we can probably show the equivalence because of some scenarios, ok.

So, the choice of the cut point is not unique and there is no guarantee that for any choice you can show the equivalence. So, what is the way out? We will take a conservative approach. We will choose the maximum cut points; that means, all branching point has a cut point and whenever I found this particular choice is causing a problem, a cut point is causing a problem I will remove it, ok.

And what is the idea of removing? So, suppose initially say I have chosen this as a cut point and I have a this path and I found that for this path, there is no equivalent path there and I am going to remove this cut point, ok. So, I remove this cut point, then what will happen? That means I am actually extending this two path to the next cut points. So, I am removing this cut point means I am extending this path to the next cut points. So, that is the idea, right.

So, what I am going to do? I will take a conservative approach, I will choose maximum cut points for all branching points. And whenever I found that for a particular cut point I am not able to show the equivalence. What I am going to do? I am going to remove that cut point, ok and I will extend the path. And then I will go to check equivalence. I am going to keep doing this until I reach a loop point, right.

So, suppose I start from this, I remove this cut point, I remove this cut point, I remove this cut point, I remove this cut point, then I reach, so this path now this is the path and I show that this equivalent. But say, for example, I take this path and I do not find any equivalence, right.

Then the next step, you try to remove this cut point, right. But if you remove this cut point then this loop is not cut by one cut point, so then your definition of path cover is wrong. So, you cannot go beyond a loop, right. So, you can only remove cut points till they are not going beyond the loop, right. But if they go beyond the loop you cannot do you have to say, I do not know, I this program may not be equivalent. So, that is where the completeness comes into the picture, right.

(Refer Slide Time: 42:35)

**Outline of the algorithm:**

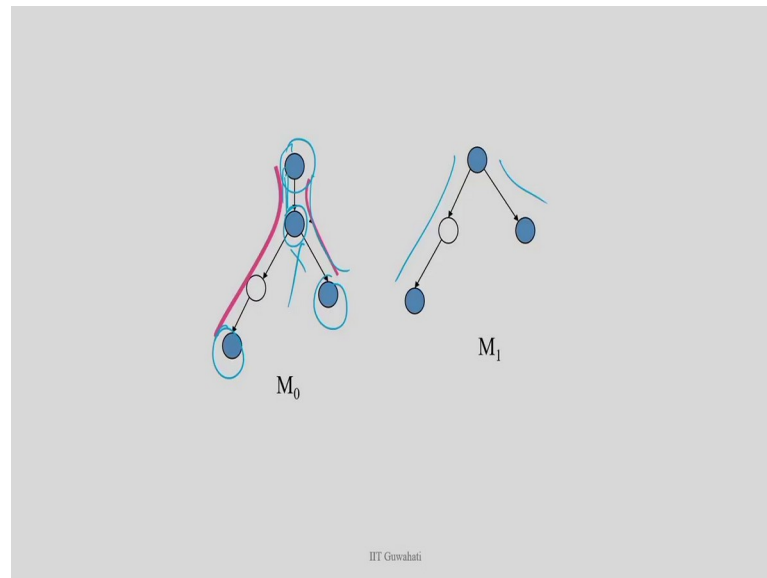
1. Insert the cutpoints in  $M_0$  and find the initial path cover  $P_0'$ .
2. Find equivalent path in  $M_1$  for each member of this set.
  - a. If equivalent path found. Put that path in  $P_0$ .
  - b. If equivalent path is not found for any of the path, extend this path up to its next cutpoints in all possible way. Remove this path from  $P_0'$ . Put all the extended paths in  $P_0'$ .
  - c. If path is not extendable report the non equivalence.

IIT Guwahati

So, that is the overall idea. So, the modified approach to handle this scheduling verification is that I am going to take the intersection variable and the variable that is not in the intersection, I have only you keep track of those variables till they are the last use after that I will ignore the temporary variables. And to handle this control structure modification, what I am going to do?

I will insert cut points initial path cover, I will find out based on the conservative approach all branching states. And then I will try to find the equivalent for each path from this path cover if I found the path cover that is fine, I mean if I found the equivalent of a path in the other program then it is fine, if not, I am going to extend the path up to the next cut point in all possible way. It basically means I just remove that next cut point, right.

