


System Design using Verilog
Dr. Shaik Rafi Ahamed
Department of Electronics and Electrical Engineering
Indian Institute of Technology, Guwahati

Introduction to Verilog
Lecture - 02
Verilog Ports, Data types and Assignments

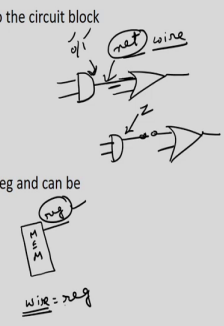
In the last class, we are discussing about the basics of Verilog. So, we have discussed about this number representation, logical operators, mathematic operators, conditional operators, etc. ok.

(Refer Slide Time: 00:52)

Data Type



- **Net Types** ✓
 - Physical Connection between structural elements.
 - Carries the value of the signal it is connected to and transmit to the circuit block connected to it.
 - If driving end of a net is left floating, the net goes to z.
- Ex: **wire** ✓
- **Register Type:** ✓
 - Represents an abstract storage element. ✓
 - A net or wire connected to reg takes on the value stored in the reg and can be used as input to other circuit elements
- Ex: **reg** ✓
- Default Values
 - Net Types : z
 - Register Type : x



So, next one is the data types. So, basically, we have two data types; one is called a net, another is register. So, in the net data type as I have discussed in the earlier lecture also so, this net represents the physical connection between the structural elements. Suppose if you want to connect the output of one gate to the input of other gate so, this is we will call this as a net.

So, net basically represent the physical connection between the structural elements, and it carries the value of the signal, the output of this AND gate, if this is 0 or 1 so, this net will carry this information 0 or 1 and it will transmit to the input of OR gate. Suppose if the

driving end of the net is floating for example, if I have something like this, I have to connect to this one, but this is floating.


Then this net value we will assume a high impedance state z . So, the example of this one is wire. So, basically, we can define this net as a wire. The second type is a register type. As the name implies register, which can be used to store the information ok, a register is capable of storing the information. So, this register represents an abstract storage element.

So, this is again it can be net or wire connected to the register takes the values stored in the register and can be used as an input for the other circuit element ok. Suppose, if I store this register, again we are going to define later register can be a scalar or vector ok. Suppose if I have some block say memory block and if I define this as a register and if I define this wire is equal to register, whatever the information on the register will be taken care by this wire.

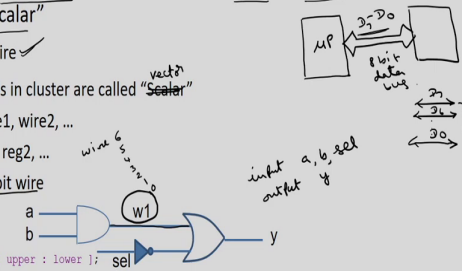
If I connect this wire to some of the other circuit element, then the whatever the information on register will be carried to this particular circuit element ok. So, the example of this register type is reg, reg is the keyword used for the register and wire, wire is the keyword used for the net. So, the default values for the net it is high impedance state, register it is don't care.

(Refer Slide Time: 03:48)

Scalars and Vectors



- Entities representing single bits (whether the bit is stored, changed or transferred) called "Scalar"
 Ex: `wire w1;` // A 1-bit wire ✓
- Multiple lines carry signals in cluster are called "Vector"
 Ex: `wire [msb : lsb] wire1, wire2, ...`
`reg [msb : lsb] reg1, reg2, ...`
 ✓ `wire [6:0] Clear;` // A 7-bit wire



```

      Ex: reg [msb : lsb] memory [upper : lower];
reg [0 : 3] mem (0 : 63);
      // An array of 64 4-bit registers
reg mem [0 : 4];
      // An array of 5 1-bit registers
      
```

Next, we can have the scalars as well as vectors ok, the data type can be either scalar or vector ok. As the name implies scalar so, all the entities representing a single bit is called as a

scalar whether that bit can be stored or changed or transferred. So, whatever the operation is a single bit is normally called as a scalar ok example is here wire w1.

If I take this particular circuit ok here, we have three inputs a, b and selection are the inputs and output is output y whereas this w1 is not accessible to the programmer ok. So, all the I mean connections which are not accessible neither input nor output, we have to define by the wire.

