

**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 - 33**  
**RTL to C Reverse Engineering for HLS**

Welcome students, in today's class we will talk about this RTL to C Reverse Engineering for High-Level Synthesis. It looks interesting because High-Level Synthesis converts a C code into RTL, but I am talking about it here, I want to generate a C code from the RTL is the reverse of the High-Level Synthesis process ok.

So, try to understand here that this RTL to C does not mean that I want to get back my original C code, but I want to get some form of C code that represents my RTL ok. It is not that I want to get back my original C code.

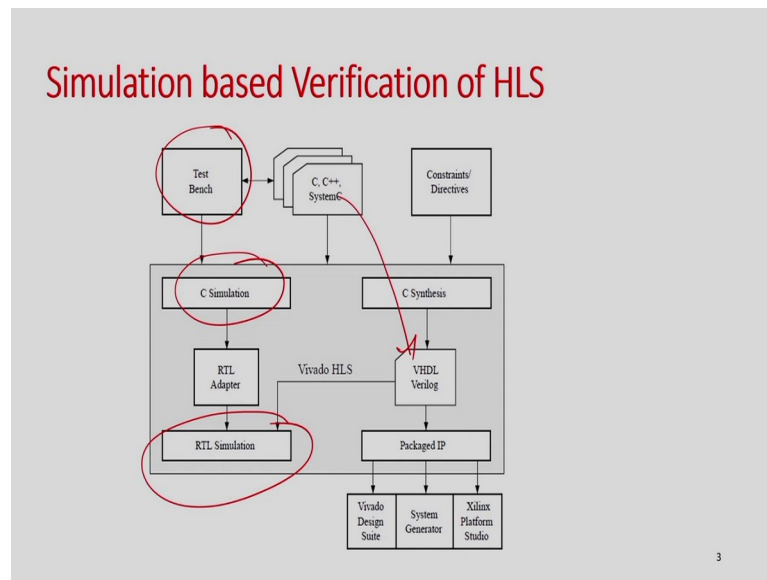
(Refer Slide Time: 01:25)

### Reference

- M. Abderrahman, J. Patidar, J. Oza, Y. Nigam, TM A. Khader, C. Karfa "FastSim: A Fast Simulation Framework for High-Level Synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, June 2021.

So, this work is primarily published here I am going to follow this paper for in this class ok, in this discussion.

(Refer Slide Time: 01:31)



So, what is the motivation for doing this right? So, I want to generate a C code from the RTL which is fine, but what is the motivation. So, to motivate yourself, let us look into this simulation-based verification that I have discussed in the previous class and we have mentioned that I will write a set of test bench. I first simulate my C and make sure my specification is correct and then I convert it into RTL, then I will do RTL simulation to verify that my RTL is correct ok. That is ok, but what is the problem here.

(Refer Slide Time: 02:05)

### Motivation

- RTL Simulation is **VERY slow** compared to C-simulation

Bench	C-sim(s)	RTL co-Sim(s)	Modelsim(s)
des	28.01	34672	36024
mips	0.985	2620	2885
aes_enc	12.656	4389	4693
aes_dec	14.442	4467	4780

*Handwritten notes: '15x' and '100x' with arrows pointing to the RTL co-Sim and Modelsim columns respectively. The 'RTL co-Sim(s)' column is circled in red.*

So, to do that you just look into this table ok. In this table what I try to do is I have taken some standard High-Level Synthesis benchmarks like des, MIPS and this and then I run that specification using GCC for say set of inputs I mean most likely it is 30 k and this is the simulation time. You can see where it is taking some 28 seconds, 1 second, 12 seconds, 14 seconds very fast right, for 30 k; 30,000 test cases right.

Now, I simulate I mean I convert this specification into RTL using Vivado HLS and then I run the two RTL simulators one is the RTL co-simulator which is within the Vivado HLS, and the Modelsim. There is another very popular simulator and see the time 28 seconds, 34,000 seconds right so; that means, it is basically 1000 times at least right 10,000 time slower.

14 seconds 4,000 second, 12 second 4,000 second, 1 second 2600 second. So, what this number suggests is the same RTL same number of test case it is at least 1000 times slower right; that means, from this discussion I can conclude that my RTL simulation is very slow and why it is specifically because it is actually do all this clock wise simulation see to do it actually analyze the all RTL in every clock and take lot of time right. And that is something a bottleneck for verification of High-Level Synthesis.

(Refer Slide Time: 03:48)

**Objective and Problem Statement**

Our aim is to perform RTL to C reverse engineering to generate equivalent C-code.

**MUST satisfy:**

- fast simulation
- functional correctness of the RTL
- cycle accurate simulation
- accurate performance estimation
- generate a highly readable and debug friendly simulation code

**MUST Supports:**

- array mapping to memory modules,
- non-inlined function calls,
- parallel execution frameworks invoked by loop unrolling, pipelining, task level pipelining etc.
- accurate simulation of pipeline stalls

So, this particular talk is talking about motivation is, if we convert this RTL to a equivalent C code what is the advantage I can give? I can now use GCC to simulate my test case and I can actually achieve this faster simulation right. The only thing you have

to make sure it is not only that I will just convert this RTL to C which is the motivation, but I have to make sure that this generated C code must satisfy these things.

Obviously, once we have the C code if I use GCC it will do a faster simulation there is no doubt about that, but it must satisfy the functional correctness of the RTL. You have to make sure that this generated C which I extract from the RTL actually is equivalent to the C code, that is the first thing. The second most important thing is the cycle-accurate simulator RTL simulators are cycle-accurate in the sense that they will give you precisely every clock information in the simulator.

If you want to see the value of the register in a clock cycle of 45 it will give you 5 points. So, basically, you can actually monitor the whole design each registers any component at every cycle. So, you have to make sure that you do not; not only generate the C code it must ensure the cycle-accurate simulation.

The second is the accurate performance estimations. So, once you run the RTL simulation it actually says how many clocks are needed and what is the maximum throughput every data will get right. So, you should be able to have the estimations. That is give the performance estimation and your C must satisfy this the most important is that you make sure that your generated C code is highly readable and it is to debug friendly right. So, highly readable in the sense that it at least you can see the code and you can read and you understand what is happening there.

And if there is a bug you identify during simulation you must be able to debug it back to RTL because you are not any longer actually simulating the RTL rather you are debugging the C code. So, this cross-relation must exist for this debugging-friendly simulation, and also this must support the following things because we have seen that in High-Level Synthesis there is a lot of optimization that happens right.

Specifically, array maps to memories and, functions become modules there are a lot of optimizations like loop pipelining, loop unrolling, task-level pipelining, and data flow optimization so, many things are happening right you must satisfy that your RTL to C conversion can support all those things right. If you can ensure both then I can use this C which is I generate from the RTL for faster simulation ok. So, that is the objective ok.

(Refer Slide Time: 06:19)

**Faster Simulation Based Verification of HLS**

- **Verilator** simulator parse the generic Verilog to C++ code
  - Disregards the FSM structure of the HLS generated RTL due to which impacts both simulation and debugging performance
- A recent work **FLASH** uses scheduling information to verify the RTL
  - Not able to detect the bug in allocation and binding phase and datapath and controller generation phase.

• Verilator: <https://www.veripool.org/>

• Y. Choi, Y. Chi, J. Wang, and J. Cong, "FLASH: Fast, parallel, and accurate simulator for HLS," IEEE TCAD, pp. 1-14, 2020

So, just to give a very idea that in this faster simulation of High-Level Synthesis is a real problem and there is two recent works, one is Verilator and one is called FLASH. Verilator basically generates a C++ code from the Verilog ok problem is that this High-Level Synthesis generated RTL is a very specific structure because it is a generic tool it does not take care of that and as a result that particular C ++ code we generate it is not so, readable and not debug friendly.

And the second thing is that this Flash basically generates a C code from the scheduling information after scheduling right. It does not generate from the RTL. So, as a result, it is not actually verification of the High-Level Synthesis because it is just verification of the scheduling. So, it cannot give the correctness of the allocation binding, data path control generation steps right.

(Refer Slide Time: 07:12)

### RTL to C Conversion

Idea: Take advantage of this RTL structure during RTL to C conversion

- In each state, the controller assigns 0/1 value to each control signal.
- As a result, a set of RT operations are performed in the datapath.
- The datapath sends results of some conditional checks to the controller. The FSM state transitions depend on those status signals.
- We analyze the datapath using control signals values in the particular state to identify the RT operation(s) executed in the datapath.

So, these are the two links for these two. If you are interested you can look into them. But, in this particular work that we have done, we have shown that I can convert is RTL into C by taking advantage of the RTL structure the generated by High-Level Synthesis tool. So, what is the RTL structure? So, as I discussed many times that the RTL that is generated by High-Level Synthesis has a very separable control on a data path right.

So, in the data path you have function unit, registers, multiplexer, and demultiplexer all the interconnections and controller is basically FSM right.

(Refer Slide Time: 07:45)

### RTL to C Conversion

FSM

DP

C

R = (R + R1) \* R2

Rewrite method

And what happens there. So, in this is say suppose sample data path right. So, data path and this is a sample controller FSM right. So, in the controller basically, the data path component has some control signals right. This control signal basically every clock controller gives 1 0 signal assigns the value to this control signal.

Say, for example, this is 1. So, if this 1 comes here. So, suppose this one is this so; that means, in this time this operation will be happening right not this one. Similarly if say this one is 0 so; that means, this particular register say this last signal is nothing, but this. So, this  $r_4$  will be updated, and say this is 0 which means this  $r_3$  will not be updated right. So, this is this.

So, this is how the controller actually controls the operations in the data path in every clock. So, this is very specific data structure of the RTL. So, what we want to do is that we want to take these control assignments. I want to identify what are register transfer level operations is happening here. If I can identify say suppose for this signals say suppose say is happening like  $R_4 = R_1 - R_2 * R_3$ .

So, I am just given example. So, if I understand that given these control assignments if this operation is happening I can replace this with this operation right. It can be there are many more than one operation that can also happen right. So, if I do this that I from this operation, this control signals I identify what are the register transfer level operation happening and I replace this by here, I replace this by this.