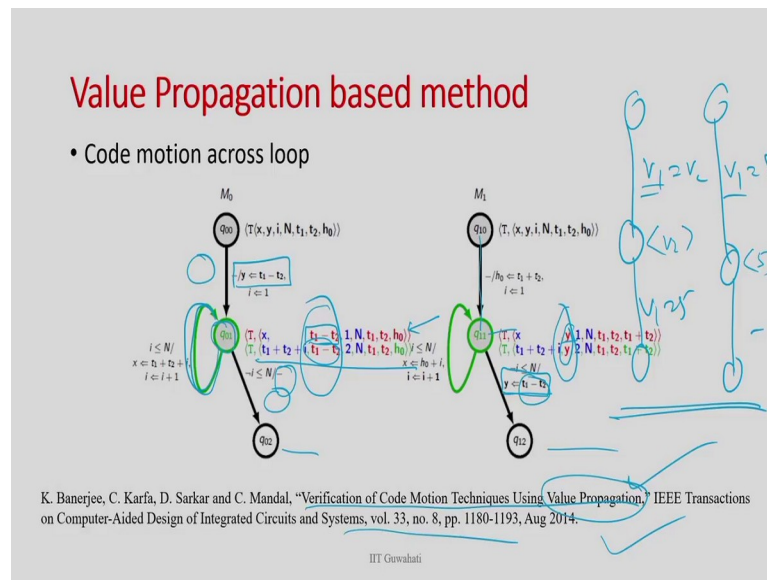
(Refer Slide Time: 43:21)



So, the basic idea is here. Say suppose initially I have this these are the cut points. I have 3 paths shown by the green, right. And try to; when I try to show the equivalence of this path, obviously, since this path is merged, I cannot show the equivalence of this path. What I am going to do? I will just remove I just remove this path to the next cut point; that means, I have removed these cut points. So, initially, there are 4 cut points, I have now 3 cut points.

Now, I can show that this path is equal to this path and this path is equal to this path. This I will keep doing until I reach a loop point, after that, I cannot extend this path, ok. So, that is the basic idea of handling this code motion. These things actually happen control structure modification, code motion, these are the cases you have to do this term, ok.

(Refer Slide Time: 44:05)



So, this is what about the method it was initially proposed, and then as I mentioned that since this particular problem is undecidable, there are many scenarios where this method says may not be equivalent, actually they are equivalent and they show that it may not be equivalent. And the primary example is that if you have a code motion across the loop, right, so, that means, there are some operations say here and it moves here, right.

So, this is a loop here. If this is happen, you cannot extend the path beyond the loop. So, you will not able to show the equivalence, right. If you we will take, so we extend a path beyond loop then we only can show the equivalence. So, in this scenario, although these two programs are equivalent, it will not able to show the equivalence. So, then in this paper, in this paper, we come up with a new idea called value propagation, ok.

So, what it does? Is basically what I am going to do I am going to instead of extending the path. What is the problem? When we compare two paths some of the variable values will not match, right. Say suppose  $V_1 = V_2$  and here  $V_1 = 5$ . It is not matching. So, whatever the mismatch value of  $V_1$ , I am going to propagate, ok. So, I will propagate  $V_2$  here and I am going to propagate 5 here.

And it might be that at this point there may be  $V_1 = 5$  and there is nothing here. So, at this point your  $V_1 \equiv V_2$ , right. So, the idea of this work is the value propagation. So, whenever there is a mismatch when we compare two paths and if I found that at certain

points there is a variable whose value is not matching, I will just propagate that mismatch value to the next subsequent cut points.

The advantage of this is that when you come to a loop point, right and if there is an operation that moves across loop, right. So, for example, here  $y = t_1 - t_2$ , it was before loop and it is after the loop. So, what will happen here? Here we can see that at the start of the loop  $y_1$  remain  $y_1$ , right. So, at this point that is the  $y$  is not updated here because there is no value of  $y_1$ . But here  $y_1 = t_1 - t_2$ .

But I can see that this  $t_1$  is  $y_1$  is defined here, right. So, I have to go beyond this loop. So, how can I go beyond the loop? What I am going to check that at the start point of the loop this is my mismatch value, I come the loop and this is my value at the end of the loop. And I can see that the  $y$  is not getting impacted, right,  $y$  remain the in both cases.

That means this expression  $t_1$  and  $t_1 - t_2$  is invariant to the loop body. So, I can propagate this value across the loop. So, basic idea is that if there is a mismatch of value at the start of the loop, I will check whether that particular value gets impacted in the loop, right. If it is not impacted in the loop; that means, it is invariant to the loop, so I can propagate that value across the loop.

