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 - 32
Simulation based Verification of High-level Synthesis

Welcome students. In today's class, we are going to learn about Simulation based Verification of High-Level Synthesis. To start with, let us understand what is verification.

(Refer Slide Time: 01:06)

Verification

- What is Verification?
- Why do we need verification of High-level Synthesis?
  - https://gcc.gnu.org/bugs/
  - "An Empirical Study of the Reliability of High-Level Synthesis Tools" – FCCM 2021
- How do we verify?
  - Correct by construction
  - Simulation based verification
  - Translation Validation/Equivalence Checking

So, verification is basically ensuring the correctness of something right, some process. So, why do we need verification of high-level synthesis? So, to answer these questions, if you understand this high-level synthesis, it is a very complex process right. So, it can go through scheduling, allocation, binding, then data path and controller generation.

So, a lot of complex transformation is happening, and there are a lot of inter dependencies are there. So, specifically, the bug or the incorrectness coming to RTL is because of some implementation error. You have to understand that this high-level synthesis tool was developed by not a single person, maybe a group of people right in a team.

And sometimes these people there might be some synchronization issues that there is some synchronization problem between two persons, or sometimes some particular corner case may be is not implemented at all right. And as a result, whenever you run some new application, there may be some scenarios which is not properly handled by the synthesis tool.

And as a result, the results or the RTL that is generated may not be correct right, so they may not be functionally correct with the C code. So, that is why we need to verify high-level synthesis or rather any synthesis process right which transforms one verse one input to another output we should always verify or check the functional correctness of the generated output with respect to the input right. So, that is what is verification. And since high-level synthesis is a complex process, we do need verifications.

To emphasize that, so if you I have just given two links here ok. The first link is a link if you go into that link it is basically reported many bugs of GCC right. GCC is a C compiler you know it is almost 30 years old right. Still, till today there are bugs. And it is so you can understand that although the tool becomes mature, there may be some corner cases that are not handled properly in the tool ok. So, that is an interesting link you can actually visit and check the bugs that is reported in recent times in GCC.

And there is another very interesting study which is published in this conference 2021 FCCM. It shows that it they took various high-level synthesis tool like Vivado HLS, Intel HLS tool, and then Bambu HLS, Legup, all the common popular high-level synthesis tool, and they do some empirical study, and they identify there are bugs in all the tools correct. They have reported many bugs so in till today.

So, this justifies that this high-level synthesis tool may have bug. And we must ensure that the generated RTL by the high-level synthesis tool is functionally correct to the input C or not right, so that is why the verification of high-level synthesis and that will be the topic of discussion for this week ok.

So, now, the question is how do we verify right? How do we ensuring the correctness of high-level synthesis? There are three conventional approach. One is correct by construction, second is simulation based verification, and third is translation validation or equivalence checking. So, correct by construction is something you making sure that all steps of the tool is formally proven to be correct ok.

So, it is basically you are ensuring that the tool itself correct by constructions or each step, each function, each line of the code is functionally proven to be correct ok. So, you can understand that in context of high-level synthesis as we mentioned many times that it is a very complex tool. So, making a correct by construction tool is very difficult, so that is why it is not something is so common approach to prove the high-level synthesis tool.

The common approach that is actually taken is the simulation based verification. So, what is simulation based verification? It is basically you simulate the generate a RTL and check whether that is correct or not ok. So, in the simulation based idea is basically you have the function, it can be anything. You have the function, you give a input, you will get a output. And you give the same input, and there is a golden circuit golden design which will give the output, this is the correct output, and then you compare right. This is what simulation based verification.

So, you compare these two, and say it is yes or no right. So, basically you run your design RTL, you with an input you check what is the correct output and what is the output obtained. If they are same that means it is a correct one; if otherwise it is wrong. And you do it for say 10,000 inputs, 20000 thousand inputs whatever possible way right, so that is what is simulation based verification.

And the third one is the translation validation or equivalence checking. So, here instead of checking the complete correctness of the tool, what we do we just checking one iteration of the tool or one execution of the tool is correct right, one translation of the tool is correct.

What does it mean? I give an input and I got a output, and then I just check the equivalence of that two right. So, you have the HLS tool right you give a C code you get an RTL and then you check the equivalence between these two right. So, you give this C here, you give this RTL, and you check whether there is yes or no ok.