So, then my FSM converted into FSM. What is this, finite state machine with a data path ok this is very interesting? So, now, this is a finite state machine with a data path and now I actually abstract out my controller. I abstract out my data path and I get a behavioral description and I can actually write this description in C. How I will explain.

So, this is what is my objective. I want to analyze my data path for the controller input every clock and I want to identify what is the operation happening, then my FSM converts into FSM finite state machine with a data path and which is basically a behavioral description of the RTL, and then I can actually utilize this behavioral description to convert or write I can rewrite this behavioral description in any language format like C ok. So, that is the overall idea right.

(Refer Slide Time: 10:20)

### Rewrite Method

- Obtain set of Micro-operations (M) in the Data-path and its corresponding control assignment.
- Obtain **active micro-operations**  $M_A$  (subset of M) for a given control signal assignment A.
- Find the set of register transfer (RT) operations performed by the micro-operations in  $M_A$ .
- The **spatial sequence** of the micro-operations is in the reverse order of the way they are selected by the rewriting method.

So, the core idea of this whole work is how to obtain these register transfer operations from the control signals. So, that is the very core idea of this whole process. If you can do this rest of the things are can be handled. That is the core idea. So, to do that what we have done is we apply a method called the rewriting method ok. What is that? So, as I mentioned that in the data path the flow of the data is actually controlled by the control signals.

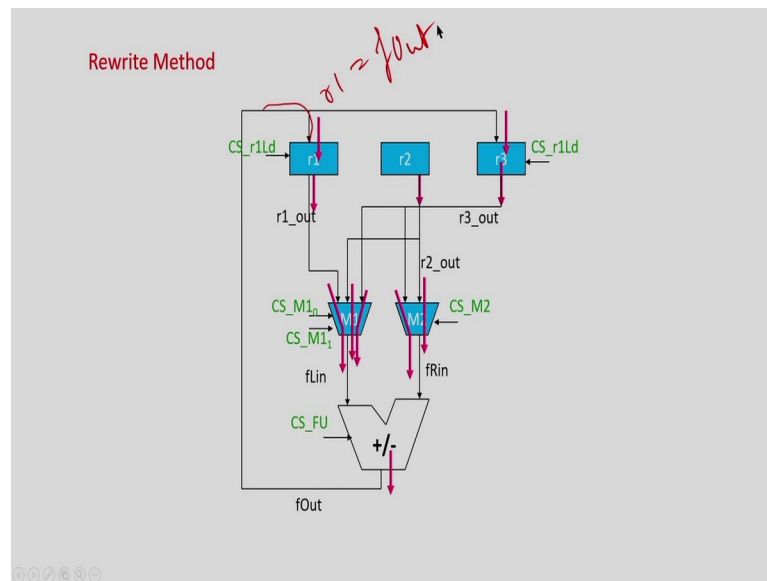
For example you take this mux again. So, in this mux there are two micro-operation is possible, either this value will come here or this value will come here and this is controlled by this. So, this is how first we have to identify what are the micro-level operation is happening data flow is happening in the data path. So, that is my first task ok. So, I will identify a set of micro-operation. Micro-operation is the minimum number of data flow from the input to the output of any component ok.

So, I will identify these micro-operations and their corresponding control signals right for example, here so, here this  $m_{out} = R_2_{out}$ , if this is equal to 0 and  $m_{out} = R_3_{out}$ . If this is equal to 1. So, I am going to identify these control signals every micro-operations and their corresponding control signal values ok. So, this my step one next is. So, in the data path, there are many such operations is happening right many micro-operation is happening.



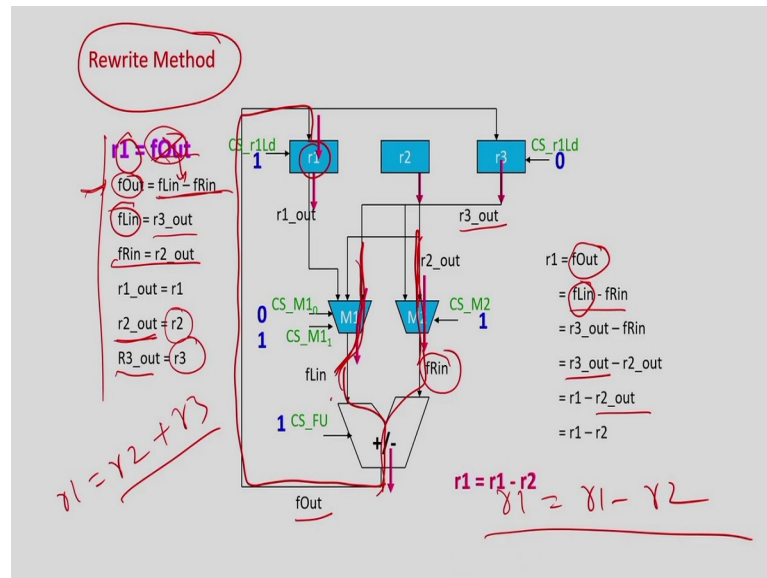
But, if I give a specific control signal only some of them will be active right for example, in this example again if this value is 0 then this operation is active. If this value is 1 then this operation is active. So, in a control state, this value will be either 0 or 1. So, one of the micro-operations is active and the other is inactive ok. So, for now, what I am going to do is to take a control assumption assignment and I will identify what are the active micro-operations in the data path ok which is possible to do.

(Refer Slide Time: 12:21)



So, I have taken an animated example. So, suppose this is my data path and these green signals are the control signals ok. So, first, what I am going to do is I am going to, identify all micro-operations all these red arrow is saying the micro-operations for example, here this is means this wire will be updated. So, this is  $r_1$  equal to this  $f\_out$  right. So, this is  $f\_out$  and so on. So, this way I am going to identify all my micro-operations this is the first step.

(Refer Slide Time: 12:55)



And say suppose I give the value to this signals is these values right. So, this becomes 1 this becomes 0, 1 becomes 0, this is 1, and so on right. If I do this not all micro-operations will be active, only some of them will be active. So, I am going to remove the inactive one. So, these are the active micro-operation in this state ok. So, this is to this point it is a clear right and I have just noted it down.

So, these are the active operations happening in the data path. The third step is very important and most important. So, these are the active operations and they are performing some register transfer operations right. So, let us try to take an example say this  $r_1 = r_2 + r_3$  right. So, this is the operation happening.

So, then what will happen? First, this micro-operation is to be happening right. So, this will happen then this micro-operation to be happening right this addition operation to happen, then this mux this data should come here, then these micro-operations should happen and then this data will come to this signal and this data will come to signal right. These are the micro-operation is happening and you can see that in the hardware it is basically this is these operations are happening and this is basically happening in parallel, but it is actually data flow is happening like this right.

So, from this, I will go here. From this, I will go this way to this and I will go to this to this right. So, then only this  $r_1 = r_2 + r_3$  is happening. So, there is a special sequence of operations. So, what I understood from this there is a sequence of operations is

happening here which actually makes sure this operation is happening and which RT operation is happening we have to identify that sequence of operations and that is where I apply the rewrite method.

So, from where I should start. I will start from a micro-operations where the left-hand side is a register when the register assignment is happening right. So, I am going to take this  $r_1 = f\_out$ . So, I will start with this micro-operation where the left-hand side has a register and then I am going to see the right-hand side whatever the signal who is rewriting that right. So, from here I can identify this signal is rewritten by this micro-operation because this is what is happening.

So, what I can do. I can replace this with this expression right. This is the rewriting I am rewriting the right-hand signal with the I identify a micro-operation which left-hand side is  $f\_out$  and I take a signal  $f\_out$  and then I will replace this  $f\_out$  by this expression  $fLin$  and  $fRin$  right. So, this is how the sequence is happening that is what is happening here. So, I take this and then I found that this  $f\_out$  is nothing, but  $f\_out = fLin + fRin$  or say  $f\_out = fLin - fRin$ .

So, then this is what happens. So, this is the sequence. Then what I can do I will take another signal again. I am going to take a signal again and I will identify is there are any operations for the left-hand side is  $fLin$ , yes this is the operation. So, then I can replace this  $fLin$  by  $r_3\_out$  because this  $fLin$  is nothing, but  $fLin = r_3\_out$ . So, this sequence of operations. So, this is what I am done.

And then similarly I take this  $fRin$ . I will replace it with  $r_2\_out$  because this is the micro-operation that is happening here. So, I will replace this by this operations then this

$r_1 = r_3\_out - r_2\_out$ , I will take this  $r_3\_out$ . I will see  $r_3\_out$  is nothing, but  $r_3\_out = r_3$ . So, I can replace this  $r_3\_out$  with  $r_3$  right. So, this is what I do, similarly, I can do this  $r_2\_out$  I will take  $r_2\_out$  and I identify there is an operation called  $r_2\_out = r_2$ . So, I can replace this  $r_2\_out$  by  $r_2$ .

So, what I got? I got  $r_1 = r_1 - r_2$ . So, for this control assignment 1001 these things in my data path  $r_1$  equal to  $r_1$  minus  $r_2$  is happening ok. So, that is the basic idea that if I just do this for all steps, I can convert this into this FSMD right that is the idea. So, once I do this I have an FSMD right. So, this is what I just to explain.

