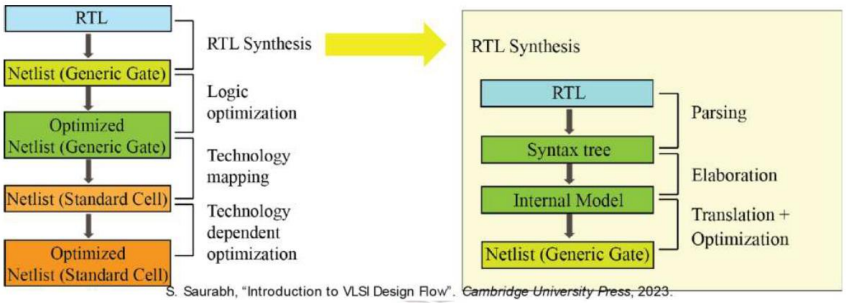
VLSI Design Flow: RTL to GDS

Dr. Sneh Saurabh
Department of Electronics and Communication Engineering
IIIT-Delhi

Lecture 15
RTL Synthesis- Part I

Hello everybody, welcome to the course VLSI Design Flow: RTL to GDS. This is the 12th lecture. In this lecture, we will be going through RTL synthesis. In the earlier lectures, we had seen that the logic synthesis consists of various smaller tasks. For example, the RTL synthesis, then logic optimization, then technology mapping and then technology dependent optimization. So, the first part of logic synthesis is RTL synthesis, and in RTL synthesis, what do we do? We translate the given Verilog code or the RTL model into a netlist consisting of generic logic.

So this is the responsibility of RTL synthesis. The RTL synthesis is also a complicated task, and we further break it down into smaller tasks. So, the smaller tasks are parsing, elaboration and then translation of various Verilog constructs in the code to the logic gates and also some optimizations which are RTL construct specific. So, in this lecture, we will be looking into the parsing and translation of some of the important Verilog constructs, and in the next lecture, we will be going through the translation of more Verilog constructs and also look into the optimization, which are specific to the RTL model.

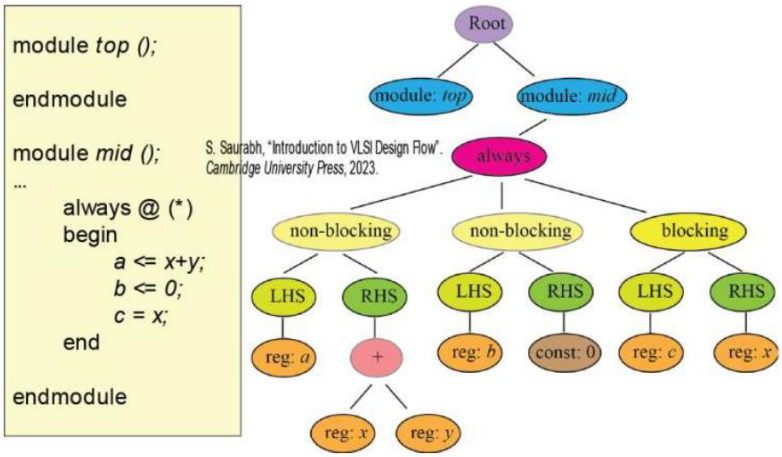


S. Saurabh, "Introduction to VLSI Design Flow", Cambridge University Press, 2023.

So, given an RTL code or a Verilog model representing an RTL, the first task that is done by the RTL synthesis tool is to read the given RTL files and populate a data structure for further processing. In the earlier lectures, we had seen that the Verilog code consists of various characters that can be grouped together into tokens. we have various types of tokens, for example, identifiers, keywords, operators, and so on. So, given an RTL file and a Verilog code, the first task that is done by the tool is known as lexical analysis. So, the lexical analysis basically breaks down the given RTL code into tokens,

various kinds of tokens, for example, keywords, identifiers, and so on. And after this lexical analysis, it checks for the grammar of the given code, whether it is following the prescribed syntax of the Verilog language and if there are say errors, then in that case the tool will report syntax error in the code, if the given code does not follow the grammar of the Verilog language.

But if the given RTL code is syntax error free, meaning that it is following the grammar correctly, then what the tool will do is that it will build a data structure which is typically a kind of syntax tree or a parse tree and this is a kind of hierarchical data structure. So by hierarchical data structure, we mean that if some object RTL construct contains some other RTL construct, then a parent-child relationship is maintained, and so on. So that makes a kind of a hierarchical data structure, or based on the parent-child relationship, the hierarchical data structure is built for the given Verilog code. So let us take an example of how it is done. So suppose we are given, or the RTL synthesis tool is given a Verilog file in which there is a module, say top, and for simplicity, we are not showing the internals of this top module, and there is another module, say mid, and it contains an always block and so on.