So, here only single output, the output of this OR gate is single bit so, we need a scalar. So, this wire w1 if I define so, this will act as a 1-bit wire. And the other hand vector as a name implies it is a multiple line that carries signals in cluster. So, basically like if want to I mean transfer the data from the microprocessor to memory, this is a microprocessor and here we have some memory suppose if it is having 8-bit data bus, this is D 7 to D 0, 8-bit data bus.

So, this is nothing but actually this is bi-directional. So, instead of writing this D 7 line is this, D 6 is this so on up to D 0 instead of writing 8 lines separately, we will write using bus ok. So, such type of the clustering of the single lines is called as a vector ok. So, the example is we can have the wire with multiple lines msb and lsb ok.

Similarly, register can have some msb, lsb instead of representing all, if you want to represent this data, you can represent with 7 and 0. So, this is example if I take wire 6 comma 0, this is a 7-bit wire. So, what are these 7 wires? One is wire 6, wire 5, wire 4, wire 3, wire 2, wire 1, wire 0. Instead of writing these many wires, we can represent with a vector wire 6 is to 0, then some Clear so, Clear is the name of that particular wire say.

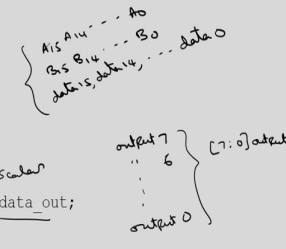
Similarly, you can represent for the memory also if you have register 0 to 3 or memory 0 to 64 ok so, this represents an array of 64 4-bit registers ok, this is 4-bit register, this represent this 4-bit register and this is 64 arrays, an array of 64 4-bit registers can be represented in this way register followed by memory.

And if I write just only simple register m means this is 5 1-bit registers if nothing is written here only register is there means 1-bit, if 0 to 3 is mentioned this is 4-bit registers ok. So, this is how we can mention, or you can represent the scalars and vectors in Verilog.

(Refer Slide Time: 07:50)

Ports

- Ports provide the interface by which a module can communicate with its environment. For example, the input/output pins of an IC chip are its ports.
- Port Declaration:-**
 - `input` Input port
 - `output` Output port
 - `inout` Bidirectional port
- Input Declaration
 - Scalar
 - `input` list of input identifiers;
 - Example: `input A, B, c_in;`
 - Vector
 - `input` [range] list of input identifiers;
 - Example: `input [15:0] A, B, data;`
- Output Declaration
 - Scalar Example: `output c_out, 0V, MINUS;` scalar
 - Vector Example: `output [7:0] ACC, REG_IN, data_out;`



Next coming for the ports. So, we have three types of the ports. So, it can be either input port, output port or bidirectional port `inout` ok, these are the keywords used for this input port, this is keyword for output port, this is keyword for bidirectional port ok. So, we can have either the ports can be either scalar or vector ok.

For example, if I write in input A, B, C in so, these are all scalars. So, this A, B, C in are the inputs ok. We can represent single keyword, we can represent as many numbers of signals as possible ok. So, we have three signals, input signals where we can represent with input.

Suppose if we want to represent a vector, we have to represent a range. So, this A is 16-bit width which is, vector B is also another vector, data is another vector. So, this is nothing but A 15 to A 0 so, this 15 is to 0, A represents A 15 to A 0. Similarly, B 15 to B 0 and data also data 15 to data 0, but all these three are defined as input ports ok.

So, there are some examples of the output ports also. Output port is defined as C_{out} it is a single bit, this is a scalar ok and 0 volts which is ground, MINUS is a minus symbol. A vector is again if you have 8 outputs, you can represent output 7, output 6 so on up to output 0, we can represent as simply 7 : 0 output.

Similarly, this output we are calling as accumulator, accumulator giving 8-bit. Similar register in is 8-bit, data out is 8-bit ok. Similarly, you can have `inout` also you can use here instead of

output if a particular port is bidirectional port ok. So, basically, we have three types of the ports: input port, output port and then, bidirectional port ok.