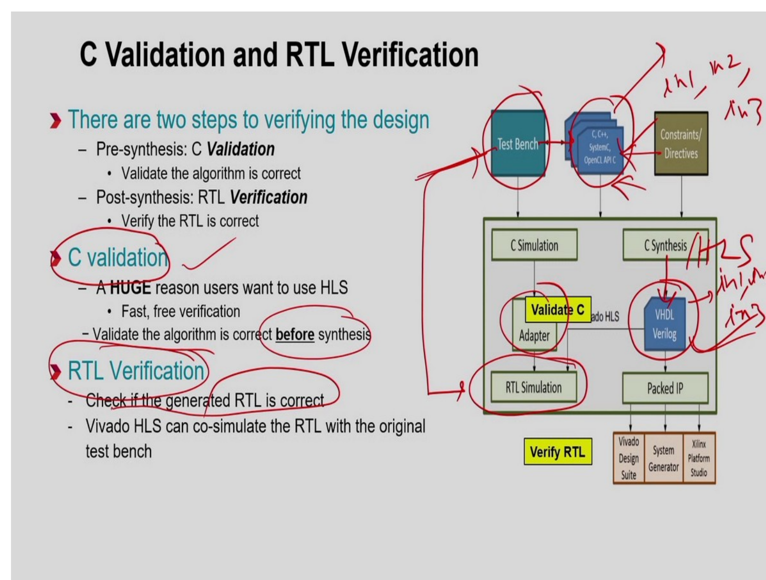
So, it does not guarantee the complete correctness of the high-level synthesis tool. Why? Because I just give one input say this input is a one merge sort, I get a merge sort output right, and then I just prove that that merge sort RTL is equivalent to the merge sort C right. If I get yes here, but it does not guarantee that high-level synthesis tool will generate a correct output for insertion sort or quick sort or bubble sort whatever it is right. It is just the one execution or one translation of the tool is correct right.

So, this is something a another very interesting approach. And it is basically formal verification. And it actually ensures that at least this particular input is this C and this RTL is functionally equivalent for all possible inputs right, s, that is the difference between simulation versus formal verification. In the simulation, we only if we run for say 100 test cases we check, this output is equivalent to 100 possible inputs. There may be 10,000 possible inputs, but I do not check for the rest of the things right.

So, simulation cannot give 100 percent functional correctness. It just say that whatever the input you have run, for that it is correct and it gives you confidence right. If you run for say 10,000 inputs and you get the correct output, all the cases, most likely this is correct, but you cannot confirm that this is a 100 percent functional equivalent right.

Whereas, this equivalence checking actually guarantees that, and they do not run any inputs right. The beauty of formal verification is that they never run a single input right. So, they just formally mathematically prove that this C and this RTL are functioning correctly ok. So, that is all approach. And in today's class, we primarily try to discuss this simulation-based verification quickly ok.

(Refer Slide Time: 08:51)



So, as I mentioned that the simulation-based verification is basically you run the things for some inputs. So, for in the context of high-level synthesis what are the things we should verify? There are two things we should verify ok. The first thing is that you write a C code for a specification right. And you have to first ensure that your C is functionally

correct right; unless your C is functionally correct how do you guarantee that generated RTL is correct right, so that is what is called C validation right.

So, if you say write a function, I will give an example in this class, and then you make sure that this basically matches your specification ok. And how do I do that? We can actually do it very simply as I mentioned you write a set of the test bench, and you have the C code written.

And you just run this input and compare it with golden outputs right. And you say if it is running for all possible inputs, this, this particular C is giving the correct output, so that means, it gives a kind of certain kind of confidence that my C is functionally correct right. So, basically, the C validation does the validation of the algorithm before synthesis happens ok.

And the second thing is the RTL verification right, so that is something you understand that you have this C, you have this high-level synthesis tool, you convert into Verilog or VHDL RTL, now you should check this one as well right. So, now, you have to check whether this is functionally correct or not.