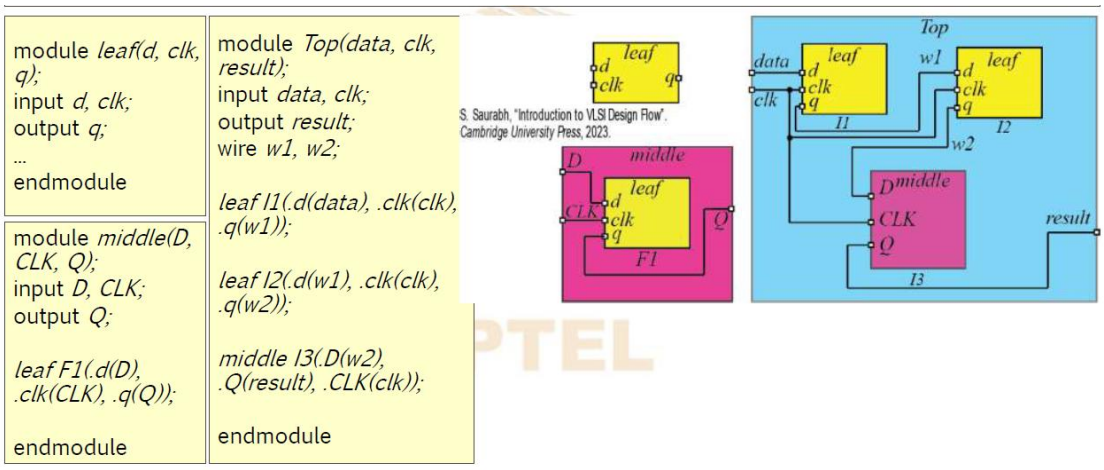


Now, for this code, what will be the syntax tree, or how will that syntax tree look like? So it will be something like this. So there will be at the top, there is a root, and in Verilog, the module is the topmost entity of design. So the root will have as children various modules. So, in this case, there are two modules, top and mid, so those will become the children of the root, and then, in mid, in the given code we have an always block. So, the always block is contained within mid, so the always block will be a child of the module mid, and then within the always block, there are three statements.

We have three statements, so each statement will form one child. So there are two non-blocking statements, these two are non-blocking statements. So, two child will be created corresponding to those non-blocking statements, and one child will be created

corresponding to the blocking statement. Now, each statement has an LHS and RHS. So, this non-blocking statement will contain the LHS part, the RHS part, and so on.

So this is how the given Verilog code will be passed and a data structure will be built, a hierarchical data structure and then once this hierarchical data structure or parse tree is built then the RTL synthesis tool goes to the next step and what is the next step? The next step is called elaboration. So what is done in elaboration? So, in elaboration, the RTL synthesis tool checks whether the connection among the RTL-specific components is legitimate. So, we will understand what we mean by legitimate connections, and if it is legitimate, then during elaboration, the connections are made in the internal model that is used to represent the given Verilog code. So let us take an example. Suppose there was a design in which we have say one module leaf, this is a leaf level module and then there is another module middle, which contains an instance of leaf.



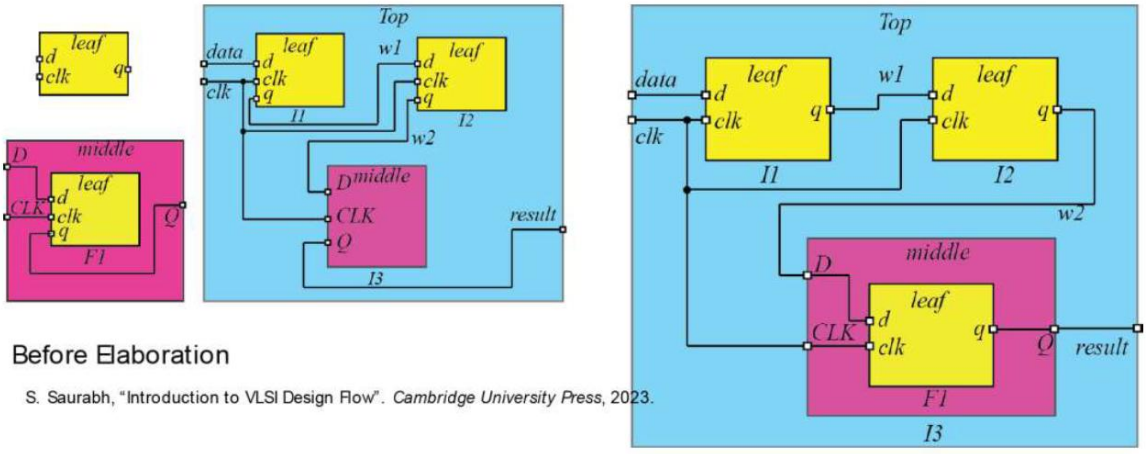
So this is an instance, instantiation of leaf. So we say that for this instance F1, F1 is the name of the instance, the master module is leaf because the name here is referring to this other module. So the master module of this instance F1 is leaf, and then there is another module, say Top, and inside Top, we have three instances, two instances of leaf. So I1 and I2 are the instances with master module as leaf and I3 is an instance whose master module is middle. So, this is the given RTL code.

If we give this code to the RTL synthesis tool, then during elaboration, the linkages between the instances and the master modules are made, and in this instantiation, if there are some problems of connections, then those problems are reported as elaboration errors. For example, if we say that the master module is leaf and in leaf, we have three ports, d, clock, and q, and here we have written d, clk, and q. So here there is no problem, but suppose we made an error, and instead of writing d here, we wrote here d1. So this kind of error will be reported by the RTL synthesis tool during elaboration. So when initially the RTL code is read at that time no linkage is created between the instances and their master module.

So, during the initial stages, the code is read module by module. So at that time, the tool does not know that what is the direction of the input, this pin d, clk and q and so on. So the tool can actually make some simplistic assumption initially. So for the leaf level module, leaf, the tool can understand that the direction of port d is input, for the clk it is input, for q it is output. But when it creates the module middle, it will process the instance F1.

So it will create an instance F1 but while processing this module, the tool will, before establishing the linkages between the instance and the master module it does not know that what is the direction of this pin d, this pin clk and this pin q. So, the tool can make some simplistic assumptions like treating all of them as input pins. So similarly in the module top there are three instances, two of leaf and one of middle. So when the tool is analyzing this particular module Top at that time it does not know that what are the directions of these pins d, clk, q. And it can make a simplistic assumption that all pins are input pins.