(Refer Slide Time: 17:00)

Rewrite Method

---

**Algorithm 1: Rewrite (RTL)**

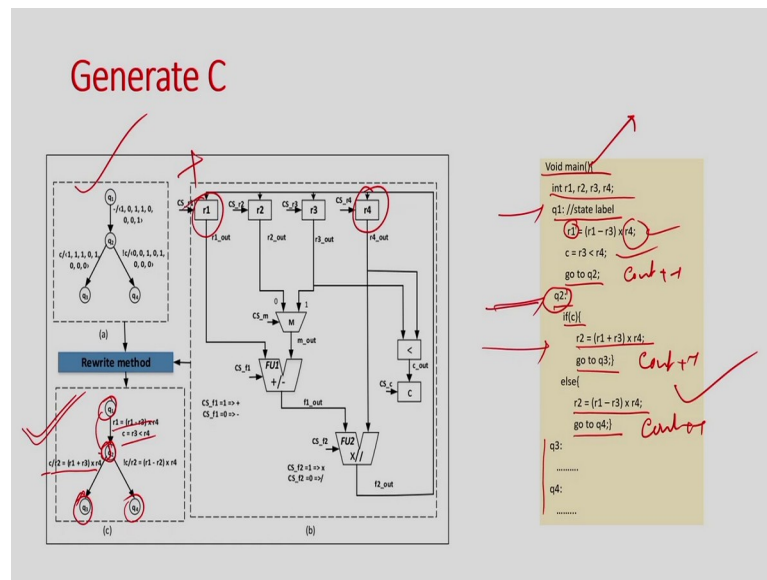
```
Input: RTL
Result: An FSM D
/* RTL consists of a Data path D and a
controller FSM F */
1 foreach state S in the controller FSM F do
2   Find the active micro-operations  $M_S$  for the control
   signal assignments in S;
3    $R_S = \Phi$ ; /* Set of RT-operations in S */
4   foreach micro-opn of the form  $\mu: r \leftarrow e_w$  in  $M_S$  do
5     do
6        $w =$  Find the left-most wire signal in the RHS
       exp  $e_w$  of  $\mu$ ;
7       Find a micro-opn of the form  $w \leftarrow e_w$  in  $M_S$ ;
8       Replace  $w$  with  $(e_w)$  in the  $\mu_e$ ;
9       while (all signals in RHS exp  $e_w$  of  $\mu$  are either
       Input, Reg or Constant);
10       $R_S = R_S \cup \{\mu\}$ ;
11   end foreach
12   Replace the control signal assignments in S of F with
    $R_S$ ;
13 end foreach
14 Return F; /* FSM F is converted to FSM D */
```

And now so, this is the overall technique and the overall methods. So, what I am going to do at the High-Level. I will take for each state I identify what are the active micro-operations in that state and then I will identify for each micro-operation where the left-hand side is a register because those many RT operation is going to happen. I will identify a micro-operation where basically for a replacement I identify micro-operations where my  $\mu$  is the left-hand side right.

So, I will replace that and then I will keep doing this until all my right-hand expression signals become input or register right. So, this is an I just whatever example I provided it is the same example you can go into the steps and you understand. I will keep doing this for all such micro-operations where left hand is a register. So, once this is done then I will replace that control signal with the corresponding RT operation that I obtained here right.

And then I am going to do it in each state right this is how I am going to convert my FSM into an FSM D ok.

(Refer Slide Time: 18:04)



So, at this point, we will understand. So, basically, it is a process that I have already explained. With this, I convert this FSM plus data path into this. So, the beauty of this method is that I now remove all my clock, reset, RAM, and ROM everything is right I do not have anything. I do not have any multiplexer. I do not have any adder multiplier. I have this behavior right. I do not have any control.

So, this is the abstraction happen. Once this happen now the question is that how I can write this in C form and the answer is very simple right. You just write a main( ) see this is a very small example there will be some input. So, you just specify the inputs. So, that will be there in a bigger practical circuits and then you just define all the variables, all the variables here as integer or whatever the type you give some type for them and then you actually write this as a level operations right.

So, I just have a level  $q_1$ . I have a level for each state that will become a level in the FSM right. So, in the state FSM, this is my state level I am going to do these operations. I will write these two operations and then I am going to state 2. So, I just go to state 2. So, I am going to write it in terms of go to. I am not writing if-else, for loop and all this. This is a very simple one. Then in  $q_2$  if this condition satisfies I am going to do this and this. So, I am in state 2.

If this condition is true I am going to do an operation then I am going to state  $q_3$  else I am going to do if these operations and go to state  $q_4$  right. So, this is how I just write for

each state operations and what is the go-to statement and these are the levels right. So, whenever you say I go to state  $q_2$  till my control flow automatically come to this time right and this is nothing, but a C code. You can see here this is something a C code that I generate from my RTL which is generated by the High-Level Synthesis tool.

And the beauty of this code is that I have an  $r_1$ . I have an  $r_1$  in my hardware. I have  $r_4$ . I have an  $r_4$  in hardware so; that means, the variables of this behavior are my registers. So, it basically has a very one-to-one correlation with my data path. This is not my input C in my input C there may be some variable  $b_1$ ,  $b_2$ , or  $b_3$ . This is not something like this. It does not have any for a loop this does not have anything it actually represents my hardware, my registers, my state these are the states right.

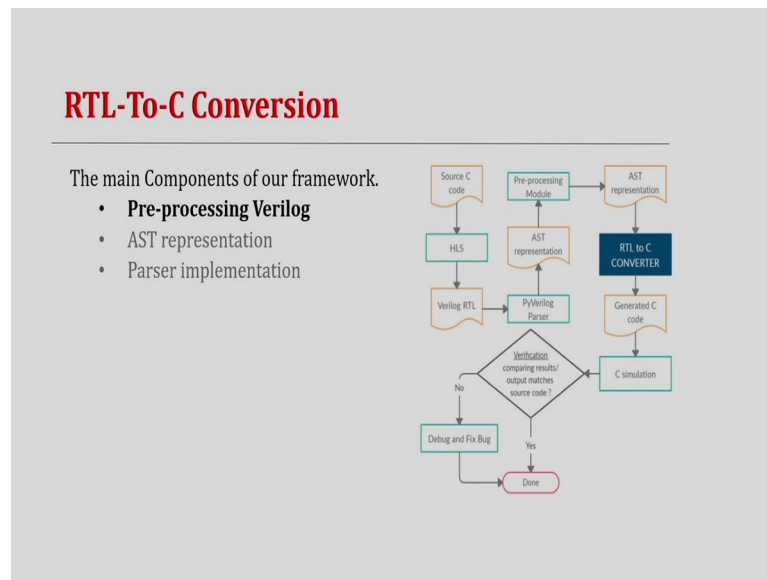
So, this is cycle-accurate because I have information for each state right. So, if I just put a counter ++ every state then I can say that if I run this how many clocks it takes this; this clock right and every clock what operation is also happening I can actually track it right. So, if I want to go to say state 2 what are the operation is happening I can print that my debug right.

So; that means, this is cycle-accurate because I have all the state information and operation is happening state-wise. I have to debug friendly because this is the register the variable represents the registers of the behavior. It is a C code. So, it is a it can be used for faster simulations. So, most of my targets whatever I specify the mass support it satisfies my this generated C code right. So, that is great.

So, only thing is that this is the basic core idea. This is how I can convert it or when you take a practical circuit there may be many complexities that will come right. So, those are the complexities that will arise when you try to convert an RTL into C that I am going to discuss next ok.

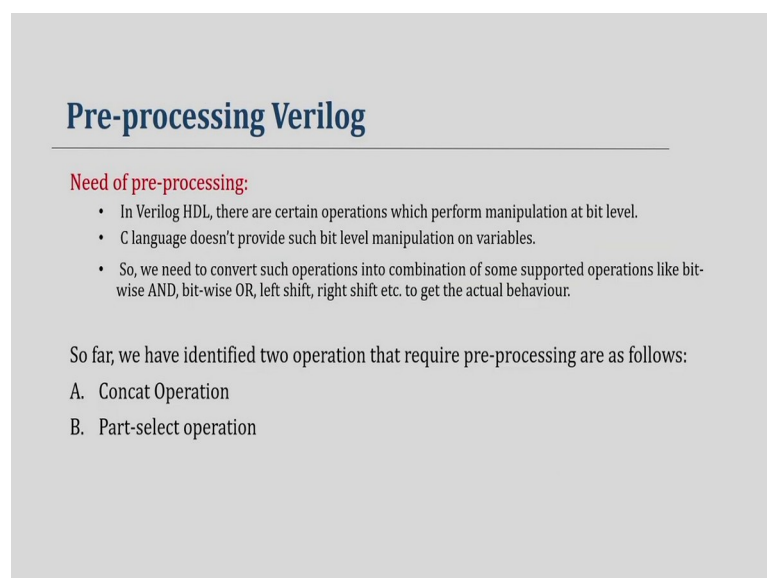


(Refer Slide Time: 22:45)



So, this pre-processing is something step that I just did not explain in the previous slide this pre-processing. Why do we need some pre-processing of the Verilog because there are certain operations which are supported in the Verilog and may not be supported in C code? So, we have to convert that operation into an equivalent operation in C ok that is what is the pre-processing.

(Refer Slide Time: 23:10)





(Refer Slide Time: 23:13)

### Concat Operation Preprocessing

- It concatenates the contents of one or more register at the end of another register.
- Multiple concatenations may be performed which is known as replication.

```
// Concat operation:
reg [WIDTH_A - 1 : 0] reg_A;
reg [WIDTH_B - 1 : 0] reg_B;
reg [WIDTH_C - 1 : 0] reg_C;
assign reg_A = {{reg_B}, {reg_C}};

// After pre-processing:
wire [(WIDTH_A - 1):0] temp_0, temp_1;
assign temp_0 = reg_B << width_C;
assign temp_1 = temp_0 | reg_C;
assign reg_A = temp_1;
```

Fig.1.4

I will give you two examples. There may be many. So, for example, the Concat operations. So, in the RTL the Concat operation is very interesting. So, basically, you can Concat two registers and you can write the data into this right. So, what is happening here is. So, both this basically you have this that you convert into this is regC content, this is regB content and this the whole thing will become my reg1.

So, this is not there are no such operations in C right which we do this what we can do it. I can rewrite this. So, I can just shift this regB by the width of the regC right. So, this is what I did here. So, I just shift the reg B by the width of reg C. So, what will happen then in this you will have regB will come here and it will be all 0 here right. So, this I store in a temporary variable then I just do this with or with regC right. So, there is a regC here.

So, if I just do a or of this, the regC value will come here right 0 and this. So, this becomes this you can understand this. So, this reg C will come here now and this is my regA. So, this is what I just do here right. So, sometimes some operations like the Concat operation which is not supported by C because I have to execute this in C, I can rewrite that operation in terms of using multiple such operations like the example I have given.

