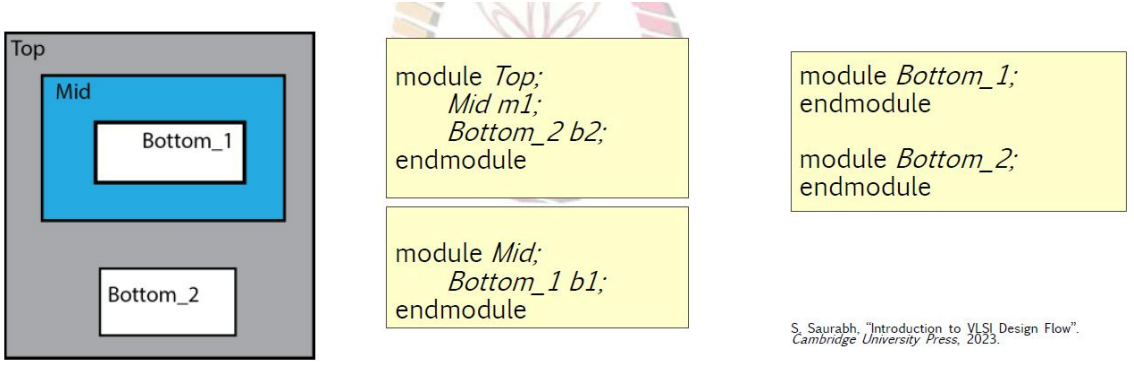
VLSI Design Flow: RTL to GDS

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

Lecture 12
Hardware Modeling: Introduction to Verilog-II

Hello everyone, welcome to the course VLSI Design Flow: RTL to GDS. This is the 10th lecture. In the previous lecture, we looked into a few constructs of the Verilog language that will help us in modeling an RTL or writing a behavioral description of our design. In this lecture, we will be continuing with more such constructs that will help us in writing a Verilog model. So, modules are the basic building blocks for Verilog design. It starts with the keyword module and ends with endmodule.

So we have a module, within this, we write everything.. So within module and endmodule, we write our the code for a module. So this is the basic, there can be many modules in our design and modules are the most important building blocks of a Verilog design and modules are instantiated inside another modules to create a design hierarchy. So, instantiation of a module means using that module in another higher-level module.



Let us take an example. For example, if we want to create a design hierarchy like this, there is a Top, then inside this, we have a Mid, and then also we have a Bottom, and inside this mid, we have another Bottom. So we can create this by writing modules, we can write module top. And inside this module we have two instances, Mid of this one and Bottom_2. So, these two are the instances within the module top.

So Mid is the name of the module, which is being instantiated, instantiate is kind of means copying everything whatever is there in another module into this module. And m1 is a name of the instance, this is an arbitrary name that we can give. And then the other

modules for example there is another module Mid and inside this there is an instance Bottom_1 whose name is b1.

And Bottom_1 and Bottom_2 are there which do not contain any other modules. So those are leaf level modules. So, in this manner, we can have a module and several instantiations within that, and the instance of one module can refer to another module, and the other module can again have other instances, and so on. This way, we can create a design hierarchy in the form of a tree. Now let us look at how we can define ports in a module using Verilog.

So ports are the interfaces of a module and it allows communications between the module and the environment. So ports can be declared with the keyword input, output, and inout in a Verilog module. It can be scalar or vector similar to net and reg. So, for example, in this case, we are saying that there is a module top, and the ports defined, and their directions are defined. Some are scalar ports, and some are vectors.

```
module top(clock, reset, test_port, data,
counter_out);

        input clock;
        input reset;
        inout test_port;
        input [3:0]data;
        output [3:0]counter_out;

endmodule
```

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

Now, let us understand how we can define instances in Verilog and their connections. So, connections for an instance can be specified in two ways. The first one is by order. So it is a kind of implicit connection.