(Refer Slide Time: 10:25)

Event Control

- Event Control
 - Edge Triggered Event Control ✓
 - Level Triggered Event Control ✓
- Edge Triggered Event Control
 - @ (posedge clk) // Positive Edge of CLK
 - Curr_State = Next_state;
- Level Triggered Event Control
 - @ (A or B) // change in values of A or B
 - Out = A & B;

Then, there are some event control elements ok. So, we know that in sequential circuits ok so, we will apply the clock signal so, the sequential circuits can be either edge triggered, or level triggered. If I take for example, a clock signal; ideal clock signal, this is ideal clock why ideal? To change from 0 to 1, this is logic 0 logic 1, this is taking 0 amount of the time, to change from 1 to 0 also, the time is 0 ok.

But practically, it will take some time. If I take the practical clock, there will be a rise time and fall time type of thing. So, 0 to 1 it will take some time, this is the transistor time required for changing 0 to 1. Similarly, 1 to 0 also it will take some time. From here to here, we will be calling one period.

Over a period, a clock whether it can be ideal or practical normally, we will use practical clock is having four portions; this is one portion, this is another portion, this is third portion, this is fourth portion. So, the transition from 0 to 1 is called as positive edge or leading edge and the duration over which the clock is equal to 1 is called as positive level.

And the transition from logic 1 to logic 0 is called as negative edge or falling edge and if clock is equal to 0, it remains at 0, then this is called as negative level. So, we can define the

flip-flops or we can design the flipflops which can be either edge triggered, or level triggered ok, but normally, this level triggered flip-flops will be having problem with race around condition.

So, in most of the applications especially in counters and all, we will use edge triggered. See in order to represent this edge triggered and level triggered so, we have to represent like at whether a positive edge of the clock clk, this is positive edge is this, this positive edge in Verilog will be represented by posedge means whenever the positive edge occurs, then only the particular operation will take place ok. So, Current State is equal to Next state.

If I take the flipflop say D flip-flop with a clock this is input D, this is output Q, this is clock, this is positive edge triggered. Negative edge triggered normally we will follow the notation like a bubble here. If I take this clock, simply if we write clk, D, output Q, this is negative edge. So, distinguish between positive edge and negative edge, we will place a bubble and without bubble ok.

So, the meaning of this one is in case of D flip-flop, the output is equal to input, but when does this output becomes input? Q is equal to D only, but at a very positive edge ok that is what we are going to explain here. So, at a very positive edge, Current State is equal to Next state ok. So, whatever the current state, it will be equal to next state means simply the output will be transferred to the input.

And if I take another example where this is a level triggered ok, at A or B, output is equal to A AND B, this is AND operation between A and B. So, in the previous lecture, we have discussed this AND operation as A, B say output Y is equal to this, this is also AND operation.

We just I mean understand the difference between these double AND and single AND. So, this particular AND operation is called this is binary operation whereas, this particular AND operation is called unary operation. So, this will take this A and B as a whole, this can have any number of the bits and it will perform the AND operation whereas, this operation is bitwise ok.

If of course, they are single bit, both will be same. If A and B are having multiple number of the bits, then unary operation and binary operations are different so, binary you cannot present for more than 1-bits ok, you have to represent you have to AND with each and every bit whereas, here, as a whole you can perform this operation ok.

So, normally, this type of operation will be used in the data flow modelling that we are going to discuss in the coming slides ok. So, this is Out is equal to A AND B means so, this output of this AND gate will become A AND B, this will be executed see whenever a positive edge occurs when A is equal to 1 or B is equal to 1.

If I apply here so, at this point when A is equal to 1 or when B is equal to 1, then only this particular operation will be executed here, here and all ok otherwise, this output will remain in the previous state ok. So, this is about the event control. So, we can control the events based on whether this is a positive level or positive edge, negative level or negative edge ok.

(Refer Slide Time: 18:04)

Assignments

Continuous assignments ✓ assign

- assign values to nets (vector and scalar) ✓
- They are triggered whenever simulation causes the value of the right-hand side to change
- Keyword **assign** ✓ unary AND
- e.g. **assign** out = in1 & in2; ✓

Procedural assignments ✓

- Drive values onto registers (vector and scalar)
- They occur within procedures such as **always** and **initial** begin
- They are triggered when the flow of execution reaches them (like in C) end
- Blocking and Non-Blocking procedural assignments
- All statements inside an **initial** statement constitute an **initial block**
- An initial block starts at time 0, executes exactly once during a simulation, and then does not execute again.
- If there are multiple initial blocks, each block starts to execute concurrently at time 0. Each block finishes execution independently of other blocks.

out
in1
in2
AND
assign

So, the next I mean operation is assignments. So, we can assign some variable to the some other variable ok. So, there are basically two types of these assignments one is called continuous assignment, another is procedural assignment ok. Again, in the procedural, we have two types again this is blocking and non-blocking. Again, this will be used with data flow modelling ok.