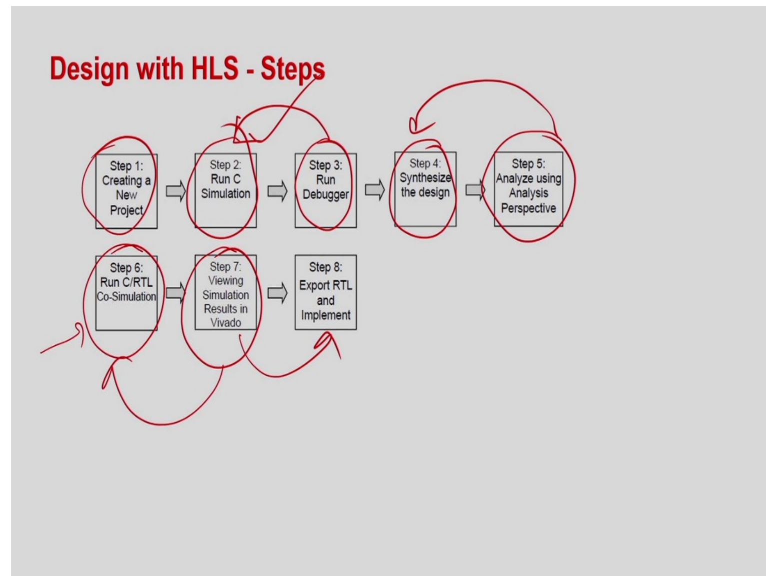
So, you can do the same process the way it is done in conventional high-level synthesis tools is that you have already written the test cases right. And I want to you reuse those test cases to verify this because the input-output specification of this RTL is the same as input C code because this Verilog is generated from this C code.

So, hence whatever the possible inputs and outputs of the circuit in the sense that variables are right. So, the input variable say this has input one, there are three inputs say in1, in2, in3. So, this circuit will also have three inputs right in1, in2, in3. So, internally these variables map to registers adding up to memory FC happen, but input and output correlation is always there for the C and RTL.

So, what I can do I can actually utilize this same test bench to verify this, but you have to do some kind of adaptation I will explain that. So, basically, this is called RTL simulation right. So, again I am going to put the same test case to this RTL, I will run it the RTL circuit using some RTL simulator. And then I will get the output, and I will compare it with the correct output, and we will just check that right. So, this is basically it is basically checking if the generated RTL is correct or not.

So, in the context of high-level synthesis, there are two simulation verification is done. You first verify your C, then you make sure that your generator your C is actually correct, and then you basically generate the RTL. Again you verify the RTL with the golden output and compare, and then make sure that your RTL is also functionally correct ok.

(Refer Slide Time: 12:03)



So, the conventional process of high-level synthesis with these verification steps included is this. You first create the project right. You, have created the project in the sense you write the specification, you write the C code right. And then you run the C simulation right.

This is step 2 where you make sure that your C is correct. Then if there is some bug, you should debug the code right. So, the GCC has a debugger called GDB right. So, you can use GDB actually debug the C code and identify where is the bug right.

So, you if so basically this is an iterative process unless you your all simulation passes you keep doing this to two steps, steps 2 and 3, and then once all simulation test case passes, that means, your C has a high confidence of you have high confidence on the correctness of the C, then you go for the synthesis. Step 4 is that you do the synthesis right.

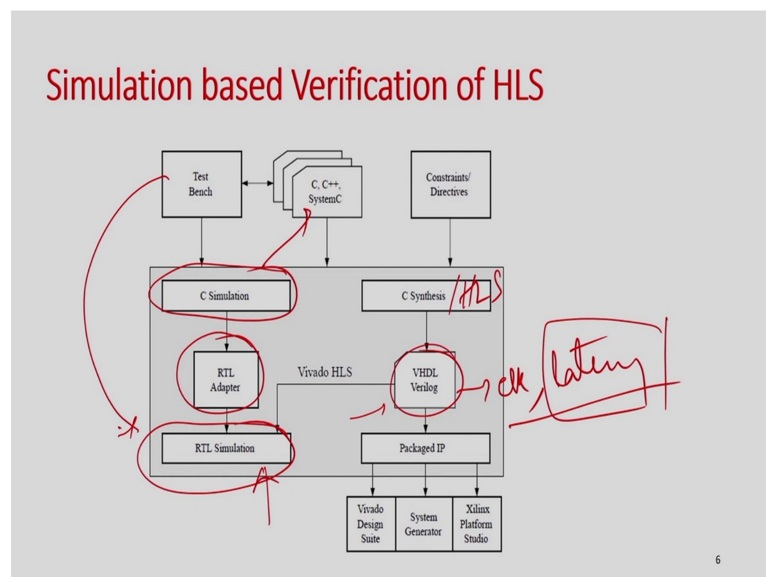
And then you basically after that you basically check whether your target clock is made, target performance is made, the area is within the limit or not. Otherwise, you go back add constant and you do this right. So, again this is a repetition here that you add some pragmas, some optimization, you apply to a pipeline, you do data flow optimizations, you apply array partitioning and so many optimizations are there. You keep doing this and you see whether your objective functions are met or not right.