So, in this case, ports connected automatically in the instantiation based on the order specified in the module declaration. So it means that suppose there is a module mid. It has got ports clock, d_in, d_out. We want to instantiate this mid in the module top. So what we can do is we can write mid, the name of the module, instance name, which is arbitrary, say m1, and then we define the nets which are going to connect each of the ports.

```
module mid(clock, d_in, d_out);
        input clock;
        input d_in;
        output d_out;
endmodule
```

```
module top(c, p_in, p_out);

      mid m1(c, p_in, p_out);

endmodule
```

So c will be connected to clock, p_in will be connected to d_in, p_out will be connected to d_out. So, based on the order, each net in the instantiation will get connected to the port in the corresponding instance or pin in the corresponding instance. So ports connected automatically in the instantiation based on order. For example, if I change the order here to p_in and then c, then p_in will be connected to the clock, and c will be connected to d_in. So, depending on the order, the connections will be made.

So this is difficult to debug. It is difficult because while instantiating, there is no information about which net is connected to which port or pin in the master module of the instance. So, in that case, it will be difficult to debug if there are some problems in the connections. And here it is only three ports. But in real designs, you can have a hundred ports in a module, and then it will be really difficult to understand which net is going to which port in the instance if we do instantiation in an implicit manner or by order.

So, the other way to do or define connections for instances is explicit connection. So in this case ports are connected by explicitly giving names, for this is an example. In this case, what we say that while instantiating, I say .clock. .clock refers to the port in the master module of mid. So this is the mid, this is an instance, and its master module is here.

```
module top(c, p_in, p_out);

      mid m1(.clock(c), .d_in(p_in), .d_out(p_out));

endmodule
```


So whatever the port name was there I defined here clock and then say that this net c is connected to clock and d_in is connected to p_in and d_out is connected to p_out. So, we are explicitly making the connection with the port, and in this order, we can define this first or last. It does not matter; the connection will always be connected. So as far as possible we should try to use explicit connection, define connections explicitly while

doing instantiation when we write a Verilog code. Now let us look at what are parameterized module in Verilog. So Verilog allows definition of constants within a module using a keyword parameter.

So, the modules in which a parameter exists are known as parameterized modules. So, while defining a parameter in a module, we give a default value to the parameter, and those default values can be overridden at the time of instantiation by using #. So let us see how it can be done. Suppose there is a module counter.

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

```
module top();
    ...
    counter #(.WIDTH(16)) C1(clk, rst, en, count1);
    counter #(128) C2(clk, rst, en, count2);
    counter C3(clk, rst, en, count3);
    ...
endmodule
```

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

In this module, I am saying that there is a parameter WIDTH, and whose default value is 4, and this parameter WIDTH will be used within this module counter. And this probably will define in some way that what is the size of this counter depending on the parameter WIDTH. Now while instantiating we can say that give the value of WIDTH as 16, rather than 4 or we can say that give the value of 128 to whatever the parameter is in this case. So the parameter here is WIDTH or that is the only parameter in this module so 128 value will be given to WIDTH. Or if we do not specify the parameter value during instantiation then in that case a default value of the parameter that is 4 will be assumed. So what this parameter allows or this flexibility gives us is that it allows us to vary the properties or the characteristics or interfaces of a module based on instantiation.

So the main module or the master module has only one parameter, which is WIDTH, and the module contains all the descriptions with respect to this parameter WIDTH. Now, while instantiating, we can vary the value of WIDTH, and we get different implementations of that module or different variants of that module by varying the parameter. So that we write one module and get different flavors out of it using parameter. Verilog also supports built-in gates. For example, there are keywords AND, NAND, OR, NOR, XOR, and XNOR, which we can use directly in our code and use it

for making logical connections and circuits. So the convention here is that the first pin is the output pin, and the rest are inputs.

```
module mygates(a, b, en, y1, y2);

    input a, b, en;
    output y1, y2;

    and a1(y1, a, b);
    and a2(y2, a, b, en);

endmodule
```

For example, suppose I use a primitive gate or built-in gate of Verilog, and a1 is the instance name, and then we give y1, a, and b. So the first one is the output pin, and the rest are inputs. So this way we can instantiate built-in gates in our Verilog code.

Now, Verilog supports operators and expressions similar to other programming languages such as C. It supports arithmetic operators like +, -, etc, logical operators, bitwise operators, relational operators and shift operators. So these are some of the operators which are supported by Verilog. Now, operators act on operands to produce results. These operands can be of various data types, such as nets, variables, bit select, part select, or array elements.