And then once we propagate, and I come at this point the value of  $y$  will match because there is no  $y$  here, but  $y$  is  $t_1 - t_2$  and here  $t_1 - t_2$  is  $y$  there, so  $y$  will match, right. So, somehow, I have to propagate the value across the loop, and for that this concept of value propagation was introduced.

And with this we can show that even if there is a code motion code transformation across the loop, I can handle those things during verification. So, in this case, that path extension method will say, [FL] I do not know whether they are equivalent or not, but if I apply this value propagation method and there are some code motions happened across the loop, I can show they are actually equivalent, ok. I am not going into detail about the whole theory. But if you are interested you can read this paper.

(Refer Slide Time: 47:41)

### Code motion Involving Loops

- Can handle code motion across loop (scenario  $S_1$ ).
- Can not handle the  $S_2$  and  $S_3$  scenarios.

R. Chouksey, C. Karfa and P. Bhaduri, "Translation Validation of Code Motion Transformations Involving Loops," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 7, pp. 1378-1382, July 2019.

IIT Guwahati

Then, we have shown in this paper in 2019, that there are 3 scenarios of code motion. One is across the loop, so there are some operations before the loop, you move after the loop, and this is the loop body, right. And this is can be handled by the VP method, the value propagation method.

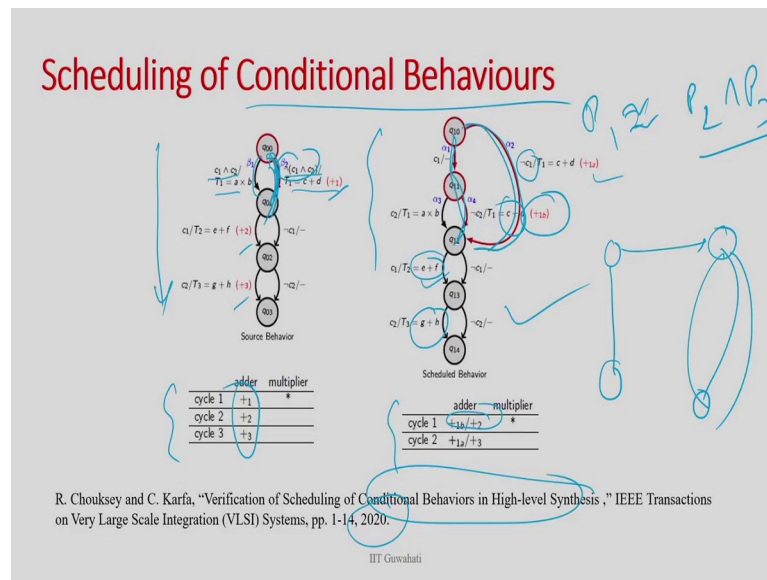
There will be two other scenarios there may be some operation inside the loop that go up beyond the loop, before the loop, right, it comes here, or there is an operation inside the loop that goes after the loop, right here. And vice-versa, it can be another way also. There are some operations, transformation can be this direction or this direction as well that there is a operation before the loop it comes inside the loop or it is inside the loop it goes out of the loop.

In this two scenario, the previous method cannot handle it because it checks the invariant at the loop point and because this operation moves across the loop this invariant checking at the loop point will fail. So, in this paper, we show that how we can guarantee that this kind of movement is safe or correct, right.

So, we had added some more clauses to the value propagation constant to the value propagation, so that this kind of moment also can be handled. This is basically again value propagation method, but with some additional features just to handle this these two scenarios, ok. Again, I am not going into detail. Interested readers can go into this paper, ok.



(Refer Slide Time: 48:59)



The next one is which we proposed in the last year is basically we found certain scenarios, where this behavior, specifically for the conditional behavior during scheduling one path may be split into two paths, ok. So, here it is shown by rate. So, earlier you can see here. So, it was the condition  $c_1$  and  $c_2$ , then you do this. And if it is a negation of that, you do this.

And it has been shown in some papers that if you have this behavior you need at least 3 cycles to execute, but if you have this behavior you need two cycles to execute. I can explain how. Say, suppose you have only one adder, right and so, if it is that then there are 3 addition operations this, this, and this.