(Refer Slide Time: 24:38)

### Part-select Operation Preprocessing

- Operation select specific part of the register.

```
// Part-select operation:
reg [63:0] reg_A, reg_B;
always @ (*) begin
    reg_A[40:0] <= reg_B[51:11];
end
// After pre-processing:
wire [63:0] temp_2, temp_3, temp_4;
always @ (*) begin
    reg_A <= temp_4;
end
// width of register reg_A = 64 bit.
assign temp_2 = reg_B & 64'd4503599627368448;
assign temp_3 = reg_A & 64'd18446741874686296064;
assign temp_4 = temp_2 | temp_3;
```

Fig.1.5

There is another one Part select say, for example, the example here is there. So, I just do it like this I take this 51. So, both are 64 bits. So, I took some bit of register A right. So, this is a register A so, this part I just took right. So, 11 to sorry till this 1 say ok 51. So, this is my regB I want to take this part of the things and I put in the regA. I want to put this till 0-64; 0 to 40 this part.

I want to take this part of the code and I want to put it here and rest I want to keep as it is because if I just do this, I am not changing this content of this part of the regA right. So, this is what this operation does. So, what I can do here I can do it very simply I can actually take an integer where I just put all 0 and this I will put all 1 and all 0 right..

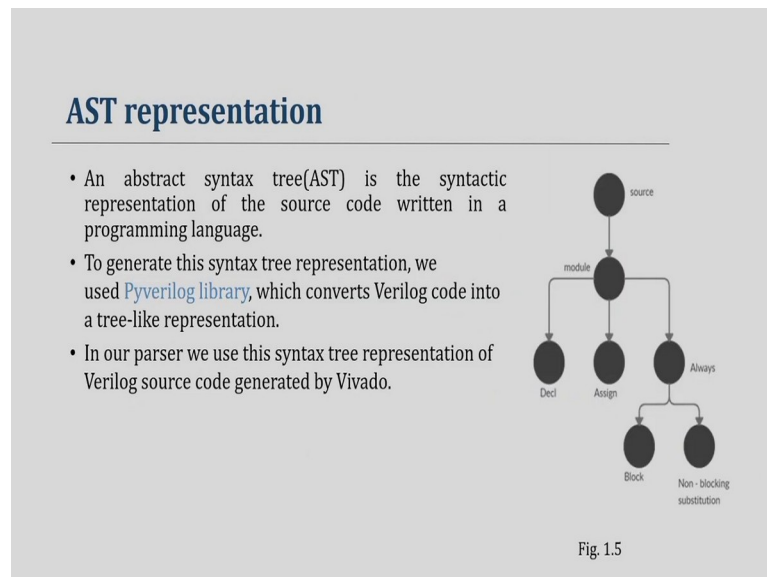
So, this is that number and if I do an AND with this. So, what will happen? So, finally, my this content I am put a temporary variable right. So, then it will happen this will all become 0. So, basically, I extract the value from 11 to 51 right. This is what I did here and then what I have to do, in the regA what I have to do I have to make sure that I will just keep this value as it is. I want to put this all 1 right I want to keep this value and I want to put 0 here.

So, what I did I just take a integer which represents that 41 to 63 these bits are 1 and 40 to 0 are 0. So, this is nothing, but this number is right and if I do a AND width this is what will happen because it is 0 end so, this will become 0. So, I erase this content, but at

this part, I keep it as it is right. So, whatever the value I had earlier I will keep it because this is end with 1.

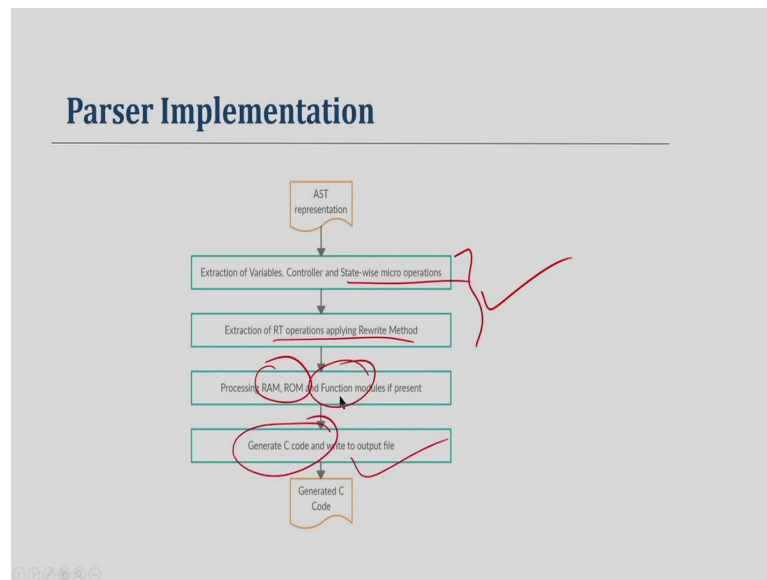
So, now everything is ready. Now, what I have to do I have to just copy this data into this right. So, that is what I have done here. So, this is how we can actually have certain operations in Verilog which you can manipulate at the bit level and you can actually do the equivalent code in C. So, this is my equivalent C code ok. So, this is what the part selected. So, there is two such operations I explained there can be many and you have to do such things ok.

(Refer Slide Time: 27:05)



AST representation as I mentioned is an IR representation where you actually have all the things as a syntax tree and you can actually do operations you can manipulate easily. So, in the flow, you have to use some kind of AST representation and we use some syntax tree for interpretation, but for discussion purposes, this is not so, important right we need to understand the idea first right.

(Refer Slide Time: 27:34)



Then the Parser implementation that you convert the operation which is not supported and you get an AST in which all those things are replaced and then you actually do that rewriting part right. So, you have to extract these micro-operations. You have to identify the rewrite operation these are all discussed. So, I am not going to go into detail things that I have not discussed yet is how-to handle RAM, ROM, and functions right. So, that I am going to discuss now.

And generated C code is also kind of discussed right. So, these are the things I miss I will just discuss now.

(Refer Slide Time: 28:05)

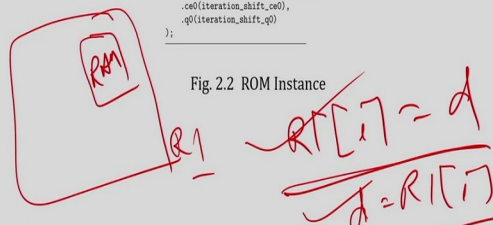
### RAM and ROM Module

```
shiftingcidi_sub_dRe #(
    DataWidth( 64 ),
    AddressRange( 16 ),
    AddressWidth( 4 )
)
sub_key_U(
    .clk(ap_clk),
    .reset(ap_rst),
    .address(sub_key_address0),
    .ce0(sub_key_ce0),
    .we0(sub_key_we0),
    .d0(sub_key_load_reg_17),
    .q0(sub_key_q0)
);
```

Fig. 2.1 RAM Instance

```
shiftingcidi_itercud #(
    DataWidth( 2 ),
    AddressRange( 16 ),
    AddressWidth( 4 )
)
iteration_shift_U(
    .clk(ap_clk),
    .reset(ap_rst),
    .address0(iteration_shift_address0),
    .ce0(iteration_shift_ce0),
    .q0(iteration_shift_q0)
);
```

Fig. 2.2 ROM Instance



So, in the RAM ROM in the hardware what is happening you have an instance right. You just instantiate a RAM I hope you are familiar with this kind of syntax that it is 64-bit data, address is 16 bit, address width is 4, and address range is 16 ok and these are the signals you are connecting to this particular module instance of the RAM right.

Similarly, for ROM right. So, it has signals like  $q_0$  and then it has address it has clock enable, write enable and data right. So, this is very standard you can actually Google it and then what I am doing. So, basically, this RAM is instantiated on hardware right. So, this is your top-level module and this there is a RAM here right and if you want to write something to this you will basically do you just put write enable equally to one in that particular clock.

I am not going to access this RAM in all clocks right all-state. It will be in a certain state that I want to write some data say what I am going to do, I am going to put this write enable equal to 1. I will put some address data some value of the address. I will put some data in my  $d_0$  and then whenever the next clock come it will be written right. So, it is nothing, but the RAM is something is say  $R_1$  right what is happening here. So, you just perceivably putting in some location  $i$  some data this is what is happening in C.

So, we have to do that right. So, what we did here is basically we identify the RAM, what are the RAM module, what are their values right, what are the way you are actually connecting these things from the instance, and then you see in which state these control

signals are becoming 1 and then if it is a write enable you just put these operations and if it is a read enable. So, you just put  $d = R_1[i]$  right. So,  $i$  is the address.

Whether you read or write. So, in this operation you have to just place in that particular clock right. So, we basically identify the modules, identify the corresponding signals and we will actually check the state in which state the writing is happening. So, if it is writing is happening we will just write this statement there, if it is reading is happening then I am going to put this statement in that particular state because the RAM is not accessing in all clocks in which clock it is getting accepted you just put that operation there right.

(Refer Slide Time: 30:20)

### RAM and ROM (Continued...)

Parser identify status of signal "ce0" and "we0" in each state and if these signal found set then place the RAM/ROM block at the end of that state in generated C-code.

```
if(1 == ap_CS_fsm_state5){
    sub_key_address0 = sub_key_addr_reg_455__temp ;
    sub_key_ce0 = 1;
    sub_key_we0 = 1;
}
if(sub_key_ce0){
    sub_key_q0=sub_key_ram[sub_key_address0];
    if(sub_key_we0){
        sub_key_ram[sub_key_address0]=sub_key_load_reg_170;
    }
}
goto ap_ST_fsm_state2;
```

RAM

Fig. 2.3

So, the ideal in the C level it will look like this. So, say suppose in state 5, you get that my clock enable = 1, write enable = 1, and my address is this so; that means, you want to write something right. So, what I am going to do I will just put this kind of  $\text{if}(\text{clock enable} == 1)$  then what I am going to do I am going to read this data whatever the address is there that value I am going to read it to this value and if the write enable is there I am going to write this value into this data right, this is how.

So, basically, I am not going to put this operation in every clock wherever this read enable or write enable is 1, I am going to do this operation. So, this will be my representation of the RAM or ROM in C code right is basically it is an array axis. So, we have to abstract that level thing and only thing we want to make sure that in which clock things is happening ok.