So, in continuous assignment so, as the name implies, we will assign values to the nets, net can be either vector or scalar, this we have already discussed in the earlier slides ok. Here, basically we are going to assign some values to the nets, it can be vector or scalar. So, when does this particular value will be assigned is so, whenever the right-hand side value will change.

So, for example, if I take this assign, assign is the keyword used for the either continuous assignment or procedural assignment, whether it can be blocking or non-blocking so, for all the assignments, the keyword is assigned. Here in this example, I have given assign out is equal to in1 AND with in2, this is actually as I have told this is behavioural, this is unary AND operation.

So, they will be triggered whenever in1 or in2 will change, this is AND gate output is out, in1, in2 ok. So, whenever the value of the right-hand side, right-hand side is this, in1 and in2. So, whenever in1 changes or in2 changes, then only this particular instruction will be executed ok otherwise not. This is called continuous assignment.

Whereas the second type of assignment is procedural assignment. So, this drive are the registers, this is normally used with the nets. As I have told there are two types of the data types, one is net, and another is register. So, to deal with nets, we will use the continuous assignments and to deal with the registers, we will use procedural assignments ok.

So, normally, this procedural assignment whether this is blocking or non-blocking always uses with always and initial, there are two more keywords so, these keywords will be used in this procedural assignment and also in addition to there are two more which is called as initial begin and end.

So, these are other two keywords we are going to use ok. So, always this procedural assignment in addition to this assign, we have to use always or initial and begin and end ok. So, this will be clear if I give some examples in the next slide. Now, they are triggered when the flow of execution reaches them, this is sequential.

So, for here, we have some assign statement so, this will be executed so, first this will be executed, this will be executed, this will be after that this will be executed in sequence like C


whereas, here, this will be triggered whenever the one of the inputs that is right-hand side will change ok that is the main difference between the continuous assignment and procedural.

Procedural is in a sequence that will be executed whereas, in continuous, whenever the right-hand side expression changes, then only that particular assign will be executed. So, all these statements inside this initial statement constitutes initial block. Similarly, all the statements in always constitutes always block.

So, initial block starts from the time $t = 0$ and execute exactly once during the simulation. If there are multiple initial blocks, then all will be executed simultaneously at t is equal to 0 ok and the execution completion that depends upon the instruction that are present in the initial block ok. This will be clear if I give the examples now.

(Refer Slide Time: 22:55)

Procedural Assignments (cond..)



- All behavioral statements inside an **always** statement constitute an always block.
- The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion if the sensitivity list is satisfied.
- This statement is used to model a block of activity that is repeated continuously in a digital circuit.
- Example

$t = 0$

```
...
initial(begin)
  Sum = 0;
  Carry = 0;
end
...
```

} Initial block

```
...
always @(A or B)
begin
  Sum = A ^ B;
  Carry = A & B;
end
...
```

} always block

Unary XOR

Unary AND

So, again two examples, one with initial, another with always ok. So, this is initial or always, always you have to use begin and end. Say whatever these statements that are present between this begin and end are called as this is called initial block and this is called always block.

So, when does this initial block will be executed? When does this always block is executed? As I have told this will start at time t is equal to 0 and this will execute only exactly once, this

is some sort of initialization as the name implies. So, initial Sum I have defined for 0, initial Carry also I have defined to 0 whereas, in always so, there is a condition A or B.

So, Sum is equal to this is exclusive OR operation in procedure or this data flow modelling, this is XOR operation, unary exclusive OR operation and this is unary AND operation. So, we know that for half adder Sum is equal to A exclusive OR B and Carry is equal to A AND with B ok.

So, when does these two will be executed? Whenever A changes or B changes because if A or B changes correspondingly, sum and carry also will changes ok that is always at A or B, these two will be executed whereas, here this will execute only once at time t is equal to 0 ok. So, that is the main difference between this initial and always blocks ok.