So operators can act on all these data types, and the number of operands for an operator is defined by the language. For example, there are some with one operand only. For example, a not operation, it involves only one operand. There are some operators which take two operands, for example, less than. Then there are, operators which take three operands.

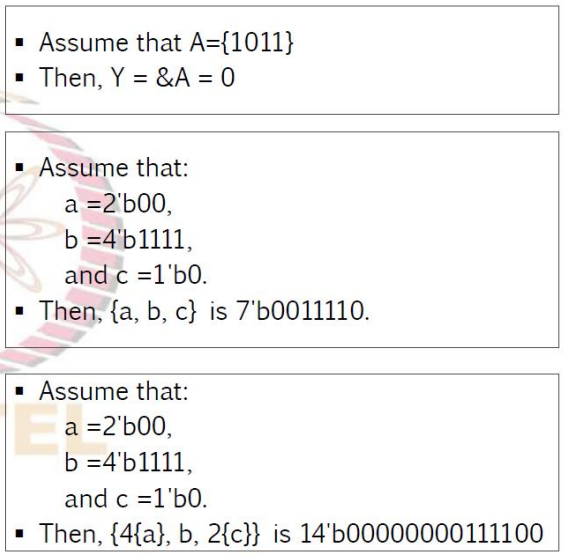
For example, this is a ternary operator. So we say c?a:b. What it means is that if c is true, then it will take a value 'a' else it will take a value of 'b'. So this takes three operands c, a and b.

So the number of operands that the operator can take that is well defined by the language. Now, we can form expressions by combining operators with operands, and yield some results. And we can use parenthesis to override the default precedence that is among the operators. For example, this is an expression. This is an expression !(a&b).

```
!(a&b);
p<q
(x+y)*z
```

And then p<q is another expression, (x+y)*z is another expression. So, we have used parenthesis to override the greater precedence of multiplication over addition. Then there are some special operators which are supported in Verilog. Let us look at a few of them. So, one is called a reduction operator.

It takes a single operand to produce a one bit result. So the reduction operators, and logical operators operate bitwise on the operand, and finally will give one-bit answer. So the computation starts from LSB and moves to MSB bitwise based on what the reduction operator is used. For example, suppose A is an operand whose value is 1 0 1 1. So what it will do is that if we write &A, that is the reduction operator & over A, which will first take the AND of the LSB 1 AND the next bit 1.



So the answer to that will be 1. Then it will take the net bit 0 AND with the present result 1, the answer will be 0. Then it will take the nexct bit , 1 AND with the present result 0, and the answer will be 0. So, the final value of y will be 0. So, this is how the reduction operator works. Then there is a concatenation operator which combines the bits of two or more data objects.

For example, if we have a of 2 bits, b of 4 bits and c of 1 bit. Now if we say {a,b,c}, so it using this concatenation operator, then it will be a 2+4+1 that is 7 bit wide number. And if we want to have many such replication or multiple concatenations, then we can use replication operator. For example, if we have a, b, c, and we want to take 4 copies of a and 2 copies of c.

So we used {4{a}, b, 2{c}}: 4{a} where the number 4 defines that you have to replicate it 4 times, and b and then say 2 times we want to replicate c, 2{c}. So, these are some of the special operators of the algorithm.

Now let us look into the block of statement. So, a group of statements within the keywords begin and end forms a block, and we can also give a name to a block. For example, if we say begin and end, we have these statements in between. So, this is a block of statements. We can optionally give a name to it my_block_name in this case and what will be the understanding or how that language will understand it or the simulator will understand it. So it will basically execute these statements within the block, within the begin end statement in a sequential manner.

```
begin : my_block_name
        statement-1
        statement-2
        statement-3
end
```

So whatever we put inside begin and end that is known as a sequential block. So statement-1 will be executed, then statement-2 and then statement-3. So these are sequential. Then there is another type of block, which is a parallel block in which we use fork and join. The statements here are executed parallely or concurrently. And these kinds of blocks are known as parallel blocks.