(Refer Slide Time: 31:16)

## Function Module

- Function module mimics the idea of function calls in C program.
- HLS tool usually generate separate verilog file for function module having a datapath and a controller FSM.
- In top module there is instance for the function module as shown in Fig. 2.4

```
calculationofkeys grp_calculationofkeys_fu_193(  
  .ap_clk(ap_clk),  
  .ap_rst(ap_rst),  
  .ap_start(grp_calculationofkeys_fu_193_ap_start),  
  .ap_done(grp_calculationofkeys_fu_193_ap_done)  
  .L(L),  
  .R(R)  
);
```




Fig. 2.4

So, similar to the RAM and ROM the function is basically again to create a module right.

So, whenever there is a function, it creates a module right. So, this is my function in this is a top-level module. So, what is happening this function may not execute again in every clock. It is specifically executed in a particular state right. So, what we have to do is we have to, basically we can actually again identify what are the modules function modules are there in the RTL and then you specify what are the basically this input signals right.

So, from this you can identify what are the input signals for these particular modules right and whenever I saw that particular values are set in a particular state, I just call that function. So, this module in C level what is this is a function call right.

So, what I am going to do. I identify this and whenever to identify I see some assignment is happening in a particular state, I just invoke this particular function in that state. It is not that again the same thing this function is not going to execute in all-state. In some states where it is getting executed it actually, we are get scheduled right and where it is getting scheduled I can understand from this signals where these particular signals are getting assigned ok.

(Refer Slide Time: 32:23)

## Function Module (Continued...)

- Similar to the RAM/ROM module parser identifies the status of "ap\_start" signal of that function, if it is set then place the function call in that state as shown in Fig. 2.5.
- Our parser supports hierarchy of function calls as well. Since, the top module waits until the completion of execution of the module it called, cycle accurate simulation is achieved by following the states of the respective FSMs.

```
if((exitcond2_fu_233_p2 == 1) && (l == ap_CS_fsm_state2))
{
    grp_calculationofkeys_fu_193_ap_start_reg = 1;
}
if (grp_calculationofkeys_fu_193_ap_start_reg) {
    grp_calculationofkeys_fu_193_ap_start = 1;
    calculationofkeys_41_45_46p_clk_4grp_calculationofkeys_fu_193_ap_done,
    grp_calculationofkeys_fu_193_ap_start;
}
goto ap_ST_fsm_state8;
```

Fig. 2.5 Sample example of function in generated C-code

So, basically the same way I just have an example here that if whenever this value is 1 some say so I am checking that this particular value is happening, I just call this function right with the corresponding argument right. So, that is all and basically, if you go to the RTL of Vivado HLS it has some ap start kind of signals which actually says that this particular function is going to execute in a particular state ok.

So, this way basically objective is that the function is become a module to identify the function module and then invoke that function call in that particular state where it is actually scheduled ok. So, that is I am going to do.



(Refer Slide Time: 33:02)

### Generate C-code

```
#include<stdio.h>
#define CONSTANT
function_prototypes();
int main(){
    //variable declaration;
    //RAM_ROM declaration;
    state1_label:
    //copy old_var_value=new_var_value;
    //place operations which belong to all states here;
    //place operations deciding condition variables here;
    //place level-triggered blocks here;
    if(condition1){
        //operations
        ..
        RAM/ROM blocks(if any)
        function_calls(if any)
        goto state2_label;
    }
    if(condition2){
        //operations
        ..
        RAM/ROM blocks(if any)
        function_calls(if any)
        goto state3_label;
    }
    state2_label:
    ...
    ...
    end:
    return;
}
```

So, with this, all these things are done and I have already specified how I am going to write my FSM into C basically using goto statement I will put a level for each state.

I am going to put the operation of that state in that state in that state and then I will say go to the next state right, this is the structure. So, every state I will talk about this old valuable thing. So, I just put the operations then if there are some conditional operations, I am going to put that the RAM block function call these are the things. I am going to do this under some conditions there may be this RAM ROM operation running then you go to this state 3 and so on.

So, this I have already discussed. So, this is a very nice way I can convert an RTL into C code and, but you have to remember this does not do not work for generic RTL. This will work only for RTL which is generated by the High-Level Synthesis tool because I am taking advantage of the data path and controller separation. If they are mixed you cannot separate out you cannot do this ok. So, this is how I generate ok.

(Refer Slide Time: 34:06)

## Major Challenges

Due to abstraction gap between Verilog and C language, we faced some major issues are as follows:

1. Data inconsistency.
2. Signed conversion.
3. Data-width mismatch.
4. Level-triggered Operation.

So, this is the basic idea, but as I mention that once you try to implement it in for practical circuit there are many challenges which come you have to face it you have to fix it and some of the challenges that I have we identified let me explain one by one ok.

(Refer Slide Time: 34:25)

## 1. Data inconsistency

- This issue occurs for RAW (Read after write) type of operations.
- When modified register values is used in other read in operation.

Example:

```
always @(posedge ap_clk) begin
  if (1'b1 == ap_CS_fsm_state2) begin
    a <= b;
    c <= d;
  end
  assign b = x + y;
  assign d = a + m;
end
```

*Handwritten notes: 'a = b' with an arrow from 'a' to 'b', 'c = d' with an arrow from 'c' to 'd', and 'a = d' with an arrow from 'a' to 'd'.*

Fig. 2.6

```
ap_CS_fsm_state2:
  if (1 == ap_CS_fsm_state2)
  {
    a = x + y;
  }
  if (1 == ap_CS_fsm_state2)
  {
    c = a + m;
  }
}
```

*Handwritten notes: Red arrows indicate the flow of data from the first state to the second state, showing that the value of 'a' is updated before being used in the second state.*

Fig. 2.7 Generated C-code (Before)

```
ap_CS_fsm_state2:
  a_old = a;
  x_old = x;
  y_old = y;
  m_old = m;
  c_old = c;
  if (1 == ap_CS_fsm_state2)
  {
    a = x_old + y_old;
  }
  if (1 == ap_CS_fsm_state2)
  {
    c = a_old + m_old;
  }
}
```

*Handwritten notes: Red circles highlight 'a\_old' and 'm\_old' in the second state, indicating that they are the values from the previous state, preventing the data inconsistency.*

Fig. 2.8 Generated C-code (After)

The most interesting one is the data inconsistency. I think this is very interesting. So, in hardware, if there are two operations is happening in the same state. So, they are running in parallel because this is hardware every module run in parallel right. So, for example, suppose in this example.

So, if I just do  $a = b$  and  $c = d$  is happening right. So, and  $b = x + y$ , and  $d = a + m$  this is happening. So, if we do the rewriting actually  $a = x + y$  and  $c = a + m$  these two operations it is happening ok. So, now, you try to understand. So, now, you can see here  $a$  is getting updated and  $a$  is getting used here. In C code if I just write this way that if (state 2)  $a = x + y$ , if (state 2)  $c = a + m$  right.

If I just write this way what will happen if you run this C code what will happen, this operation will execute first and this will be updated and this value of  $a$  will come here because this is a sequential behavior in C, but in hardware, because this two operation is running in parallel, this is the  $a_{old}$ , not the new value right. Before updating the ' $a$ ' you have some value in the ' $a$ '. So, that value is going to use here it is not the this value is coming here because this is a parallel execution right.

This ' $a$ ' will be updated at the start of the next clock, but this operation is happening in the current clock right. So, this is something is called that the problem of data inconsistency because in hardware since everything running in parallel, there is no read after write dependency (RAW), but here whenever you place in sequential in C code it will have a read after write dependencies and which is not correct because if you take the new value it is something wrong.

So, how to solve that I already given you the idea that this  $a_{old}$ . So, use  $a_{old}$  right. What I do here is that at the start of every clock I will take the value of every register and put in some old value; that means, this is my old value, and then if in the right-hand expression I am going to always use the old value. So, then what will happen if this  $a$  is and this  $a_{old}$  is not the same. So, this will take my previous value of  $a$ .

So that means, although now if I execute this behavior sequentially this is actually if taking it actually executing the parallel behavior of my hardware right. So, this is something very interesting and this is how we fix the problem that you store the value of the register at the start of every state and in the right-hand expression I am going to use the old value, not the new value. So, there is an if, even if I put the operation in the series, no read after write dependency will create here.

Because I am always using the old value which is not this value right. This is kind of  $a_{new}$  value ok. So, this is very interesting.

(Refer Slide Time: 37:14)

## 2. Signed conversion

- “Signed” keyword is used in Verilog code to convert the value into its signed representation in other words we must use its two's complement value.
- Issue occurs if we use its +ve value in place of its original value.

**For Example:**

- In Verilog code:  

```
assign X_reg = ($signed(7'd117) + $signed(Y_reg)); // (X=32 bits, Y=20bits)
```
- In generated C code (after):  

```
X_reg = twos_complement(117, 7) + twos_complement(Y_reg, 20);  
// X_reg = (-11) + Y_reg_new; ..... (4)
```

*Handwritten notes: "2's complement" with arrows pointing to the Verilog and C code examples.*

The next problem that we faced is basically signed conversion in the Verilog we have whenever there is a number is a negative number or sign number it is used by the signed right. So, sign number is basically it keeps the 2's complement of that number right.

So, although this is 1 1 7 integer it is basically not it is a two's complement of some number. So, this is not actually 1 1 7, it is actually -11 right. So, if I just write 1 1 7 here it will create a problem right. So, basically, what we did here very simple whenever there is a sign signals are there sign words are there, we always use a two's complement function right. So, we just here define a two's complement function and then I call this value with this.

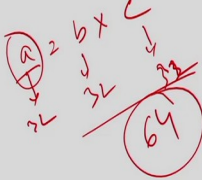
So, it will automatically convert these 1 1 7 into two's complement form of 7 bit because you have to give specify the bit width and then it will automatically this function will determine -11. So, I am not going to use 1 1 7 rather I am going to use -11 similarly this is a Yreg. So, it will just convert this an expression which is basically a two's complement of Yreg ok.