So, once you are satisfied then you go to this step, step 6 that you now do the RTL verification right. This is called RTL co-simulation because I am going to use the inputs of the C to verify this RTL right. So, I am going to co simulate this C and RTL together because I have already checked my C is functionally correct, now I am going to check this C versus RTL right, I compare that.

So, again this is something if there are just some ways you can check the waveform that if your simulation is correct or your output is correct or not, if not you identify the bug you try to fix that or you try to report, and so this is a long process right.

If there is a bug in the RTL, it is not you can fix right, you have to report the vendor, and then they probably will give you a fix, and you will get it right. So, this is how you basically close the gap. And finally, you have confidence that your RTL is correct. Then you do this export the RTL and integrate a bigger system right. This is an overall high-level synthesis process ok.

(Refer Slide Time: 14:28)



So, this figure again highlights the things that you have two kinds of simulation, C simulation, and RTL simulation. So, C simulation ensures the correctness of the C specification. And this RTL simulation sees the correctness of the generated RTL by the high-level synthesis tool ok.

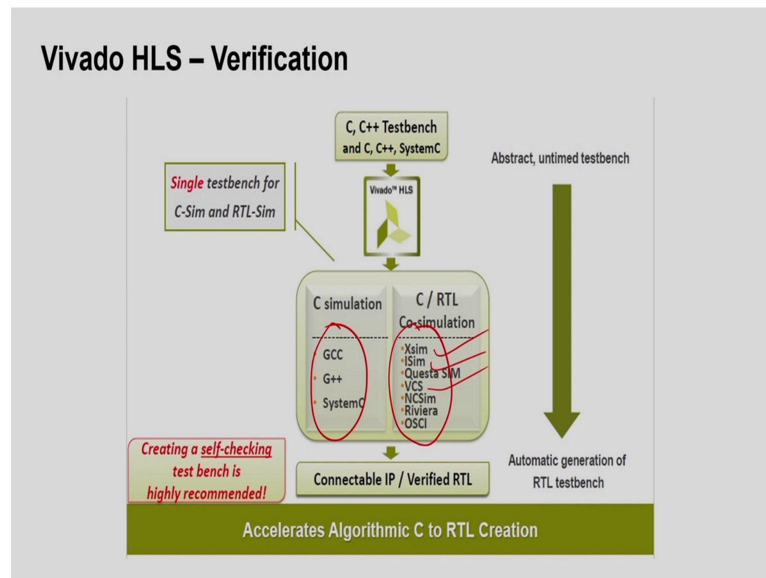
And as I mentioned earlier that I am going to use the same test bench here, but you have to do some kind of adapter because the way you run your C is not the way you run your RTL right. So, specifically because here, you have a clock, you have latency right. So, many other factors are there. So, you have to make sure that those things are already added to the test bench right.

So, you have the test bench, this input-output is the same, only thing is that you need some kind of tuning of the interface so that the same input can be given to the RTL right. So, in C, it is untimed, so you can give one input you get output and then you get the next output.

Here once you give an input probably you have to wait for say 10 clock cycles, 20 clock cycles whatever the latency of the design to get the output, then you give the next input right. So, you have to understand how this Verilog executes, and it can be a pipeline, it can be a data flow, or it is a task pipeline right.

So, there then you have to give the input to every clock. So, all these things because high-level synthesis converted it knows what is the RTL and how it is generated, and hence it will actually tune that or adapt that C code so that the same test input can be used for simulation of the RTL ok.

(Refer Slide Time: 16:02)



So, for C simulation, we usually use GCC, G++ system, and C simulators, and we are all familiar. And for RTL simulation, you have to use a simulator that is for RTL simulation right. I cannot use GCC to simulate the RTL because you have to know the convention of the semantics of the RTL now. So, what are the common simulators are there? Xsim, VCS, ISim, there are many. So, you can actually use any commercial vendor for that right XSim is for Xilinx.

(Refer Slide Time: 16:31).