So a Verilog supports control statements if-else, case, for, while, repeat similar to C programming language. For example, if you want to write an if statement, we will say that if this condition is true, then do something. So this is how we can write if, a conditional operation. Similarly, if there are multiple options available, then we can use a case statement.

```
if (sel == 1'b0) begin
        y = a;
end
else begin
        y = b;
end
```

```
input [1:0]sel;
case (sel)
        0: y = a;
        1: y =b;
        2: y = c;
        default: y = d;
endcase
```

So case and sel is a signal here. Now depending on the value of sel, whether it is 0, 1, or 2, we assign either a, b, or c, and if none of them matches, then a default value d will be assigned to y. There are two variants of case, one is casez and casex. So casez treats z as don't care. And casex treats x and z both as don't cares.

Then there are loops. For example, for loop, so it is similar to other programming languages. So, in this case, it starts with something and does some part until a termination criteria is met. So this is a criteria which is saying that till when to continue this loop.

```
for (i = 1; i < 8; i = i + 1 )
begin
        state[i] = 1'b1;
        y[i]= c[i];
end
```

```
while (i < 8) begin
        y[i]= c[i];
        i = i + 1;
end
```

```
repeat (8) begin
        y[i]= c[i];
        i = i + 1;
end
```

So it will be continued till I<8. Similar kind of looping we can do using while. So we say while (i<8) do these things. So, the same statement is written in terms of a while, or we can use repeat (8); repeat statement if we know that this has to be done a fixed number of times, 8 in this case, then we can say repeat it and write the code within begin and end. Now let us look into the structured procedures that are available in, in Verilog. So there are four constructs for structured procedures.

So the first one is initial block, second is always block, third is function and fourth is task. So now let us look at all of them one by one. First, let us look into the initial and always block. So what is an initial block? Initial block is a block of code which starts with the keyword initial. And what is an always block? Always block is a block of code starting with the keyword always.

Now, both types of blocks are enabled when the simulation begins, that is, at time 0. But what differentiates them? So, the initial block executes only once and stops when it reaches the end of the block. And always block executes repeatedly throughout the simulation. So this is the main difference between initial block and the always block.

```
module myTop(datain, dataout);
        input datain;
        output dataout;

        reg clock, counter, dataout;

        initial begin
                clock = 1'b0;
                counter = 1'b0;
        end

        always begin
                #10 clock = ~clock;
        end
endmodule
```

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

So let us take an example of it. So here I am showing an initial block, and an always block. So, in this initial block, there are two signals, clock and counter, and at time 0, these two signals will be initialized to 0. And once it reaches end, the simulation will stop for this particular block. But what will happen for the always block? The always block will be executed throughout the simulation.

Suppose initial value of clock was 0. Suppose the initial value of clock was 0. So #10 means that there is a delay of 10 time units in executing this statement. Then after 10 time units the value will become 1, since here clock=~clock, means negation of clock after every 10 time units. So after 10 time units it will again change to 0. Then again after 10 time units it will again become 1. So we will get a infinite sequence of 0, 1, 0, 1, … until the end of simulation.

Now always blocks are useful when we have event control for them. A controlling event can be a value change on a net or a variable. So there are multiple ways to control when an always block is executed. One of them is what is known as using the value change on a net or variable, and this is known as level sensitive always block. So here we say that in this example always @(en) begin rega=regb. What it means is that whenever there is a change in enable (en) signal, this block of code will be executed.

```
always @(en) begin
      rega = regb;
end
```

So here, the event control is provided by the change in the value of the enable signal or en signal. So we can also define an event based on the direction of change of a net or a variable, and that is known as edge sensitive always block. So, there are two types of edge sensitivity. The first one is positive edge. So this is a kind of sensitivity when a signal transitions towards 1.

For example, transition from 0, x or z to 1. So when it transitions towards 1, then we say that it is a positive edge. Remember that Verilog is a four-valued logic. So we have to not only bother about 0 to 1 transition but we also have to consider the transition x to 1 or z to 1 as a positive edge. Also, you should note that the transition from 0 to z/x is also treated as posedge as per Verilog language. Similarly, negedge is a transition of a signal towards 0, and the transition can be from 1, x, or z to 0, or it can be from 1 to z or x.

```
always @(posedge clk) begin
      q=d;
end
```

All these kinds of transitions are treated as negedge. For example, if we write a code like always @ (posedge clk) begin q=d end. So what posedge means here is that

whenever the signal clock will make a transition from 0 to 1, then this block of code will be executed. Other conditions under which this block of code can be executed are based on x and z values as described in this rule. So there are two ways of triggering an always block. The first one is the level sensitive where a signal changes a value, then we say that the always block will be executed.