So, this is simpler, but you have to use the twos complement function to convert them next one is also very interesting.

(Refer Slide Time: 38:27)

### 3. Data-width mismatch

- In Verilog, if data-width of LHS and RHS of a operation is not same then it automatically adjust.
- If we directly realized it into C language than it can cause overflow or underflow issue during assignment.
- Solution is improve the rewriting method. As shown in Fig. 2.7



```
// RTL Code
reg [20:0] r_A;
reg [20:0] r_B;
reg [31:0] r_C;
wire [31:0] w_C;
always @ (posedge ap_clk) begin
    r_C <= w_C;
end
assign w_C = r_A * r_B;
// Incorrect Generated C code
unsigned long long int r_A, r_B, r_C;
r_C = r_a * r_B;
// Resolved C code
unsigned long long int r_A, r_B, r_C;
r_C = (r_A * r_B) & 64d'4294967295;
```

Fig. 2.7

So, in the data path in hardware you always know that you can actually have any specific data path width right your width can be anything right and register can be 43 bits, 27 bit, 7 bit, 9 bit, 65 bits anything right, but in C you have integer float and so, on right long long int. So, integer is 32 bits, long long int is 64 bit and so on. So, how we will represent these 43 bits in C that is a problem right.

Because you have say 33 bits you have integer say 32 bits then it is a problem and specifically the problem will more severe. It is a underflow and overflow problem. I will give an example say suppose you are doing  $a = b * c$  if say this is 32 bits, this is 32 bits if you multiply this result will be 64 bits and say a is also 32 bits then in hardware if you just do this operation automatically it truncate the 33 to 64 bits that part of the bits because its automatic truncation happen.

Because since the data width of a is 32 bit it will only keep the 32 bits or say it is only 43 bits only it will keep although result is 64 bits, it will keep the 43 of bits of that and it will store in this. So, this automatically happens as there is no problem of overflow and underflow in C Verilog because this truncation happen automatically or zero padding is already happened.

But, if you convert them because as I mentioned earlier that whenever you convert you have to give a data type of this variable and we usually give int long int and this type of data path. So, you do not have the opportunity to keep this 43 bits, 17 bits, 15 bits right.

So, that will create an overflow or underflow right, to avoid this what we do basically is we just take those particular bits right. So, that is the basic idea right.

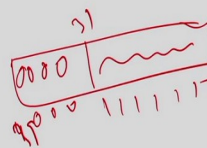
So, here is the example again you have to make sure that you take only the value that will be stored in Verilog and the same part of the value in the hardware right. So, to answer the first question that how do we represent 43 bits, I will use a 64 bit long long int how do I store a 15 bits data, I will use an int 32 bits int to store that, but I will make sure that whenever I do the assignment I will only store 15 bit. I will never store the 16 to 31 bits.

Then it will be a problem how I can do that it is very simple operation masking right. I just mask that value with a 15 bits right the example that I took its here again this say  $r\_C = r\_A * r\_B$  you can see here all are 32 bits right.

(Refer Slide Time: 41:02)

### 3. Data-width mismatch

- In Verilog, if data-width of LHS and RHS of a operation is not same then it automatically adjust.
- If we directly realized it into C language than it can cause overflow or underflow issue during assignment.
- Solution is improve the rewriting method. As shown in Fig. 2.7



```

// RTL Code
reg [20:0] r_A;
reg [20:0] r_B;
reg [31:0] r_C;
wire [31:0] w_C;
always @(posedge ap_clk) begin
    r_C <= w_C;
end
assign w_C = r_A * r_B;
// Incorrect Generated C code
unsigned long long int r_A, r_B, r_C;
r_C = r_a * r_B;
// Resolved C code
unsigned long long int r_A, r_B, r_C;
r_C = (r_A * r_B) & 64d'4294967295;

```

Fig. 2.7

So,  $r\_A$  and  $r\_B$  is 21 bits and  $r\_C$  is 32 bits ok. So, as a result. So, this result will be 42 bits right and I want to take the 32 bits of that ok.

So, what I am going to do is. So, I do multiplication. So, this will be 40 bits, but what I do here I will just put a all 1, 32 bits right. So, this is nothing, but  $2^{32} - 1$  right. So, this is this figure and these are all 0. So, this integer is nothing, but this right 2 to the power 32 minus 1 and if we do a AND of this what will happen. So, I have this 40 bits 0 to 39

and then I just try to do a AND where these are all 1 right. So, this is tool 31 bit and these are all 0.

So, if I do a AND of this what will happen this will be erased right. So, this will all become 0 and this will retain the value. So, effectively I store this data into a long long int or say int because this is 32 bits, but I remove the value which is not going to be stored in hardware. So, this is how I manage right. So, I always do a mask-appropriate one and just end and make sure that whatever the operations or the data will be stored in Verilog or in hardware that same thing will going to mimic in C.

(Refer Slide Time: 42:23)

### 4. Level-triggered operation

- When “always(\*)” is encountered in place of “always(posedge)” then such block get triggered when the status of the registers on RHS in all the operations changes.
- Solution is to perform write before any read and use the updated value.

<pre>//RTL Code always(*) begin if(cur_state == 1'd2 &amp; reg_X == 1'd1) begin b &lt;= a; end end always(*) begin if(cur_state == 1'd2 &amp; reg_X == 1'd1) begin a &lt;= c; end end assign c = a + 5;</pre>	<pre>//Incorrect C Code State_2: b_old = b; a_old = a; if(reg_X == 1) { b = a_old; } if(reg_X == 1) { a = a_old + 5; }</pre>	<pre>// Resolved C code State_2: b_old = b; a_old = a; if(reg_X == 1) { a = a_old + 5; } if(reg_X == 1) { b = a; }</pre>
---	--	--

Fig. 2.8 *Handwritten notes: "poncky" and "a = 2, b = 15, b = 2, c = old, = a + 5"*

Fig. 2.9

So, these are the things you have to take care. The last one is another very complicated thing is the level-triggered operations. The difference between level triggered and edge-triggered operation is that edge-triggered whenever the positive edge of clock comes then it happens in the level-triggered whenever the signals change right in the any RHS operations expression signal changes, value changes that operation is going to execute right.

So, I will take the same example here that earlier I have taken. So, b equal to say what is happening here, I have happening  $b = a$  and here  $c = a + b$ . So, basically,  $a = a + b$  and  $b = a$ , this two operation is happening right. So, if it is normal operation and this is always(\*) means level-triggered if it say here is the passage of clock was there then it is basically this is `a_old` right, it is not taking the new value.

But, since this is a level-triggered operation what is going to happen that whenever this value of a is changing here, this operation is going to execute and it is actually taking this new value of a, it is basically a+5 ok. So, the fix that I had did earlier will create a problem here because I use a\_old here, that a\_old will create a problem. So, what I am going to do it now. I am not going to put a\_old if it is a level-triggered operation.

It is complicated, but the idea is that if it is a level-triggered operations. I will not use the a\_old value. I am going to use 'a' itself and if it is edge-triggered I will use a\_old value right that is simple and also this ordering is important.

(Refer Slide Time: 44:00)

### 4. Level-triggered operation

- When “always(\*)” is encountered in place of “always(posed edge)” then such block get triggered when the status of the registers on RHS in all the operations changes.
- Solution is to perform write before any read and use the updated value.

<pre style="font-family: monospace; font-size: 0.9em;">//RTL Code always(*) begin if(cur_state == 1'd2 &amp; reg_X == 1'd1) begin   b &lt;= a; end end always(*) begin if(cur_state == 1'd2 &amp; reg_X == 1'd1) begin   a &lt;= c; end end assign c = a + 5;</pre> <p style="text-align: center; font-size: 0.8em;">Fig. 2.8</p>	<pre style="font-family: monospace; font-size: 0.9em;">//Incorrect C Code State_2: b_old = b; a_old = a; if(reg_X == 1) {   b = a_old; } if(reg_X == 1) {   a = a_old + 5; }</pre> <p style="text-align: center; font-size: 0.8em;">Fig. 2.9</p>	<pre style="font-family: monospace; font-size: 0.9em;">// Resolved C code State_2: b_old = b; a_old = a; if(reg_X == 1) {   a = a_old + 5; } if(reg_X == 1) {   b = a; }</pre>
---	--	--

Because if you just put b = a and then a = a + 5 in C, this a will not take this value right. So, I will just see if there is an expression and I will order these operations according to the data flow.

So, you have to do some kind of data flow analysis in the tool and you have to find out the order. So, level triggered is a very complicated scenario, but you have to manage them this is how when you generate the C code ok.



(Refer Slide Time: 44:24)

### Table of contents:

- Introduction
- Literature survey
- RTL to C conversion
- **Modelling Hardware Parallelism**
  - Loop unrolling
  - Instruction Level Pipelining
  - Task Level Pipelining
- Experiment Results
- Conclusion

So, that is all. So, now, as I mentioned that whenever you develop this RTL to C converter, you have must support all the complicated optimizations of the HLS and some of them are important is the loop unrolling, instruction-level pipelining, and task level pipelining ok. So, let us try to understand them and how whether they will create a problem in generating C from RTL.

(Refer Slide Time: 44:46)

### Loop Unrolling

- Loop unrolling transforms the loops by creating multiple copies of the loop in RTL design, which allows some or all iteration occur in parallel.
- The loop unrolling increase the concurrent data access and improve the throughput and latency of the design.
- Loop unrolling can be done fully or partial unrolling.
- In both type the unrolled FSM model is similar to the baseline FSM where the register transfers form the iterations of the loop.
- Since the overall structure of FSM remains the same, our simulation model successfully emulates the correct functionality for a loop unrolled mode.