And after elaboration what it will do is that it will create linkages between this instance and the master module, this instance and the master module, and this instance and the master module. And then a design hierarchy will be built and the connections will also be created. So let us understand that after elaboration, what will happen. So, this was the view of the design when no elaboration was done. So, in this case we see that the modules are separate.



So the modules and instances are totally separate. But after elaboration, those will be linked together. So now at the top level, so this is the top level module. In this top level module, we have two instances of leaf with name I1 and I2. Now, the directions of the pins are inferred from looking into this master module of leaf, and the direction of pin d will be taken as input; for pin clk, it will be taken as input; and for pin q, it will be taken

as output. Similarly, the correct direction will be inferred, and then a design hierarchy will be built, and the connections will be made.

Now, during this stage, there can be errors, elaboration errors that the tool can report. Now, when can the error be? So there are some common situations. Suppose while we are running the logic synthesis tool or RTL synthesis tool and these three modules were in separate Verilog file and by mistake we gave only these two files and forgot to give this file. In that case, the tool will not be able to establish what this module leaf is because the master module of the instance F1 is leaf, and we have not given the module corresponding to the leaf by mistake.

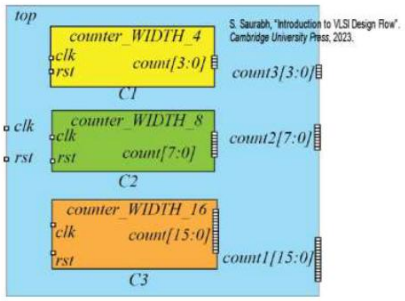
So if we make such kind of mistake, then the tool will not be able to establish the linkages between the instances and the master module, and as a result, what will happen is that the tool may report an error. It will report an error that there is no reference to the module leaf in the RTL files, and therefore, it is assuming the leaf to be a black box. Now this is a situation which we encounter commonly in logic synthesis and it means that probably we have not given the reference or we have not given the module which contains the description of the module leaf. So that is one mistake that can be made or that can be reported during elaboration that is the black boxes in our design. So typically, there should not be a black box in our design; otherwise, even the direction of those pins cannot be inferred for their instantiation, and then the tool can behave unexpectedly.

So, we should look into the warnings or errors corresponding to the black box in our design carefully. Other kind of errors can occur with respect to the port names. For example, if we make an error in the name of the ports while instantiating, then that problem will be reported during elaboration, or there is a mismatch in the width of the buses and the pins in the instantiation then the tool can report error. Another responsibility of elaboration is that it processes parameterized module. So, in earlier lectures, we discussed that Verilog supports parameterized modules, meaning that we can have parameters in our module, and based on the value of the parameter, the module can have different interfaces. For example, the width of a bus or a port can change based on the parameter, and the parameter's value can be different in different instances.

So in that case what the RTL synthesis tool will typically do, it will create separate modules with different interfaces for each distinct set of parameters. Now, what it means, let us look into an example, and then we will understand. So suppose there was a parameterized module counter with a parameter WIDTH, with a default value of 4, and there is an output port with the name count, and its size depends on the value of this parameter WIDTH. So if the value is the default value, then the output port count will be treated as count[3:0]; it will be 4 bits wide.

```
module counter(clk, rst,
count);
        parameter WIDTH=4;
        input clk, rst;
        output [WIDTH-1:0]count;
        reg [WIDTH-1:0]count;
...
endmodule
```

```
module top(clk, rst, count1, count2,
count3);
        input clk, rst;
        output [15:0]count1;
        output [7:0]count2;
        output [3:0]count3;

        counter C1(clk, rst, count3);
        counter #(8)  C2(clk, rst, count2);
        counter #(.WIDTH(16)) C3(clk, rst,
        count1);

endmodule
```



S. Saurabh, "Introduction to VLSI Design Flow". *Cambridge University Press*, 2023.

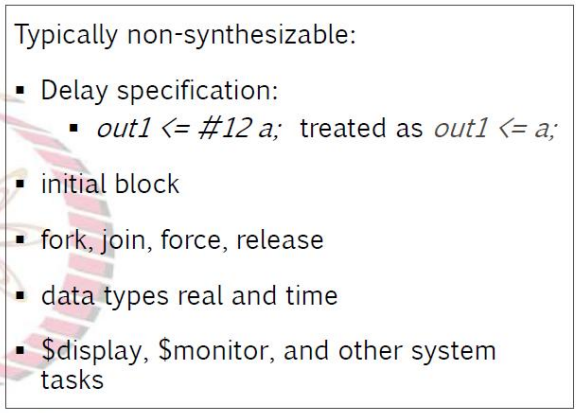
Now, during instantiation, we can change the value of the parameters. For example, in this instance, C1, we are not changing. So in this case the default value of 4 will be used for count3 for the instance C1. For the another instance, C2, we are saying that instead of 4, use a value of WIDTH as 8. So in this case, the width of the bus, width of the port count will be 8 instead of 4.

And similarly in the third instance for the C3 we want the WIDTH to be 16. Now, in this case, what the tool will do is internally create three different modules: one for width 4, another for width 8, and the third one for width 16. So how will the design look like? It will look like something like this. So for the instance C1, the count will take a value of [3:0] or it will take a WIDTH of 4, for C2, 8 and for C3, it will be taking 16. Now, what will be the name of this new module that will be created during elaboration?