The other is that there is a direction to the change, meaning that it will get triggered only when the 0 to 1 transition is happening, not the other way around. Sometimes we need to specify the triggering event as an 'or' of various triggering events. So if that is the requirement, then what we can do is that we can separate various events using 'or' statement or comma (,). For example, if we want that this always block gets triggered when there is either a posedge clock or negedge of reset, then we can write like this @(posedge clk or negedge rst).

```
always @(posedge clk or negedge rst) begin
      q=d;
end
```

So, the or separated list of triggering events is called the sensitivity list. So, in this case, there are two signals in the sensitivity list: clk and rst. Sometimes, we need to put all signals read in an always block in the sensitivity list. We want to do that when we are modeling an all combinational logic. So in that case what we can do is that we can use @* to indicate that that block will be triggered whenever there is a change in any of the signals which are read within that always block.

For example, if we say that always @* begin and in between that we write q=((a&b|c)^d)|e, then this block will be triggered when any of the signals a, b, c, d, e which are read, undergoes any change. So, if there are lots of such signals, it may be cumbersome to write all of them in the sensitivity list. So in that case we can simply write always @*. Now, let us understand when to use the initial block and when to use the always block.

So, the initial blocks are used in test benches when we do simulations. We will discuss this later in this course. And it is used to initialize our design for simulation. And always blocks are basic entities which are used for modelling or behavioral modelling of the code. So always blocks are typically used to define our circuit. And initial blocks are mostly used for simulation or test bench purposes.

So a module can have any number of initial block and always block. There can be multiple always blocks, initial blocks and there is no precedence between them. All these blocks get executed at the beginning of the simulation time, and there is no predefined order in starting the execution of these initial and always blocks. And it also means that

initial blocks need not be executed before the always block. So initial and always blocks provide a mechanism to model the concurrency of the hardware.

Now why there is no predefined order? Because in terms of hardware when there is a hardware and there are multiple hardware then we are not sure when we apply an input where the signal will be going first and which one will start operation first. That kind of control is difficult when there are parallel operations. So, in that sense, initial and always block provide a mechanism to model the concurrency of the hardware where anyone can start the operation first. And that is why there is no predefined order defined in the language considering that parallel hardware works in this manner. So, as a designer, what we should do is we should think of the initial block and always block as different hardware components that are working in parallel and whose order of starting the execution is not within our control.

And this this is important because simulator or simulation tool is free to choose any order. So, we should ensure that by choosing any order of execution, we get the correct result. So, the result should not depend on the order of execution of the always blocks in a given Verilog module. Now let us look into functions and tasks.

So, functions and tasks can be used to model repeated codes. So the function starts with the keyword function and ends with endfunction, and the task starts with the keyword task and ends with endtask. So, let us first understand the differences between functions and tasks. So a function can have one or multiple inputs at least there should be one. And it should produce only one output while task can have zero or multiple inputs and it can also have zero or multiple outputs or inouts.

Now functions cannot have delays modeled as posedge, negedge or # delay. While task can have delays modeled by posedge, negedge or #. And functions are used to model only the combinational circuit, while tasks can be used to model both combinational and sequential circuits. And functions can call another function but cannot call another task. While task can call other task as well as function.

So, these are the most important differences between functions and tasks. Now, let us look into the uses of these structures. So this is an example where function is used. So, there is a function within a module.

The name of the function is myfunc. And the inputs are x, y, z. So there are three inputs. And the output is x-y+z and it is assigned to myfunc. So there is a signal or register which is having a name as myfunc which can be used inside this function. We can call these functions, so there are two calls of functions. In first myfunc(a,b,c) we are giving the inputs as a, b, and c.

```
module top(a, b, c, d, out1, out2);
    input a, b, c, d;
    output out1, out2;

    function myfunc;
        input x, y, z;
        begin
            myfunc = x-y+z;
        end
    endfunction;

    assign out1 = myfunc(a, b, c);
    assign out2 = myfunc(b, c, d);

endmodule
```

```
module top(a, b, c, s1, c1, s2, c2);
    input a, b, c;
    output s1, c1, s2, c2;

    task mytask;
        input x, y;
        output sum, carry;
        begin
            sum = x ŷ;
            carry = x & y;
        end
    endtask;

    mytask(a, b, s1, c1);
    mytask(b, c, s2, c2);

endmodule
```

So these will be mapped to x, y, z in this function. And the answer will be corresponding to this expression a-b+c and that value will be assigned to out1. So this is what the equivalent code will be of this function. And let us look into an example of a task.