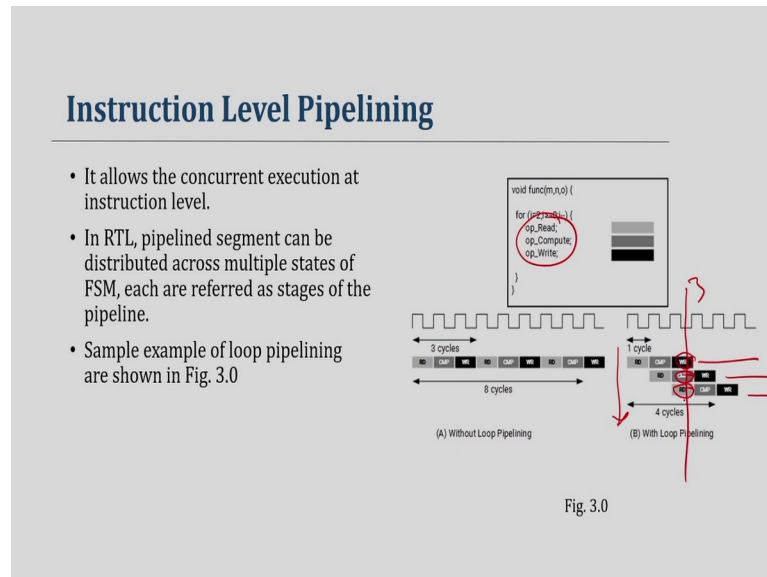
```
void my_top_function(x, y, z) {  
  for (n=3;n>=0;n--) {  
    a[n] = b[n] * c[n];  
  }  
}
```

So, loop unrolling you know that it basically you have a loop which you basically have to unroll it right. So, now, it is basically sequential state of code and you can understand

clearly that if there is a loop and you convert them it is a sequence state line of code and it will not create any problems because it can be handled by or parser.

So, this loop unrolling automatically get handled you do not have to take any special care for that next one is the Loop Pipelining.

(Refer Slide Time: 45:13)



So, Pipelining you know that whenever there is a loop you try to execute them in multiple iterations of the loop you write run in parallel right. So, this is iteration 1, this is iteration 2, this is iteration 3 and you want to execute in every clock you want to run some part of the iterations in parallel right.

So, if I consider a three-stage pipelining in this third clock the third stage of the first loop, the second stage of the second loop, and the first stage of the third iteration which going to execute right. So, the third stage of the first iteration, the second stage of the second iteration, and the first stage of the third iteration is going to be executed right this is how things will happen.

(Refer Slide Time: 45:53)

### Instruction Level Pipelining (Continued...)

- Each pipeline stages are divided into sub-stages which are usually controlled by some activation flag of that sub-stage.
- In RTL, we found that no two stage execute in parallel, but the sub-stages of stage can execute in parallel according to their respective activation flag.
- Instruction level pipelining can be applied ON-
  - At Function : pipelined different instances
  - At loop: pipelined different iteration.
- Fig. 3.1 shows sample example function warp with loop pipelining.

```
void warp (int Img[ROWS][COLS], int Out[ROWS][COLS])
{
    // a,b,c,d,e,f,i,j are defined as macros
    int x, y, destX, destY;
    loop-1.1: for (y = 0; y < 8; y += 1) {
    loop-1.2: for (x = 0; x < 8; x += 1) {
        #pragma HLS loop_flatten off
        #pragma HLS pipeline
        destX = (ax + by + cx + y + d);
        destY = (ex + fy + ix + y + j);
        Out[destY][destX] = Img[y][x];
    }
}
```

Fig. 3.1

So, now let us try to understand the corresponding controller FSM. So, let us take this example and say I have a nested loop and the inner loop is pipelined ok. I just put this pipeline and I am actually calculating some address x and y and in that x y, I am going to write some value right. So, there is two part to it. So, the address calculation x and y and writing to the memory right. So, there are these things that are happening. So, now, if your pipeline it say suppose you pipeline into two stages.

(Refer Slide Time: 46:23)

### Instruction Level Pipelining (Continued....)

The representative C-code of pipeline stage are shown in Fig. 3.2

```
// Code for other states.
STATE_S3 //pipelined state S3
x_old = x; destX_old = destX; destY_old = destY; S3_1_flag_old =
S3_1_flag; S3_2_flag_old = S3_2_flag;
//Code to update S3_1_flag and S3_2_flag
if (S3_1_flag_old == 1) { // S3-1 code block
    img_CE = 1;
    img_addr = addr_calc(x_old, x_old);
    destX = Calculate_DestX(x_old, y_old);
    destY = Calculate_DestY(x_old, y_old);
    x = x_old + 1;
}
if (S3_2_flag_old == 1) { // S3-2 code block
    if (x_old < 8) {
        out_WE = 1;
        out_Adr = addr_calc(destX_old, destY_old);
    }
}
// other code in S3
```

Fig. 3.2

So, what is going to happen?

So, this is my stage, it will create a state single state in the controller FSM and it will actually there a two-stage pipelining happening. So, stage 1 and stage 2. So, this is how it will look. So, this is the inner loop. You can see here there is a loop here and this is the outer loop right so; that means, whenever we apply a pipeline it will create a single state, a single state in the controller FSM and within the single state there will be multiple stages because it is a pipelining there will multiple stages.

And the example that I have taken it has two-stage pipelining and there are two stages. In the first stage, you calculate the address x and y and in the second stage you write into the array in that particular address ok. So, that is what is happening. You can see here the stage 1 I calculate the address and then in the next stage, I just write this. So, how I can convert this into C code. You can understand that these particular two stages happening, but they are actually executing two different iterations.

And there are some stage signals are there in the controller. So, for example, this is the flag for stage 1 and this is the flag for stage 2. So, I will just write this is my state 3. In state3 if this particular flag is 1; that means, the state2, 1 is enabled I am going to calculate this right and if this particular flag is 1, then I am going to write this ok. So, this is how I just and it is basically in the RTL it will be always like this and these signals are already available right.

So, and this is actually executing  $i+1^{\text{th}}$  iteration and this is  $i^{\text{th}}$  iteration and the corresponding data will be also stored in x and y right. So, I do not have to take much and this will be my generated RTL code, I mean generated C code where I have a state I have two substages which is nothing, but control by this flag and whenever this flag is 1 that particular stage will be executed for an iteration ok.

So, this is also will not create many problems only the basic syntax to be understood and can be managed.

(Refer Slide Time: 48:33)

## Task level pipelining

- FIFO Style and Ping-pong Style

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

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

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

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

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

The real problem or the real challenge will come whenever we handle the task level pipelining which I have already explained it in the previous class where I just talked about the data flow optimizations which is basically when you have a sequence of loops you can actually run them in parallel right.

So, for example, here I have module\_1, module\_2, module\_3, module\_4 the same example. I have explained in my previous classes and you call them in sequentially module 1 2 3 4. So, if you run this in C the module\_1 will execute first, then 2, then 3 and 4. It is basically total execution time will be the execution time of 1 2 3 4, four modules time right.

But, if you just apply this task level pipelining in this loop that I have applied here in the data flow, what the RTL tool will create it will create four modules. One for each module and then these modules will run in parallel right and the data that is generated by this will be stored in a FIFO in this FIFO because there is a channel from module\_1 to module\_2 and there is a dependency because module\_1 is creating B it is used here creating C it is used here and so on.

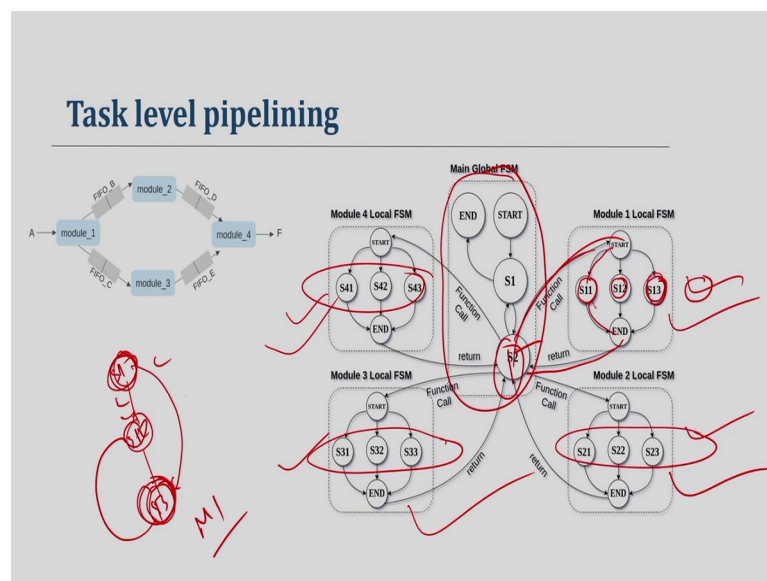
So, it will create it will put some intermediate buffer which is either FIFO or Ping-pong style and then what will happen in the hardware these four modules will run in parallel and whenever some data get produce, it will be written here and this module wait for any

data whenever the data available it will start the next iteration right so; that means, ineffectively all these models running in parallel.

And they actually have a synchronization because if there is a blocking read whenever there is no data you cannot read it and when the buffer is full you are going to write it. So, this is how they will be synchronized ok. So, now, you can understand this particular module is running in parallel all modules running in parallel and now I have to write that execution in C and I cannot just write this because this will not be cycle-accurate because as I mentioned here to execute this C code, it will total time will be a time of module\_1, time of time module\_2, time of module\_3, time of module\_4.

But, here since they are all running in parallel one clock all four modules is running. So, this is not exactly this execution right. So, I have to make sure that my generated C code is cycle-accurate otherwise I cannot debug my C code right.

(Refer Slide Time: 51:06)



So, to do that we take a very nice idea that what I am going to do I create a global main( ) ok. In the global main( ) what I am going to do I am going to call all these modules in parallel.

So, the idea is that every clock you execute one state of all four modules ok. Say suppose my module 1 state diagram is like this. So, let me explain this the basic idea here is that in every clock I have to execute one state of every module that is the idea.

So, what I am going to do it here I am going to say this is my say the state diagram of say module 1 ok. So, this is the controller FSM. So, there are three states right. So,  $S_{11}$ ,  $S_{12}$  and  $S_{13}$ . How they transits have the transition happen that is given here.

So, in the model that I am going to create, I just put these three-state in parallel. If there are four states I am going to put four states in parallel right and I always maintain what is my current state. So, whenever I just call this function the current state will execute say suppose the current state is say  $s_{11}$ . So, this current  $S_{11}$  will execute and say in the next state is under the current condition is  $S_{13}$ .