So, it is tool-specific,. Typically it will use a combination of the name of the parameter and its value. So it may be say the master module name initially was a counter. So, in this case, during elaboration, the new module will be created, and its name can be counter _WIDTH_4 for this module; for the other, it can be counter_WIDTH_8 to denote that the value of the WIDTH is 8, and so on. But note that the name of the master module that will be created by the tool is tool specific, the tool is free to choose any. Before going through various RTL constructs and understanding how they are synthesized to logic gates first let us understand that which RTL constructs or Verilog constructs are synthesizable and which all are not synthesizable.

Why? Because some Verilog constructs are there which support other design task such as functional verification and those are not meant to be synthesized. So, a given RTL synthesis tool may not support all synthesizable Verilog constructs. This is also a point that should be noted that the language as such does not say that what is synthesizable and what is not synthesizable. So it is tool specific that which RTL construct is supported by that tool for synthesis and which constructs are not supported by the tool. And therefore, as a designer we should be aware of the Verilog constructs which are supported by the synthesis tool that we are using and then write out Verilog code taking that into account.

So there are some RTL constructs which are typically not supported by RTL synthesis tool. So let us understand or take a look at them. So the first thing we should note is the delay specification, for example if we write a statement out1 which is assigned to a after a delay of, say, 12 time units. So this kind of delay specifications are not supported by synthesis. So, these delay specifications are typically useful for simulations.

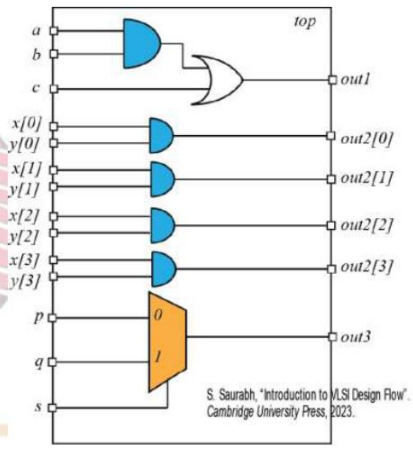


In the simulations, we want to understand the Verilog code, or we want to verify whether our RTL model behaves correctly when there is a delay of, say, 12 time units. So the #delay kind of constructs if it exists in the RTL code it should be interpreted as that this means this delay is meant to simulate and see that what happens to our RTL model if this much delay happens. But it is not meant to say to the synthesis tool that insert a delay element which has a value of 12 time units. So the delay typically what the synthesis tool will do is that in these cases the delay specification for example, the #12 in this case will be totally ignored. So the RTL synthesis tool will treat the statement as out1 is assigned to a in a non-blocking manner.

So the #12 will be totally ignored. So, there are other RTL constructs that are not supported in synthesis, for example, initial blocks. Initial blocks are useful in writing test benches that we have seen earlier. And then there are fork, join, force, release and data types real and time these are not supported for synthesis. And there are system tasks, for example, $display, $monitor, and others, that are not supported by the RTL synthesis tool. So, these system tasks are useful in verifying our design or doing functional verification or simulation. Now let us look into the RTL constructs which are synthesizable and how they will be synthesized.

- Combinational circuit elements inferred

```
module top(a, b, c, p, q, s, x, y, out1, out2, out3);
    input a, b, c, p, q, s;
    input [3:0]x, y;
    output out1;
    output [3:0]out2;
    output out3;

    assign out1 = (a & b) | c; // logic network
    assign out2 = (x & y); // bitwise logic network
    assign out3 = (s) ? q : p; // multiplexer

endmodule
```

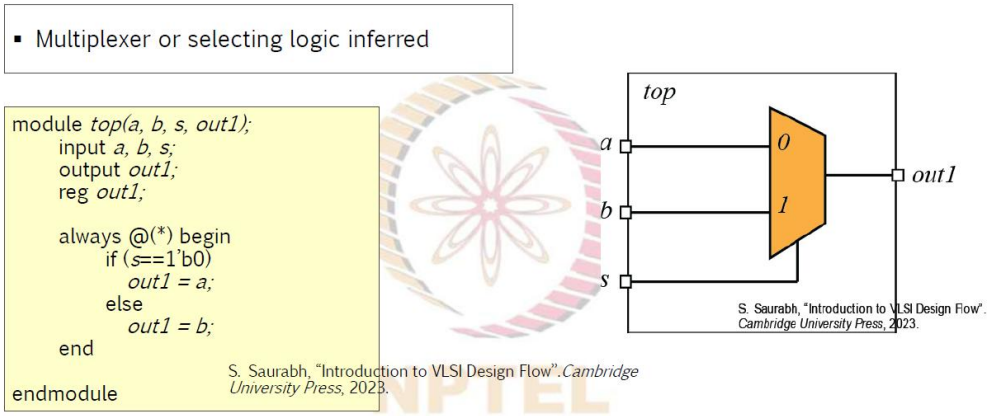


S. Saurabh, "Introduction to VLSI Design Flow". *Cambridge University Press*, 2023.

So first, let us look into the assign statement. So how is a continuous assignment or assign statement synthesized if it exists in the RTL code. So, corresponding to the assign statement, a combinational circuit will be inferred, and what combinational circuit will be inferred will depend on what is on the RHS of the assigned statement. So, let us take an example and understand this. Suppose there is a module top and in which we have say three assign statements.