So, you have you need 3 cycle because the condition is such that there is no mutual exclusive condition, right. So, if there are two operation have a mutual exclusive condition, you can actually put them into same adder because only one of them will execute, right.

For example, here this condition of this  $\neg (c_1 \& c_2)$ . So, there is no guarantee that there are mutual exclusive, right. So, as a result, you need 3 cycles using one adder. But here the way I split this condition, I just put it this is not  $c_1$  and this way. So, the these 3 paths equivalent to these two paths now, ok.

Now, now we can execute this 1b and this 1b and e+f, these operations because their condition is mutual exclusive this is  $\neg c1$  and this is  $c1$ . So, they are mutual exclusive. So, I can execute this, these two operations in clock 1, because one of them will only execute. In the next clock, I can execute this or this ( $g+h$  &  $c+d$ ), ok.

So, I can see that because of this conditional transformation the number of clock cycles can reduce. So, that is the scheduling part. But this brings a lot of complexity in the verification because once you take a path, how do you check the equivalence in the other program? We will just take, you just scan all the paths in the other program starting from this corresponding state.

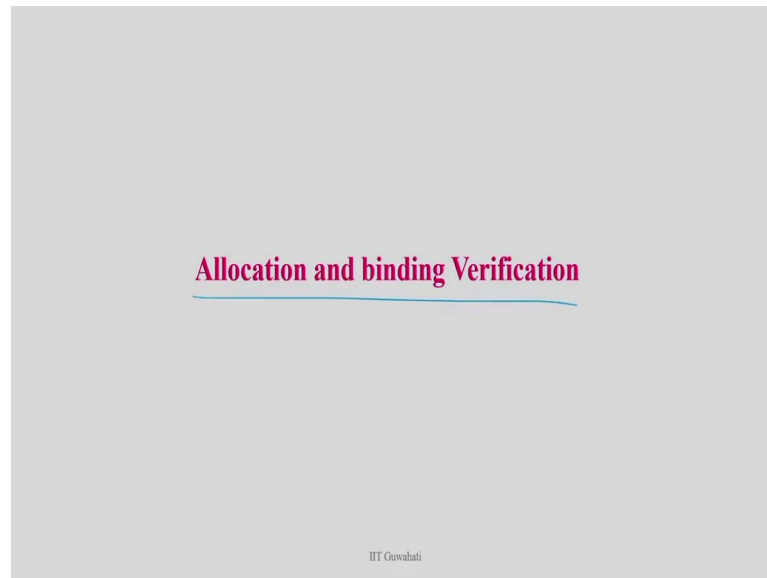
And if we found one path which is equivalent to this, then you will say this is equivalent if you there say 10 paths here and none of them is equivalent to this. You will say the equivalence cannot be found, right. But here you can see here the example that I have given this path is equivalent to two paths here, but that checking was missing in the previous paths.

Basically, here one path can be equivalent to the union of path1 and path2 and path3, right. That is what this is happening here and this is what things we have to handle in this particular paper, right. In this paper, we show that how we can identify one path is equivalent to the union of multiple paths.

So, earlier it was depth wise search, now we also have a breadth wise search, where one path can be equivalent to multiple paths and then we have shown that we have just (Refer Time: 51:52) that VP method to handle this kind of scenarios as well, ok. These are the recent works. You can again go through the paper for detailed theories. So, that is all about scheduling verification. So, you can understand that since this there are lot of complex transformations happening, so there are more methods to come to handle those complexities.

But this is a very interesting area of research because there are so many optimizations happening in recent times, specifically the data flow things is not handled here, you can actually think of that how can we handle the data flow during scheduling verification. So, those are the things is to be done. The other two steps are very simple, the verification task is simple. So, I just briefly cover them without going into detail.

(Refer Slide Time: 52:34)



So, the next one is the allocation and binding verification.