(Refer Slide Time: 24:54)

The slide is titled "Assignments (cont.)" and features a logo in the top right corner. It contains the following text and diagram:

- Procedural Assignments**
 - Blocking assignment**
 - Blocking statements are executed in the order they are listed. ✓
 - The assignment operator (`=`) is used in these statements (acts much like in traditional programming languages).
- Ex: `reg a, b, c, reg_1, reg_2;`
- Initial ✓
- begin ✓
 - `a=1;` ✓
 - `b=0;` ✓
 - `c=1;` ✓
 - `#10 reg_1=1'b0;` ✓
 - `#5 c=reg_1;` ✓
 - `#5 reg_2=b;` ✓
- end ✓

The diagram shows a timeline for time t (Sec) from 0 to 20. At $t=0$, `a=1`, `b=0`, and `c=1` are assigned. At $t=10$, `reg_1=0` is assigned. At $t=15$, `c=0` is assigned. At $t=20$, `reg_2=0` is assigned. A vertical line at $t=0$ is labeled "# delay". A circled "`=`" is labeled "blocking". A circled "`<=`" is labeled "non-blocking".

So, as I have told in procedural, you have two types of assignments again so, one is blocking assignment, another is non-blocking assignment ok. As the name implies blocking this will block, non-blocking means it will not block the other operations ok. So, in blocking assignment, they are executed in the order they are listed ok.

And the main difference is we are going to use is equal to operator in blocking assignment whereas, in case of non-blocking assignment, we are going to discuss in the next slide, we are

going to use less than equal to ok, this is the one of the main differences between blocking and non-blocking assignments.


So, I have given the example of the blocking assignment. So, I have defined the registers as a, b, c, register 1, register 2 and I have given the initial, begin and end ok. So, I have given this a register I have defined with 1, b with 0, c with 1, this is this will happen at t is equal to 0 ok. So, there are I mean different ways to represent these delays. So, this symbol hash symbol represents delay ok.

If these delays are not there, all these statements will be executed at t is equal to 0 because these delays are present here ok. So, if I take this timescale at t is equal to 0 this is time, at t is equal to 0 at this point so, a will become 1, b will become 0, c will become 1 and this represent that after 10 seconds, if I give this in seconds, after 10 seconds somewhere here at 10 seconds.

The register 1 is assigned with here this register 1 will be assigned with this is the I mean number representation that I have discussed in the earlier lecture so, 1 stands for 1-bit, b stands for binary, the bit value is 0 so, this is equal to 0. Then, the second statement represents there is some other delay 5 so, after 5 more delays means total becomes 15 so, this will add together ok.

So, at this point, c will be assigned to register 1 so, c was initially at t is equal to 0, 1 now, c will become register 1 value which is 0. After 5 more seconds at 20 seconds, register 2 is assigned to b value, b value was 0. This is how this all these statements will be executed in sequence ok. If these delays are not there, all will be executed at t is equal to 0 only ok. So, this is the example of blocking assignment and here, we are going to assign with is equal to symbol ok.

(Refer Slide Time: 28:28)

Assignments (cont.) 

- **Nonblocking assignment**

- Contains \leq assignment operator ✓
- A non-blocking statement does not block the execution of other statements in the list.
- Evaluates all the right-hand sides for the current time unit and assigns to the left-hand side at the end of the time unit

Ex: always @ (posedge clk)

```
✓begin
  B <= A; ✓
  C <= B; ✓
  D <= C; ✓
✓end
```

executed simultaneously

$\langle =$ not less than or equal

~~B = A~~
~~C = B~~ old B
D = C old C

So, on the other hand, the other type of procedural assignment is non-blocking assignment. As I have told so, here we will use less than or equal to, but do not confuse with this one, this is not equal, less than or equal to ok, this is only assignment this is not less than or equal to, this is only the notation, but exactly not this is a less than or equal to ok.

So, this will be assigned by this operator and a non-blocking statements does not block the execution of the other statements in the list. So, unlike this previous one, here unless otherwise I will execute this one, I cannot execute this so, this particular statement is blocking this statement and also this statement.