So here the task name is mytask. And the inputs are x and y, and there are two outputs. So in this case, we had only one output in case of function. Here there are two outputs, sum and carry. And these are the calls of the task mytask(a, b, s1, c1), mytask(b, c, s2, c2). So, depending on what the requirement or repeated code or repeated structure we have in our hardware, we can use either function or task in our Verilog code.

Now, let us look at various types of assignments in Verilog. There are two types of assignments in Verilog: continuous assignment and procedural assignment. Now, what is a continuous assignment? Continuous assignments provide values to nets, and the keyword for continuous assignment is assign, and it is used to model combinational circuits. So here is an example, assign out=(s)?a:b, if s=1 then a, else b. So it is a kind of a multiplexer that will be created for out. Now this is called a continuous assignment and it is assigning a value or giving a value to a net whose name is out.

```
module mymux(a, b, s, out);
    input a, b, s;
    output out;

    assign out = (s) ? a : b;

endmodule
```

```
always @(posedge clk)
begin
    q = d;
end
```

```
always @(posedge clk)
begin
    q <= d;
end
```

There is another type of assignment, which is a procedural assignment. Now, what are procedural assignments? Procedural assignments provide values to variables such as reg or integer types. So the values that are given to these or that are assigned in procedural assignment they remain there. And they do not change their values until the next procedural assignment. So values get assigned, and it remains there until it is overridden by another value.

So it is used inside structured procedures, initial block, always block, functions and tasks. So whenever we want to write values or assign values inside the initial block, always block or functions and tasks there, we need to use reg type or integer type and use procedural assignments. So there are two types of procedural assignments. One is known as a blocking assignment, and another is a non-blocking assignment, as shown here. So if we write q=d, that is a blocking assignment.

What does that mean? We will just see. And if we write q<=d, the symbol is same as less than or equal to. Depending on the context, it will be understood whether it is less than or equal to or it is a procedural assignment. So q<=d, this means that the value of d is assigned to q in a non-blocking manner. What do we mean by a non-blocking manner? We will just see.

So, there are two types of procedural assignment. First one is blocking, another is non-blocking. The symbol for blocking is = and for non-blocking <=, we have just seen. An easy method to remember the symbol for the assignment we can think of that for the blocking assignment, blocking is a smaller word and has a smaller symbol, and non-blocking is a bigger word and has a bigger symbol. So, for a blocking assignment, what happens is that the execution of the next statement is blocked until the current statement is executed. So, it is similar to what we do in our programming languages.

So one statement is executed, then the next one, and so on. So it is done in a sequential manner. And what happens for non-blocking assignment? Execution of the next statement is not blocked until the current statement is executed. The statement is not blocking the other statements while the first statement is being processed, the next one can continue processing and so on. That is what is done for non-blocking assignment, and it means that non-blocking assignment somewhat models a kind of parallel execution.

Now let us understand a little more that what non-blocking assignment is. When we say something, say A, assign it to B in a non-blocking manner, so this assignment is done in two steps. In first step what is done is that only the RHS part is evaluated. Suppose it was a complicated signal ((B&C)|Y), some complicated expression was there on the RHS.

So, whatever the current value of RHS is that is evaluated in the first step. And the execution moves to the next statement. The LHS is not yet assigned the value. That part happens at the end of the simulation.

We will be understanding this more precisely in the next lecture. But the important thing is that whenever there is a non-blocking assignment, the execution happens in two steps. In the first step, only the RHS part of the expression is evaluated, and its current value is kept somewhere with the intention that at the end of the simulation time, I will assign the current value of RHS to the LHS. So, in the next step, the LHS will be assigned, but the execution is allowed to continue to the next statement and so on.