(Refer Slide Time: 52:37)

The slide contains the following text and diagrams:

- Data-path Verification:**
  - Objective:** Verify the register sharing among the variables.
  - Challenges:**
    - $V_1$  contains the variables and  $V_2$  contains the registers.
    - No. of registers < No. of variables. So, cardinality of  $V_1$  and  $V_2$  are not same.
  - Additional Information:**
    - The state mapping function:**  
 $f_{sm}: Q_1 \leftrightarrow Q_2$ 
      - Control structure of the scheduler output is not modified.
      - This function is a bijection.
    - The register binding function:**  
 $f_{rb}: Q_2 \times V_2 \rightarrow V_1 \cup \{\emptyset\}$

Handwritten notes in red and blue ink include:

- $V_0 \wedge V_1 = \emptyset$
- $V_0 = V_1$
- A diagram showing a node  $P_4$  with children  $P_5$  and  $P_6$ .  $P_5$  has children  $V_2$  and  $V_3$ .  $P_6$  has child  $V_5$ .  $V_2$  and  $V_3$  are crossed out with red lines.  $V_5$  is also crossed out with a red line.
- Equations:  $V_4 = V_4 - 5$  and  $V_6 = V_5 - V_2$ .
- A mapping function: 
$$\begin{cases} R_1 = \{V_1, V_6\} \\ R_2 = \{V_2, V_3\} \end{cases}$$

IIT Guwahati logo is at the bottom center.

So, in this phase as I mentioned that your variable get map two registers, right. And there is no control structure change, nothing change only the variable become register. So, for example, suppose you have a program like this which is say basically it is  $V_1 = V_2 + V_3$ , right, and say here  $V_4 = V_4 - 5$  and say  $V_6 = V_5 - V_2$ . So, this will be replaced by; so, they are basically replaced by, so this say this  $v_1$  become  $R_1$ , say this  $v_2$  become  $R_2$ ,

this v3 becomes a R3, this v4 becomes a R4, this v5 becomes a R5 and so on, right. So, this variable will become registers.

So, I have two version of the program where the control structure remains the same number of operation remain the same, only one program is in terms of the registers, another program in terms of the variable. So, there is no common variable. If you now just think about this V0 and V1, the intersection is null  $(V0 \cap V1) = \emptyset$ . There is no common variable because all variable are get renamed, right.

So, if we try to apply the equivalence checking method because we cannot compare two paths, because the when you try to compare two paths, we usually check the corresponding variable value. Here there is no corresponding variable, all variables are different. So, to do this we need this mapping information, right. How do you verify? Because I am going to take this information, the register mapping information, how, which variable map to it is registers.

Say for example, say R1 store say V1 and V6, right and say R2 store V2 and say V8 and so on. So, this mapping information I will extract from the scheduling tool. And what I can do in high-level? I can rewrite my register allocation program 1 of the program using this mapping information, right.

So, for example, since I know this R1 is V1, so I can replace in the program after allocation binding this R1 by V1, right. So, using this mapping information I can rewrite one program in terms of the variables of the other program. And once you do this, these two program become have the same variable, now this V0 become V1. Then I will check they are equivalent or not.

And what it verify here? If the, that mapping is wrong then you will say your equivalence checker will say this is not equivalent. If the mapping is correct, then only after replacing these variables your program will show the equivalence, right. If the mapping is wrong, your final value of the variables will be different, right.

So, this is what is the idea here that you have two version of the program where the all variables are different, I am going to extract the mapping information, and I am going to rewrite one program in terms of the variables of the other program and now these two programs have the same set of variables and I am going to check in the equivalence. If

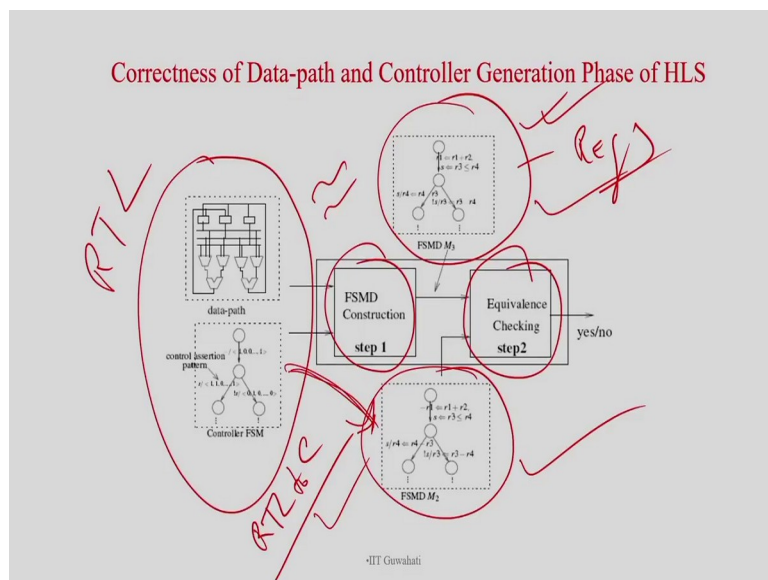
equivalence is shown, that means, the am mapping is correct. If equivalence is not shown then this mapping is wrong, right. That is all.