C Simulation

```
float findSQRT(int number){
    int start = 0, end = number, mid;
    float ans;
    // To find integral part of square root
    while (start <= end) {
        mid = (start + end) / 2;
        if (mid * mid == number) {
            ans = mid;
            break;
        }
        if (mid * mid < number) {
            ans=start;
            start = mid + 1;
        } else{
            end = mid - 1;
        }
    }
    // find the fractional part up to 5 decimal
    float increment = 0.1;
    for (int i = 0; i < 5; i++) {
        while (ans * ans <= number) {
            ans += increment;
        }
        ans = ans - increment;
        increment = increment / 10;
    }
    return ans;
}
```

```
// Test Bench
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main()
{
    for (int i = 0; i < 100; i++) {
        scanf("%d", &N); //read from file alternatively
        x = sqrt(N);  //Golden output
        goldenOp = floor(pow(10,5)*x)/pow(10,5); //adjust decimal
        fnOp = findSQRT(N); printf("%f ", findSQRT(N)); //test Output
        if(goldenOp != fnOp){ //Compare
            printf("SIMULATION FAILS for %d", N); exit();}
    }
    printf("SIMULATION SUCCESSFUL");
}
```

And now I will give an example here. So, for example, suppose you have to implement hardware for a square root function say. So, you know for square root there is a function called SQRT in the math.h, but you cannot synthesize that sqrt into RTL right. So, then you thought of writing an iterative version of the square root. And what is happening here, you can actually go into detail later.

So, basically here I give the number. And then I identify in this part of the loop the integral. So, suppose it is 16, so you will get 4. So, you will get 4 here. But if it is say 15, your output will be some 3.7 something right. So, that integral part you try to identify here. And this part of the code you try to identify the fractional part ok. And up to 5 decimal points ok.

So, I am not going to in detail. Basically here the idea is that you do a binary search of things that you have to say you want to identify for 16. So, what are you have to do you try to take eight and you check whether 8 into 8 is equal to 16 or not right? If not, then I am going to search in this place right. So, I am going to take the middle of that. So, now, I am going to take 4, then I will take 4 into 4 equal to 16. So, ok, so I got it.

So, this is what is happening in this place right, it is kind of a binary search. You just take the middle, you check whether that square is greater than or less than. And based on that, you search in this place or that part ok. So, this is what is happening here. And the fraction part is doing the same thing, it just identifies the next five parts ok.

So, let us so, this is what you have implemented right. Now, this may or may not work right. So, this may not be your correct square root function, it may not give you the output. So, you have to verify that is why I say that your C specification once you have written, you must make sure that this function actually does the square root right. So, then you have to do a C simulation or C verification.

So, I am just written the test bench corresponding to this. You can see here I have the main function here and I check for 100 such random input. So, I take random inputs N. And this is my golden output because the square root function is already implemented, and that is correct. So, I get the actual golden output. And I adjust the golden output to 5 digits because it can be say 3.33333 infinite, but I need only 5 bits because this output is 5 bits. So, here I get the golden output up to 5 decimal points.

Then we call this function findSQRT() with the same input right, and then get an output. So, I call this findSQRT() function to get the output. And now I am going to compare if they are not the same, I will say that SIMULATION FAILS. And if all simulation passes, so I am going to exit it here. So, your simulation is not passing. Then otherwise if this loop successfully completes, that means, this is correct, so then I will say that SIMULATION is SUCCESSFUL.

You can notice here that main( ) is my test bench ok. And this function is the top-level function. So, in HLS always my actual function I will always give the function I want to implement. So, find a square root or anything. And main( ) is my test bench ok. And in the test bench, I am going to call this function my top-level function from the test bench, and I will compare it with the golden output ok. So, this is the overall idea.

(Refer Slide Time: 20:03)

Run C Simulation

- The C test bench includes the function main()
- The top-level function is used for synthesis.
- How to write good test cases?
- HLS reuses the C test bench to verify the RTL design
  - No RTL test bench needs to be created when using Vivado HLS.
- If the test bench checks the results from the top-level function, the RTL can be verified by simulation.
- *HLS automatically creates the infrastructure to perform the C/RTL co-simulation*

So, in the simulation, as I mentioned the test bench includes in main( ), and then the top-level function will be used for synthesis. And so one important point here is how do we write the good test cases. So, you have to make sure that your inputs actually cover all possible, possible scenarios of the function; otherwise, you might miss certain things right, so that is something you have to ensure.