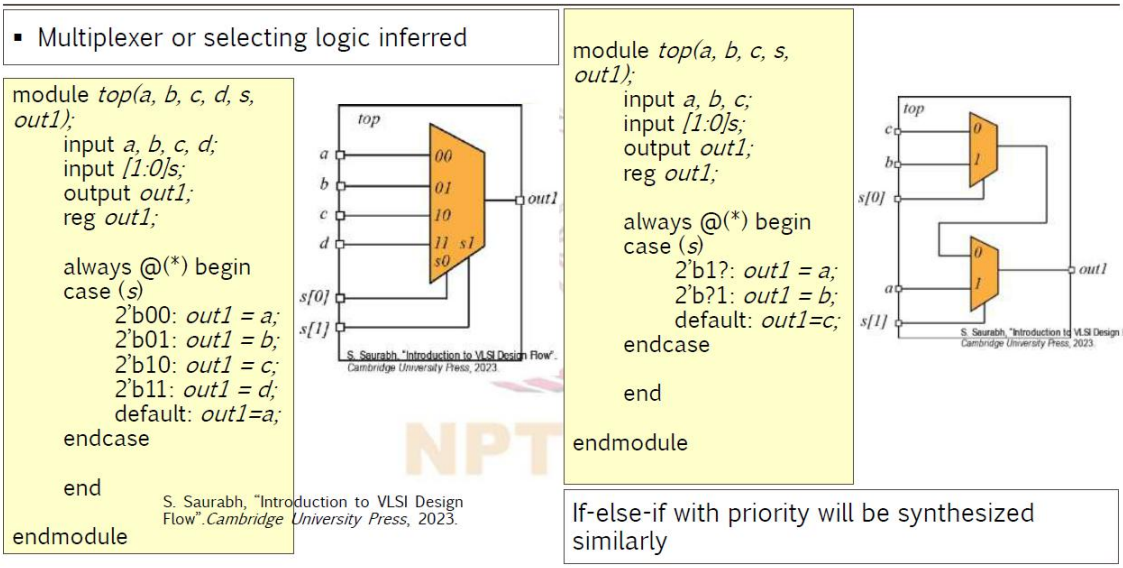
So the first assign statement says out1 is equal to (a & b) | c. So this is the RHS of the assign statement and the logic will or the RTL synthesis tool will instantiate logic gates corresponding to the RHS. So it will be something like this. So for the output port out1 an AND and an OR gate will be instantiated. So gates will be instantiated corresponding to these logical operators. Similarly, for out2, since out2 is of size four, it is a bus of size four.

So, in this case x and y are also of size four, so when we assign out2 is equal to x & y, there will be four AND gates that will be instantiated. So x[0], y[0] will be ANDed and out2[0] will be driven by that and so on. And for the third one out3, we have a ternary operator. So when s is true, then out3 takes a value q; otherwise, it takes a value of p. So this is a description corresponding to a multiplexer.



So, the tool will instantiate a multiplexer in which s is appearing on the select line, and p and q will be on the data line. Now let us look into how the statement if-else will be synthesized by the RTL synthesis tool. Now, if-else is a kind of a multiplexer kind of thing, meaning that if something happens, then one line will be selected; otherwise, some other line will be selected. So, corresponding to the if-else statement, a multiplexer will be instantiated in our design, and the select line of the multiplexer will correspond to the condition that appears in the if condition.

So, let us take an example and understand. So let us look into this always block, in this always block there is an if-else statement. Now if (s==1'b0), i.e., s=0 then out1=a, else out1=b. So, corresponding to this, a multiplexer will be created. So, out1 will be driven by this multiplexer. On the select line we will have s why we will have s because it is coming in the if condition. So here we have the select line, and when a select line takes a value of 0, then a is driven to the out1 port, and as represented in this RTL code, and if s takes a value of 1, then b will be driven to the output port.



So this is how the if-else kind of RTL construct will be synthesized by the RTL synthesis tool. Now, how will the case statement be synthesized? So the case statement will also be synthesized into a multiplexer or a select logic is inferred. So the size of this multiplexer will depend on what is the complexity of the case or the signal appearing in the case statement which is selecting various values. For example, suppose we have a case statement like this. So, in this case statement, selection is based on the signal s, and what is the size of s? The size of s is of 2 bits.

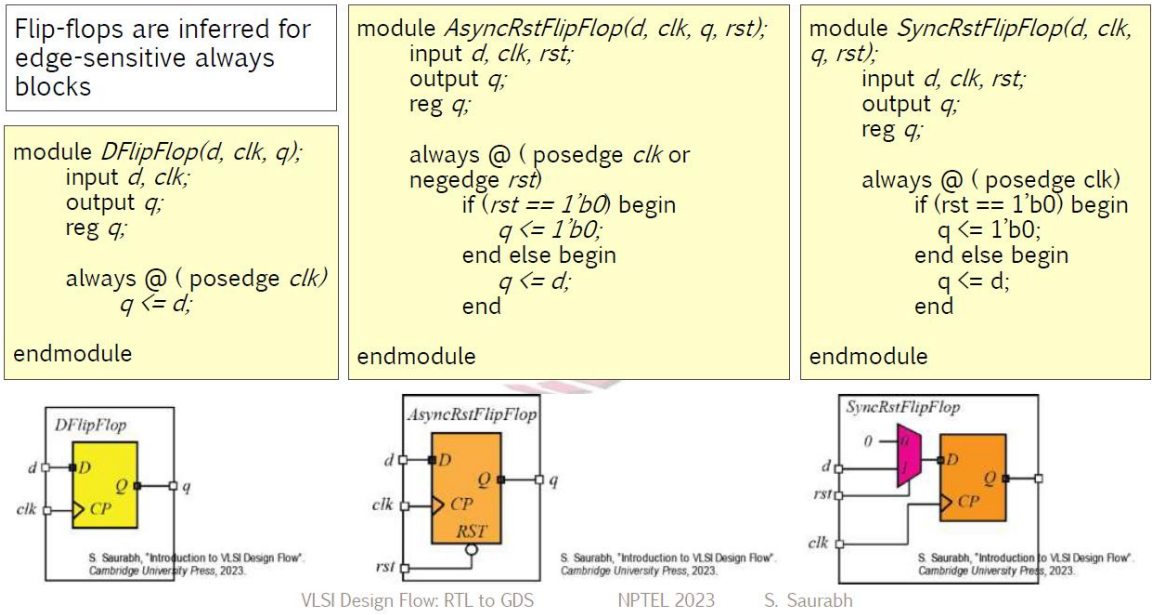
So, in the multiplexer, that will have 2 lines or the 2 bits on the select line. So this kind of case statement will basically translate into a 4 to 1 multiplexer because there are 2 select lines s[0] and s[1] and if these 2 take a value of 00 then line a will be selected. If they take the value of this one as 1 and this one is 0, then b will be selected, and so on. So, this is how the case statement will be selected. Now, in a case statement, Verilog supports that there can be multiple matches.