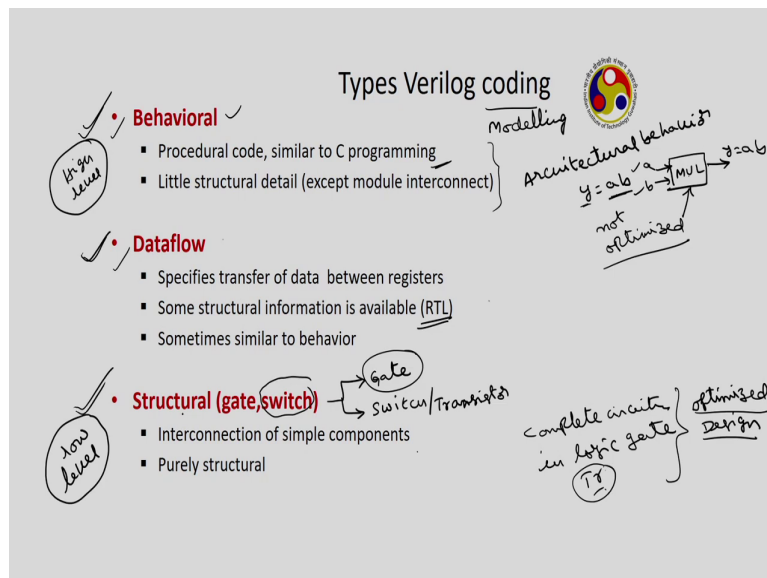
So, this particular statement is blocking this statement ok that is why the name blocking. So, this particular statement is blocking these two statements whereas, in case of non-blocking as the name implies so, the execution of the current statement does not block the execution of the other statements ok.

So, this will be a clear with this example. So, always at positive edge of the clock, begin and end is required whether it is a I mean blocking or non-blocking assignments so, what you have defined is B is equal to A, C is equal to B, D is equal to C so, these three statements will be executed simultaneously.

So, this statement will not block this, this, this statement will not block this ok, like in case of the previous blocking assignment ok. So, at every positive clock edge so, B is defined with A so, B will become A and C will becomes B, but the new value of B is A initially, some value will be there ok.

So, whatever this C will be assigned is this is old B because this statement and these statements will be executed simultaneously unless otherwise this is executed, we cannot assign this value here so, the value that is assigned to C will be the old B. Similarly, D is assigned to C means this is not this particular new C, this is old C. So, see how we can execute this non-blocking statement without blocking the other statements ok.

(Refer Slide Time: 31:23)



Then coming for this different type of Verilog modelling or coding or modelling. Basically, we have broadly we can classify into three modelling so, one is called behavioural, another is called data flow, another is structural. Of course, in structural, I will define this as a two, one is gate level as well as switch level or transistor level.

This is low level of abstraction so, we will have all details of the circuitry, and this is high level, and this is medium level, moderate one ok. So, this behavioural is exactly same as the C type of the programming. So, it becomes easier to write the behavioural code, but it will not be optimized ok.

So, we are going to represent with some blocks only ok, the system some sort of the architectural type of thing ok, this is architectural behaviour we are going to multiplication say y is equal to a into b , this is multiplication, we will not mention what is the operation inside this, this is only simple multiplication. What is the architecture inside this multiplier? We will not mention, this is MUL will take a block with a and b so, output y is equal to a into b .

So, what is this circuitry that performs the multiplication of a and b will not be considered at behaviour level ok because of that so, this will design its own multiplier and it will perform this y is equal to ab so, in that way this is not optimized whereas, in data flow, normally we will use a RTL type of modelling.

So, we have some details of this, but not gate level ok whereas, here, we will be having the complete circuitry of the multiplier either in terms of logic gates if it is gate level, if it is switch level even we have transistor ok so, this will be the more detailed one so, we can easily optimize this because we know the entire internal circuit of this one, we can provide some pipelining, parallel processing, some sort of the optimization techniques and we will get mostly optimized design here.

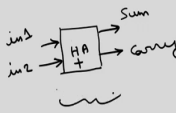
So, if want more optimized design, we have to go for the structural ok, but the problem is for the complex systems so, structural design becomes somewhat difficult ok, it will be consisting of thousands of millions of the gates so, it becomes difficult to I mean write the structural modelling. So, for normally the complex systems will go for the behavioural. They are simple systems; it is better to always use structural and this data flow is in between ok.