So this is how the execution of non-blocking assignment is done. So, let us understand this using an example. So in this module there are two initial blocks, the first initial block and the second initial block. So, both these initial blocks will start executing at time 0. Now, in the first initial block, all the statements are blocking assignments. In the second initial block, all the statements are non-blocking assignments. Now when the first initial block is being executed then the simulator will encounter the statement a = #10, 1'b1, #10 means that there should be a delay of 10 time units.

So it will start executing at time 0 and the statement is saying that after 10 time units a should be assigned a value of 1. So at t=10 time units a will be assigned a value of 1. Then it goes to the next statement. Next statement says that after 30 time units assign value of 1 to b.

```
module top( );
    reg a, b, c, p, q, r;

    initial begin
        a = #10 1'b1; //at time = 10
        b = #30 1'b1; //at time = 40
        c = #20 1'b1; //at time = 60
    end

    initial begin
        p <= #10 1'b1; //at time = 10
        q <= #30 1'b1; //at time = 30
        r <= #20 1'b1; //at time = 20
    end

endmodule
```

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

So currently, t was 10, so 30 will be added, and t becomes now 40. So at t=40, b will be assigned a value of 1. Then comes the third statement which says c=#20, 1'b1. So after a time of 20 time units that is 40 plus 20 that is t=60 time units c will be assigned a value of 1. So this assignment has happened in a blocking manner, meaning that the execution does not reach the next statement until the first one has done all the processing. Similarly, it does not reach the third statement until the second one has also done the process.

So this is a kind of blocking assignment similar to traditional programming language. But something interesting happens in non-blocking assignment. Now in non-blocking assignment as I said the execution happens in two steps. So when it encounters p<=#10 1'b1 it simply says that I want to assign RHS, i.e., 1 to p and at what time? At time t=1010. So it just schedules or plans schedule to write p or write 1 to p at t is equal to 10 and after this scheduling it moves to the next statement.

It does not wait till 10 time units and till the LHS is updated. When it goes to the next statement, it says that the RHS is 1'b1, meaning that it is 1. So it will say that I schedule to write 1 to q at t=30, and it moves to the next statement. Similarly, it schedules to assign 1 to r at time t=20, and it moves. Now it reaches the end. Now at the end it will update all the values, at time 10, p will be getting a value of 1, at 30, the value 1 will be written to q and at 20, value 1 will be written to r.

So as per what was scheduled to be written, the LHS will be updated all of them together. So in that manner, in that sense, this p, q, r has been written in a parallel manner, and that is what the difference is between the simulation of a blocking assignment and a non-blocking assignment. Non-blocking assignment, as we see, is somewhat modeling a parallel execution or parallel assignment of the registers or

variables. Now, in addition to the constructs that we saw, the Verilog supports various system tasks and functions. So these system tasks and functions help us in debugging and in verification.

So the name of these functions always start with a $. A few examples of system task and functions are $display, $probe, $monitor, $stop, $finish, $reset, $random, $time, etc. So these are some of the useful system tasks and functions, and their names are most of the time self-explanatory, but you should be going through the manuals to understand more deeply how these system tasks and functions behave or work. So these are some of the important references that I have used in this lecture, and you may go through them. And there are lots of materials available over the internet on Verilog, and those are also quite useful. But the Bible, of all those things, is the language reference manual. So whenever in doubt, please refer to the language reference manual, and that is the most authoritative kind of description of the language.

So, to summarize what we can say is that in the last two lectures, we have looked into various constructs of Verilog, which will help you in writing a Verilog model or RTL model. Now, in the next couple of lectures, we will be looking into how these constructs are used or interpreted by simulation and synthesis because we need simulation to verify that what we have written is as per our expectation. And we need synthesis to get hardware out of it because in RTL, when we write in terms of using the constructs that we have described, those will be a kind of behavioral model of our design in terms of how the signal goes from one registers to other and so on. The corresponding hardware or the logic or the netlist out of it has to be interpreted by looking into the constructs, its meaning and then mapping it to a hardware and that is what the RTL synthesis is. So, we will be looking into RTL simulation and RTL synthesis in the next couple of lectures. Thank you very much.