So if we write, say, '?' or don't care conditions, in that case, there can be multiple matches, and in that case, what will the behaviour of the Verilog code? It will be that whatever matches first that will be selected by the case statement is what is defined by the language, and therefore, if there are multiple matches in the case statement, then the tool will infer or instantiate a kind of priority logic. So, let us take an example and understand it. Now suppose the code is like this, the case statement is like this. Now s is again of 2 bit it is a 2-bit line. Now, how many possibilities are there for this signal to take?

So it can take 00, 01, 10 and 11. So what we are saying is that or what this description says that if '1?', means that this '?' can take any value 0/1, out1 takes a value of a meaning that if it is 11 or 10 in both these cases, a value of a will be driven to the out1. And then what the second line says that if it matches either with 01 or 11 out1 should be driven to b. But note that the case 11 is already covered in the first case. So it cannot match in the second case. So only the 01 case will take a value of b, and then the default is 00, and in the 00 case, out1 will get a value of c. So now we have to instantiate logic gates such that this kind of priority is maintained in the logic structure or in the implementation of the circuit. So, let us look into how the instantiation will be done. So it will be something like this. So what we have here is that if s[1] takes a value of 1, then, out1 gets a value of a, it will get the value of a, this line will be totally ignored.

So what it means is that if the value of s is 10 or 11, in both cases, out 1 will get a value of a. So this is what we wanted our priority to be. Now, if it does not match, it does not get the first, or it does not match the first statement or first condition '1?', then it goes to the next statement. And in that next statement what it does is that it looks for 1 in the s[0] or the lower bit. So if s[0] is 1 then b will be selected and it will be going to out1 and if s[0] is 0 then c will be selected and to the out1.

So the required priority that we wanted to have as described in this RTL code that is maintained by this circuit. So, this kind of priority can also be enforced by an if-else-if kind of construct. In that case, also, the priority-based logic gate structure will be implemented in the netlist generated by the RTL synthesis tool. Now let let us take a few examples of an always block and then understand that how always block will be synthesized. So first, let us take a few examples for the case when the always block is edge sensitive.



So let us consider this module DFlipFlop. So in this module, we have an always block that is sensitive to the clock signal clk, and whenever a positive edge to the clock signal appears, then whatever is in there on the d that appears on the q or that is assigned to q otherwise q retains its old value. So this is a description of a D-type flip flop, and in this module, the RTL synthesis tool will basically instantiate a D flip flop as shown here. So whenever a clock's positive edge appears at the clock, whatever is in the data will be propagated to the q pin, and it will appear at the port q. Now let us take another example in which the always block is sensitive to both clock signal and the reset signal and it is sensitive to posedge clock and negedge reset. So when rst takes a value of 0, then q is assigned a value of 0; otherwise, q is assigned a value of d.

So note in this case that this clk and rst these two signals can appear or be independent of each other, and therefore, this reset signal is a kind of an asynchronous reset, meaning that it can come any time of the clock period and it can reset the flip flop. So corresponding to this description, flip flop will be instantiated in which there will be an asynchronous reset pin, and reset is active when it takes a value 0 and, therefore, will have a bubble here, meaning that when reset is equal to 0 then q will take a value of 0.

Now let us take another example of an always block, which is sensitive just to the clock signal. So this always block will be triggered only when posedge clock appears. Now when it enters, it looks for first the signal reset, and if reset is equal to 0, then q becomes 0; otherwise, q takes a value of d.

So once the clock edge comes, then it looks whether that what is the value of reset. If reset is equal to 0, then q becomes 0. So it is a kind of a synchronous reset kind of thing. So we can look into this part and consider it as a multiplexer and therefore, a circuit like this can be inferred by the tool. So there is a flip flop, and the d input that is coming to the flip flop is either 0 or d because this if-else statement corresponds to a multiplexer that we have seen earlier. So if reset is equal to 0, then this line will be selected, and on this line will have a 0 constant value assigned to this multiplexer.

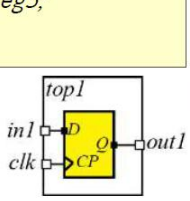
And then if the reset value is 1, then whatever is at d or the d port that appears at the D pin of the flip flop. So now, this is how this module or this RTL construct will be synthesized . Now the RTL synthesis tool can further optimize it for example, instead of using a multiplexer it can use an AND gate or so on. So, whatever we are discussing in this lecture, the tool may instantiate some other logic gates also. So you should note this point that what we are showing in this slide the tool may instantiate some other logic gates also but whatever the tool will instantiate it will be functionally equivalent to what we are discussing in this or showing in this slide.