So, I will explain this with an example so that we will get the clear difference between these three types. So, of course, I am not going to discuss now here the switch level ok, I will take one example. So, I will discuss how this behavioural code will be appears data flow and structural ok.

(Refer Slide Time: 35:14)

Behavioral Design (Half Adder)

- `module adder_b(in1, in2, sum, carry);`
- `input in1, in2;`
- `output sum, carry;`
- `assign {carry, sum} = in1 + in2;`
- `endmodule`



Addition is a high level construct Verilog compiler will generate the code for addition

We have taken a simple example of Half Adder. So, behavioural so, half adder will be taken as a simple block, I do not know what is the circuit inside this, I will take as a block which is plus block with in1, in2 inputs, output is sum, carry. So, you have to define this, this is module name you have to write, this is module is the keyword and this adder underscore b, b stands for behavioural is the name of this module.

This is optional one, this is name of the module and we have to mention all the inputs and outputs here and we have to clearly define using input, output ports as we have discussed in the earlier slides, in1 or in2 are the input ports, carry and sum are output.

Then, here we will write a single statement like assign carry, sum is equal to in1 plus in2 ok. So, this addition is high level construct, the Verilog compiler will generate the code for the addition ok simply it will add and whatever the sum is assigned, sum is a resulted that will be assigned to sum, carry resulted will be assigned to carry. I do not know what is the circuitry inside this, just this is abstract level modelling.

(Refer Slide Time: 36:48)

Data Flow Design (Half Adder)

- module adder_@ (in1, in2, sum, carry);
- input in1, in2;
- output sum, carry;
- assign sum = in1 \oplus in2;
- assign carry = in1 \otimes in2;
- endmodule

Unary XOR
Unary AND

Boolean logic - more fundamental construct than addition

On the other hand, if I use this data flow design, these things are same, I have given just name as d for data flow, this is optional one so, the remaining all are these are same except for the name I have changed. Then here, we will use assign statement as I have told assign statement ok. In this we are going to use blocking statement so that we can define this sum and then, carry ok.

So, assign sum is equal to this I have told, this is unary exclusive OR operation. So, sum is nothing but exclusive OR between both the inputs, carry nothing but this is unary AND operation, then end module. So, this is somewhat I mean better than the previous model. Of course, this is also not giving the complete details ok, but we are going to perform this exclusive OR and AND operations.

(Refer Slide Time: 37:53)

✓ Structural Design (Half Adder)

- module adder_s(in1, in2, sum, carry);
- input in1, in2; ✓
- output sum, carry; ✓
- and g0(carry,in1,in2);
- xor g1(sum,in1,in2);
- endmodule

Assuming and and xor gates are available

Gate level
Switch level
Behavioral

Structural

And the low-level modelling is a structural design ok. Here also this I have given as a name as for structural otherwise, these three statements are same, here we need the circuitry so, what is the circuitry of this one ok? Carry should be AND operation, sum should be exclusive OR operation. So, we are defining this as gate 0, gate 1 ok.

So, we have to write directly AND gate, OR gate, we are going to use primitives, and is the primitive used for the AND gate or XOR is the primitive used for the XOR gate ok. So, AND g0 is the gate name, this is optional, you can give any name here.

Then, you have to mention the output followed by the inputs so, one output carry, two inputs because we already define this input in1 and in2 as input through the input port declaration, sum and carry we have defined as output port using this declaration. So, the similarly exclusive OR the sum is output, in1, in2 are the inputs this is endmodule.

So, here we should know this gate level circuitry of whatever the circuit that we are going to model ok. Unless otherwise you do not know this gate level circuit, we cannot write the code for structural design ok that is why this is more optimized ok. So, there are about three different things and in structural also we have instead of gate level, you can have switch level also.

Here, I have given just a simple example of this half adder to understand the difference between these three types of the modelling. So, in the coming lectures, we will discuss in detail about each type of the modelling. So, of course, first I will start with the structural modelling which is gate level, then I will discuss about the switch level, then I will discuss about the data flow, these two comes under structural, then I will discuss about the higher level, which is called behavioural in detail with some examples ok.

Thank you.