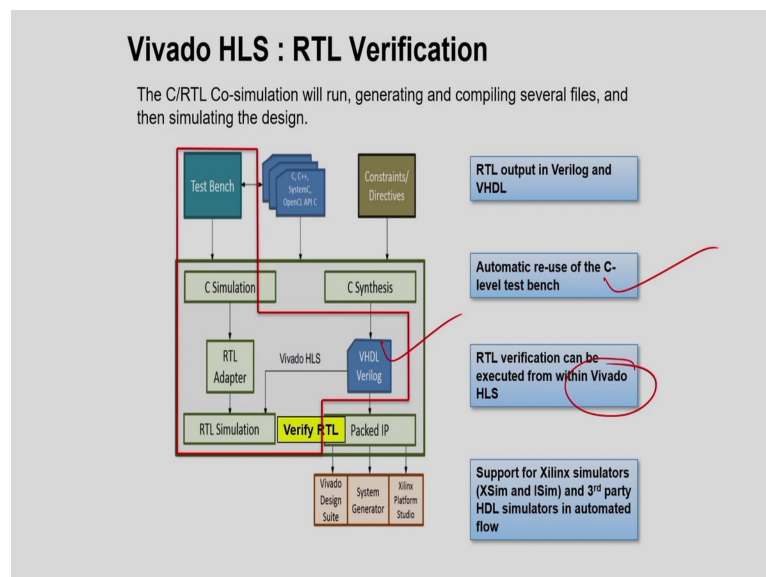
And there are many techniques I am not going to detail that functional coverage test coverage. So, many techniques are there. If you are interested, you can Google it but you

have to make sure that your input is something covering most of the scenarios of the functionality of the design ok.

And then as I mentioned earlier that that high-level synthesis actually automatically adapts that same test bench for verification of the RTL also. So, you do not have to do any manual work. If you do C simulation and you do HLS, it will automatically convert the test bench into a corresponding test bench for RTL to include which includes the latency, which includes the clocks, and you will be a making a Verilog file instead of C file.

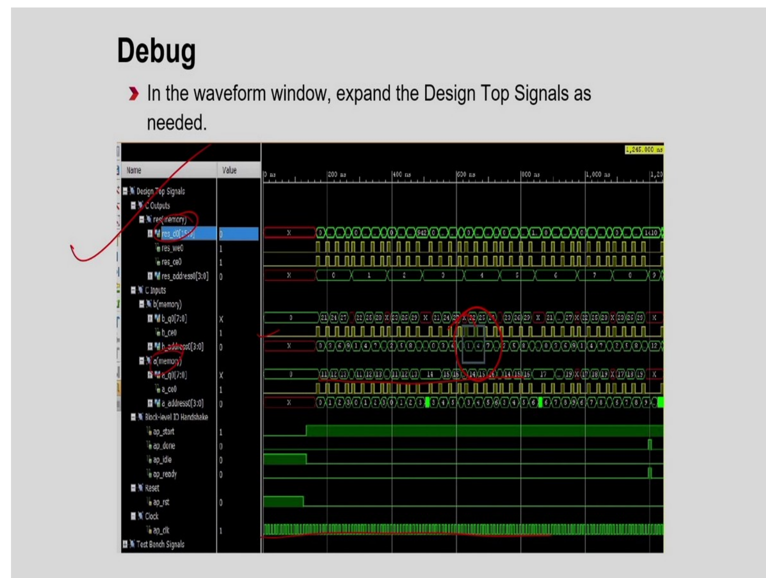
So, those things it will do right. And it will actually instantiate the module which actually implements the top level function. And then you can actually just do the RTL co-simulation, and it automatically verify the RTL, so that something it is already there in the most of the conventional High level Synthesis Tool ok.

(Refer Slide Time: 21:30)



So, this then basically this RTL verification ensures the variable correctness of the RTL. And as I mentioned that automatically it reuse the C level test bench and can be executed within the Vivado, it can be executed within the Vivado itself right, Vivado HLS tool, because I talked about Vivado HLS here because I am going to use the Vivado HLS for the experiment part ok in this class.

(Refer Slide Time: 21:53)



And if I as I mentioned that if you found there are some simulation mismatches, you have to always open the waveform. In the waveform, you have to select the particular signal which is causing the problem and you have to seek their corresponding value in every clock right. So, you can see here this is basically the clock, and you can actually every clock how all signals are getting modified. You can actually have to analyze this. You should have the expertise to understand this.

And then specifically say suppose you have to identify this is the problem here. And you go back to the circuit and fix it right or you report it to the vendor ok. So, this is how the RTL verification is done. And this is how the correctness of the high-level synthesis can be ensured by simulation-based verification ok.

Thank you.