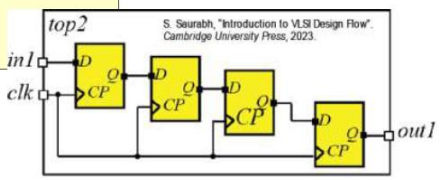
So, instead of a multiplexer, the tool can always instantiate an AND and OR gate, and so the exact implementation of the RTL construct is dependent on the tool. So, for a given Boolean function, the tool can implement in many different ways, but based on the meaning of the Verilog construct, some of the tools will always generate a circuit that is functionally equivalent to what we are discussing in this lecture. Now, the always block can be synthesized in different ways depending on whether we are assigning a signal in a blocking manner or non-blocking manner. So, let us take an example and understand it. So another important point that should be noted is that the Verilog language as such was invented for the primary purpose of simulation or verification.

So you might remember that Verilog language, or the word Verilog, was derived from verification and logic. So the initial purpose why Verilog was invented was for verification using simulation. So the syntax and semantics of simulation was very well defined initially and then later on the Verilog language was used for the logic synthesis, RTL synthesis purpose. So later on when RTL synthesis tools were developed they used the same syntax or they followed the semantics of the simulation that was being followed, in the synthesis also. So the handling of blocking and non-blocking assignment in always block is done differently.

```
module top1(in1, clk, out1);
      input in1, clk;
      output out1;
      reg reg1, reg2, reg3, out1;

      always @ ( posedge clk)
      begin
            reg1 = in1;
            reg2 = reg1;
            reg3 = reg2;
            out1 = reg3;
      end
endmodule
```

```
module top2(in1, clk, out1);
      input in1, clk;
      output out1;
      reg reg1, reg2, reg3, out1;

      always @ ( posedge clk)
      begin
            reg1 <= in1;
            reg2 <= reg1;
            reg3 <= reg2;
            out1 <= reg3;
      end
endmodule
```





How is it done? It is done in the manner such that it is consistent with the simulation that is what is an important point to note. So once we understand that how things will be simulated then the synthesis tries to mimic that thing. So, let us take an example and understand it. So suppose there is an always block in which there is a clock signal. This always block is triggered whenever a posedge appears at the clock signal. So now, in this always block, all the assignments are in a blocking manner.

So it is similar to a programming language. The first statement will be executed, then the second, then the third, and the fourth in sequence, and when one statement is executed, the others are blocked, and so on. So in this case whenever the clock signal, posedge clock appears then whatever is at in1 will be assigned to reg1, and then whatever is in at reg1 will get assigned to reg2, and then whatever is at reg2 will get assigned to reg3, and whatever is at reg3 that gets assigned to out1. So, all these assignments happen at a single clock edge. So in effect whenever the clock edge, posedge clock edge appears whatever was at in1 that appears at out1. So in effect this reg1, reg2, reg3 these are redundant. So, in effect this always block is doing whatever is at in1 that is passed to out1 whenever the clock edge appears because all these statements are executed in sequence, and as a result, finally, what happens is that out1 gets a value of in1.

So in this case the tool will instantiate only one flip flop and the D pin will be connected to in1 and the Q pin will be connected to out1. So this is how the tool will synthesize this always block. Now let us take another example in this example, everything is the same; the only difference is that we are using a non-blocking assignment. Now remember from earlier discussion that the non-blocking assignment is executed in two steps. In the first step, whatever is on the RHS that is evaluated and scheduled to be assigned to the LHS later on at the end of the simulation. So what happens in this case whenever clock edge appears positive clock edge appears so whatever was that value, current value of in1 that

is not assigned to reg1 but that is scheduled to be assigned to reg1 that is an important point.

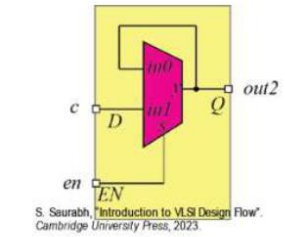
The value of reg1 is still the old value but what the reg1 value will be assigned later on it is the value of in1. And then what happens to the reg2, the current value of reg1 it is not yet updated to in1. So whatever is the current value of reg1 that is scheduled to be assigned to reg2 but reg2 value is still not updated. And in the next step whatever is the current value of reg2 that is assigned to reg3 and the reg3 value is not yet updated. And in the third case whatever is the value of, current value of reg3, that will be assigned to out1 but out1 is not yet updated.

Now at the end of the simulation, all these updates happen, so in effect, what happens is that this kind of structure will be synthesized to a shift register kind of thing. So this register will correspond to reg1, this register will correspond to reg2, this one will correspond to reg3 and this one will correspond to out1. So four registers will be created in this case, and whatever the value of in1 is, that will be assigned to reg1. But it will be assigned when the clock edge comes and what was the current value of reg1 that will be assigned to reg2 and so on.