(Refer Slide Time: 55:38)



The next one is data-path and controller verification.

(Refer Slide Time: 55:42)

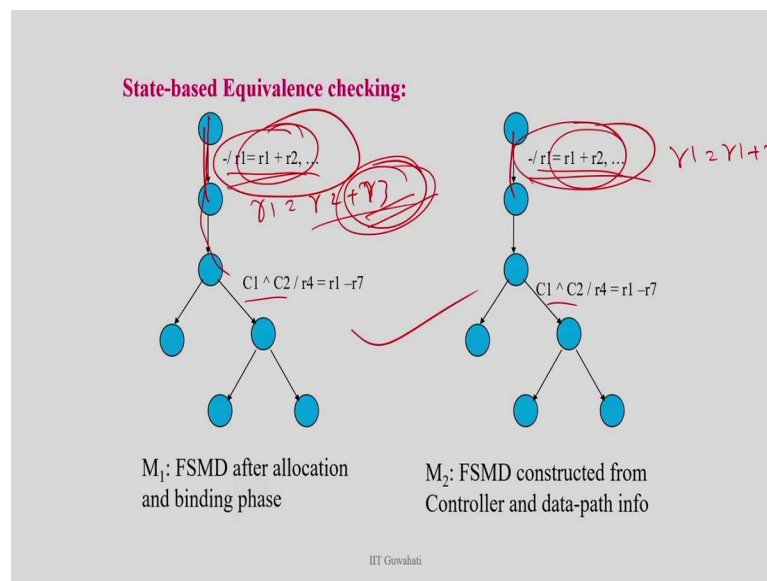


So, here it is more challenging because your program is RTL, it is not C, right. So, I do not have the C representation of the output. But the program after the allocation binding is basically a program, where the variables are the registers, right. That I have already defined.

So, I cannot just compare this because the equivalence checking method that I have talked about is basically for this kind of FSM, not for this RTL. But in the last class we have discussed about this RTL to C converter, right, RTL to C. So, with this step, I can actually extract a C behavior, FSM, from this, right. That is what exactly I have shown that. Once you do that you can check the equivalence, right.

So, the idea of this data path controller generation verification phase is you extract this FSM from this data path and controller. And the method is already discussed in the yesterday, in the last class. And then once you extract these two programs, you can check using the equivalence checker, right. And the point here is that once the main challenge is how do you extract this FSM, once you extract this FSM this structure of these two FSM is exactly the same, operations are exactly the same, there is no change, right.

(Refer Slide Time: 56:55)



So, you have exactly the same operation happening in every state, conditions are exactly the same, number of states, number of paths, number of transition, everything will be equivalent. So, I may not be able to go for path level equivalence. We can just check each transition whether the conditions and operations are same or not.

So, here the verification problem is much simpler because there is no code motion. The number of conditions same, number of transitions same, number of operations each transition must be the same, if not there is a problem. So, I have to just check each transition what is the corresponding operation whether they are equivalent or

not, right. But the primary challenge here is the extraction of this FSMD from the RTL. Once you do this equivalence checking problem be simpler.

And if you if the operations, if the transitions are not the same that mean I can say there is some problem in the data interconnections, right. So, that is how I can actually verify, that if say here  $r1 = r2 + r3$  is happening, and actually in the program should be  $r1 + r2$ , I will say that there is a problem here. So, your data path has some problem because I am expecting here  $r1 + r2$ , but you are getting  $r2 + r3$ , right. So, this is how I can actually verify the data path and control generation phase, ok.

So, with this I conclude today's discussion. So, just to summarize, this we can actually we have discussed in today's class a formal method, how do we check the equivalence of two programs, the basic theorem I have define defined, and then for each phases the scheduling, allocation binding, and data path control generation phases, I identify what are the challenges to be solved in this problem.

And we found that scheduling is the most complicated steps, and most of the research in this particular area formal verification of high-level synthesis target the scheduling verification. But the other two phases are relatively simpler problem because the control structure is does not change much, operation does not change much, ok. So, but there are lot of research is going on specifically in the verification of scheduling phase still, ok. So, with this we conclude today's class.

Thank you.