So, then the correct state will be returned into  $S_{13}$  and then it will end and it will go back to here next time whenever I call because my current state now is  $S_{13}$ , now  $S_{13}$  will execute right. So, basically, you can see here I have a created module where I just put these states of the actual FSM in parallels right and there may be many states here I just put 3, but it can be 4, 5, 10 states and I always maintain a flag which is the current state and whenever I come here I execute the current state.

So, what is going to happen here in state\_2 which is a single clock in my cycle-accurate model, I will execute one state of this module, one state of this module, and one state of this module then I will move to the next clock. So, this particular model is cycle-accurate and it actually captures the parallel execution of the whole four modules right. So, that is the idea.

(Refer Slide Time: 53:26)

**Objective -- revisited**

Our aim is to perform RTL to C reverse engineering to generate equivalent C-code.

<b>MUST satisfy:</b> <ul style="list-style-type: none"><li>fast simulation</li><li>functional correctness of the RTL</li><li>cycle accurate simulation</li><li>accurate performance estimation</li><li>generate a highly readable and debug friendly simulation code</li></ul>	<b>MUST Supports:</b> <ul style="list-style-type: none"><li>array mapping to memory modules,</li><li>non-inlined function calls,</li><li>parallel execution frameworks invoked by loop unrolling, pipelining, task level piplining etc.</li><li>accurate simulation of pipeline stalls</li></ul>
--	--

So, let us now just move to the generated let me just revisit my objectives. So, I have generated C code which will always give you faster simulation I will put this since I have taken care of all the inconsistencies all these things. So, my generated RTL will be functionally correct and then I also make sure that this is a cycle-accurate simulation because I actually keep all state information as I mentioned earlier and it actually gives you accurate performance because it actually can execute the clockwise behavior right.

And since I just put all the registered names as it is. So, it is a highly readable code and also debugs friendly because I can always analyze things in the context of Verilog. So, all the objectives that I specify is satisfied right and I also mentioned how to handle memory, how to handle function, and how to handle this loop unrolling pipelining and pipelining everything. So, all these objectives is supported by the RTL to C converter that I have discussed today. Just to conclude I will just give you the results.

(Refer Slide Time: 54:28)

## Properties of the Benchmark

- We performed experiment on several standard example programs from CHStone Benchmark Suite and Bambu HLS Tool.
- *dfadd*, *dfmul* and *dfsub* are control intensive benchmark programs with several conditional statements but no arrays. Whereas larger benchmark programs like *des*, *aes* and *mips* are data intensive programs with several arrays and function calls.

Benchmark	#Dns	#If-else	#Array	#Func	#Loop
aes.dec	949	24	15	13	13
aes.enc	979	24	15	11	13
des	354	1	9	6	11
mips	313	0	5	1	5
dfsub	955	34	1	17	0
dfadd	554	32	1	8	0
dfmul	522	28	1	17	0
arf	53	2	0	0	0
motion	52	0	0	0	0
waka	33	2	0	0	0



(Refer Slide Time: 54:41)

### Parsing Result

Benchmark	#C	#RTL	#Gen-C	Parser Runtime(s)
aes_dec	949	3154	5776	0.334
aes_enc	979	2799	4784	0.237
des	354	2330	2856	0.189
mips	313	1779	5906	0.848
dfsub	955	2203	2856	1.097
dfadd	554	1724	2132	0.646
dfmul	522	2237	2858	0.593
arf	53	351	375	0.010
motion	52	415	471	0.014
waka	33	270	281	0.007

So, we have taken various benchmarks. You can see these benchmarks are very big it has lot of if-else, arrays, functions, and loops. So, they are very complicated examples right then the parsing time we convert the RTL that is generated for this benchmark by Vivado HLS, we convert back into generated C. We can see here the number of lines in the generated C and the RTL is mostly similar.

So, this is the number of lines in C and this is the parsing time you can see here within one second we can actually generate the C from the RTL which is very fast ok.

(Refer Slide Time: 55:05)

### HLS simulation Result

- we compare our simulation Framework with Vivado HLS C-simulation, ModelSim RTL simulator, Verilator Simulator and Vivado HLS RTL Co-simulation (XSIM).

Benchmark	Simulation Time (seconds)									
	FastSim	C-sim	Speedup	RTL Cosim	Speedup	ModelSim	Speedup	Verilator	Speedup	
aes_dec	20.176	14.442	0.72x	4467	221.4x	4780	236.9x	316.4	15.7x	
aes_enc	19.32	12.656	0.66x	4389	227.2x	4693	242.8x	296.23	15.3x	
des	34.43	28.01	0.82x	34672	1007.9x	36024	1047x	723.01	21.1x	
mips	1.782	0.985	0.55x	2620	1455.5x	2885	1618.8x	20.4	11.33x	
dfsub	0.807	0.717	0.89x	8	10x	14	17.5x	4.117	5.1x	
dfadd	0.629	0.561	0.89x	7	11.20x	13	20.7x	3.92	6.2x	
dfmul	1.062	1.374	1.29x	10	9.43x	17	16x	7.165	6.8x	
arf	0.624	0.601	0.96x	6	9.7x	11	17.7x	3.93	6.33x	
motion	0.491	0.565	1.15x	6	12.24x	10	20.4x	3.3025	6.73x	
waka	0.411	0.454	1.10x	8	19.46x	11	26.77x	2.72	6.62x	
Average			0.91x		298.40x		326.45x		10.13x	

So, this is the most important result. You can see here we have taken these benchmarks and this is the time taken by our tool FastSim ok. So, which is our converter and this is the time taken by if we simulate the input C code using GCC and this will also I simulate using GCC.

And then I take this RTL co simulator of the Vivado HLS. We take model C and the verilator the tool that I talked about earlier ok and we can see here if we compare the run time, it is 91.91. So, it is bit slower 0.1 percent slower than the C simulator. So, almost we achieve the speed up the time which is similar to the C simulation right. So, it is the say 20 seconds, it is 14 seconds, It is 0.6.

But, RTL co simulations its slower we have already discussed earlier and our tool is almost 300 times faster than this RTL code simulator, with ModelSim it is 326 time faster, for Verilator it is 10 x faster. This is also generated C++ code, but this is since it does not take care of the data path and controller it is not that faster. So, the tool that we develop is as fast as the C simulator and it is much faster than any RTL simulator.

So, this RTL to C conversion is actually really helpful in having a faster simulation verification of High-Level Synthesis ok.

(Refer Slide Time: 56:30)

### Results for Loop unrolling and Pipelining

- We also compare simulation time after applying loop unrolling and pipelining optimization.

Benchmark	FastSim(s)	Simulation Speedup		
		RTL Cosim	ModelSim	Verilator
aes_dec(u)	31.6	135.13x	142.47x	12.58x
aes_dec(p)	33.57	126.57	136.64x	12.77x
aes_enc(u)	26.07	155.96x	169.12x	14.48x
aes_enc(p)	32.62	127.49x	137.12x	12.58x
des (u)	41.4	697.05x	726.77x	17.4x
des (p)	46.5	797.22x	829.87x	20.35x
<b>Average</b>		<b>339.90x</b>	<b>356.99x</b>	<b>15.03x</b>

(Refer Slide Time: 56:35)

## Results for Task level pipelining

- We apply dataflow optimization for both FIFO and PIPO style on some standard example and compare the simulation time with other simulators.

Benchmark	FastSim(s)	Simulation Speedup		
		RTL Cosim	ModelSim	Verilator
toy (pp)	23.7	20.46x	22.3x	6.06x
toy (ff)	26.6	18.73x	20.11x	5.4x
mergesort (pp)	31.2	25.2x	28.6x	8.7x
Insertionsort (ff)	39.5	21.7x	23.8x	7.6x
histogram (pp)	42	23.3x	26.2x	9.4x
FFT (pp)	153.2	117.6x	124.2x	10.1x
<b>Average</b>		<b>37.8x</b>	<b>40.9x</b>	<b>7.9x</b>

So, there are some results for unrolling pipelining, but that is not so important because the basic idea is important we also have results for task-level pipelining for various benchmarks and we have a getting a benchmark speed up there as well.

(Refer Slide Time: 56:44)

## Performance Estimation

- We analyze and compare the performance of our framework as shown in Table on the benchmark examples.

Bench	Vivado Synthesis (Clock cycles)		FastSim (Clock cycles)		RTL Cosim (Clock cycles)	
	Min	Max	Min	Max	Min	Max
aes_dec	?	?	5654	5654	5654	5654
aes_enc	?	?	3006	3006	3006	3006
des	125065	125321	125425	125427	125425	1254257
mips	?	?	3383	3683	3383	3683
dfsub	8	21	9	19	9	19
dfadd	7	20	9	18	9	18
dfmul	8	22	12	20	12	20
arf	7	7	7	7	7	7
motion	6	6	6	6	6	6
waka	2	3	3	3	3	3

So, the performance which is I just try to highlight here. We can see here this FastSim give a performance is similar to the RTL co simulations you can see here the minimum clock and maximum clock always same as the RTL co stimulation.

So, it actually gives an accurate performance estimation that I mentioned earlier and C synthesis also gives some approximate in some cases it cannot give specifically if there is a loop where the loop bound is not known, but in our case, you can always give because we run the actual data right and it is exactly same as RTL co simulations. So, it actually meets the performance estimation requirement.

(Refer Slide Time: 57:21)

### Summary

- we convert HLS generated RTL to equivalent C-code by taking advantage of RTL structure.
- The generated C-code is used for **faster simulation** which is around 300 times faster than tradition RTL simulators.
- Our tool also support advance HLS optimization such as unroll, pipeline and dataflow.
- fast simulation
- functional correctness of the RTL
- cycle accurate simulation
- accurate performance estimation
- generate a highly readable and debug friendly simulation code

So, in conclusion. So, in this class, we discussed an RTL to C conversions for High-Level Synthesis and we have shown that particular C can be used for faster simulation verification of High-Level Synthesis and our tool supports all the requirements of an RTL simulator. So, that way it can be used for simulation-based verification of High-Level Synthesis ok.

Thank you.