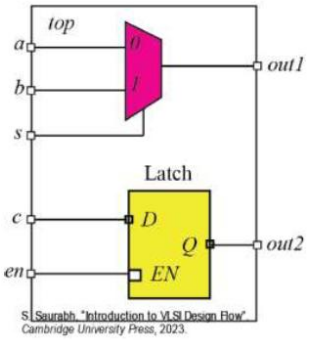
So, this is how it will be synthesized. Now let us look at when an always block is level sensitive then how it is synthesized. So when the block does not contain edges in the sensitivity list, then either combinational circuit elements or latches will be inferred. When the value of the variable is updated in every possible path in the conditional branches within an always block, then combinational circuit elements will be inferred. And if there is some path in the conditional branches where the variable retains its old value, then a latch will be inferred.

```
module top(a, b, c, s, en, out1, out2);
      input a, b, c, s, en;
      output out1, out2;
      reg out1, out2;

      always @(*) begin
            if (s==1'b0)
               out1 = a;
            else
               out1 = b;
      end

      always @(*) begin
            if (en==1'b1)
               out2 = c;
      end
endmodule
```

```
      always @(*) begin
            if (en==1'b1) out2 = c;
            else out2 = out2;
      end
```



S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.



S. Saurabh, "Introduction to VLSI Design Flow". Cambridge University Press, 2023.

So, let us take an example and understand what it means. So let us consider this module top in which there are two always block. In the first always block we see that out1 variable is assigned a value of a when s takes a value 0 and assigned a value b when s takes a value 1. So s can take either 0 value or 1 value and in both cases either a or b is being assigned to out1. So out1 is assigned in all conditional branches within the always block and therefore, in this case combinational circuit element and specifically a multiplexer will be inferred.

Now let us look into the other always block. Now, in the other, always block if the enable signal is taking a value of 1, then out2 takes a value of c, and if it is 0, then what happens is not specified. So, in that case, if enable takes a value of 0, then out2 retains its old value because the value of out2 is not being refreshed or being written in that case when enable is equal to 0. So it means that whenever this block is triggered and the value of the enable signal is 1, then out2 will get a value of c; otherwise, it will retain its old value. So we can consider this always block and the one shown here as equivalent. So we write if enable = 1'b1 then out2 takes a value of c else out2 is equal to out2 it is retaining its value and if we draw a circuit for this, again it is a if-else kind of structure and therefore a multiplexer will be inferred, the multiplexer will look like something like this.

So whenever enable is 1, whatever is at the c line will go to out2; otherwise, it will take the old value, and therefore, we have made a connection, a feedback. Now the multiplexer with a feedback is similar to a latch. So this structure, this complete structure is nothing but a latch, a D latch. Remember that what is a D latch. So, just to recap, what is a D latch? D latch has a pin as D and enable and a third pin as Q. So whenever enable is high, whatever is the value at D goes to Q, and whenever the enable is 0, then it holds the previous value.

So, this description shown in this figure is the same. If enable is equal to 1, then whatever is c appears at out2; otherwise, it retains the old value, and there is feedback. So, corresponding to this module top, the two instances will be created first for the always block in which out1 is being written in all conditional branches, and therefore, a multiplexer or a combinational circuit element will be purely combinational circuit element will be inferred, and for the second always block a D latch will be inferred. So sometimes we actually want the D latch to be or latch to be inferred because of the always block, but many times what happens is that latches get inferred because of wrong RTL coding. What happens is that latches often get inferred due to incorrect modeling of the combinational block and what mistake we can make, we can inadvertently miss updating value in one of the branches of the case statement. Suppose we are writing a code for an FSM or a state or some RTL code in which there is a complicated case statement.

```
module top(in1, out1);
    input [0:1]in1;
    output out1;
    reg out1;

    always @* begin
        case(in1[0:1])
            2'b00: out1 = 1'b0;
            2'b01: out1 = 1'b1;
            2'b10: out1 = 1'b1;
        endcase
    end
endmodule
```

S. Saurabh, "Introduction to VLSI Design Flow". *Cambridge University Press*, 2023.

So a case statement in which the select line is say of 8 bits and based on the value of the select line various operations are being done and in we wanted this case statement or case block to be inferred to a purely combinational circuit element but because we missed some of the case statement a latch will be inferred because for the cases where we have missed the assigning value to a variable it implies that it will retain the old value and therefore, a latch will be inferred. For example, consider this code. In this code we have in1 is on the select line for the case statement. Now it has got 2 bits. So, there are 4 possibilities 00, 01, 10 and 11 for the signal in1. Now, for 00, we are writing the value of out1. For 01, we are writing the value of out1; and for 10, we are writing a value of out1, but for case 11, we have missed writing a value that may be a mistake, an inadvertent mistake while writing the RTL code and therefore, in this case a latch will get inferred.

So, to avoid such problems, what we can do is that we use default statements. So, whenever there are complicated cases at the end of the case statement we also write default and for example, and then we write out1 is equal to say 1'b0. So, if we use a default statement, then for the case nothing matches in the earlier branches, it will fall back to default, and the variable will be updated or refreshed therefore, a combinational circuit element will be inferred rather than a latch. Other thing we can do is that at the beginning of always block we assign the value of the signals to a default value for example, we can write out1 is equal to 1'b0 before the case statement. So, whenever the always block is triggered out1 will be assigned one value, of course it will be a reassigned later on down the code, but in the conditional branches if we are missing writing value to the signal out1. Then this value of out1 that we have written at the top of the always block that will come into action and a combinational circuit element will be inferred rather than a latch.

So, to know more into what we have discussed in this lecture you can refer to this book and just to summarize what we have done in this lecture. So, in this lecture we have

looked into the RTL synthesis. So, we saw that RTL synthesis consist of various task. For example, first, the parsing of the RTL code is done, then the elaboration is done, then we translate the Verilog construct to the corresponding logic gates, and then optimizations start. So, in the next lecture, we will be looking at the translation of some more Verilog constructs, and we will also look into the optimization that is RTL specific. Thank you very much.