Digital VLSI System Design

Prof. Dr. S. Ramachandran

Department of Electrical Engineering

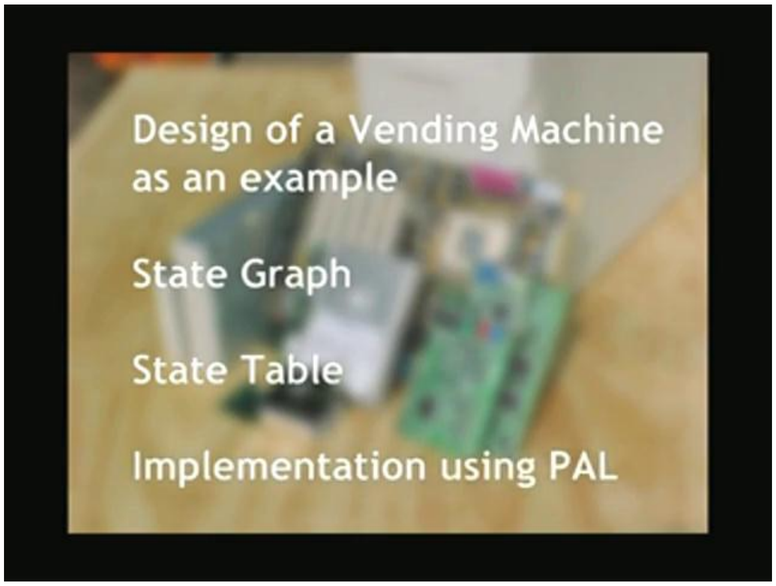
Indian Institute of Technology, Madras

Lecture - 10

Verilog Modeling of Combinational Circuits

Slide – Summary of contents covered in previous lecture.

(Refer Slide Time: 01:11)



Design of a Vending Machine as an example

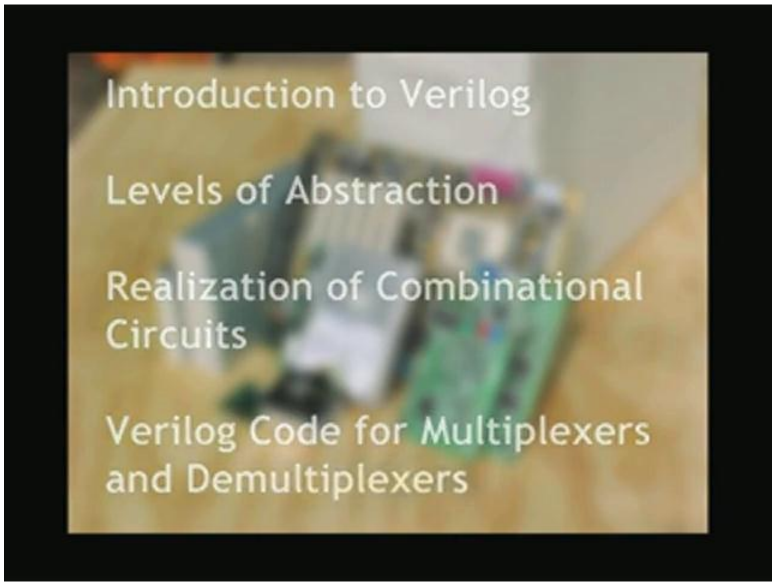
State Graph

State Table

Implementation using PAL

Slide – Summary of contents covered in this lecture.

(Refer Slide Time: 01:35)

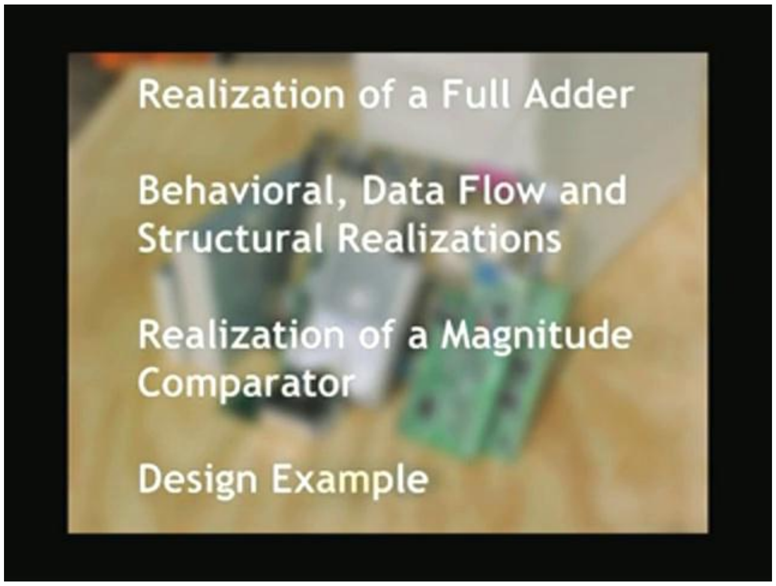

Introduction to Verilog

Levels of Abstraction

Realization of Combinational Circuits

Verilog Code for Multiplexers and Demultiplexers

(Refer Slide Time: 01:55)


Realization of a Full Adder

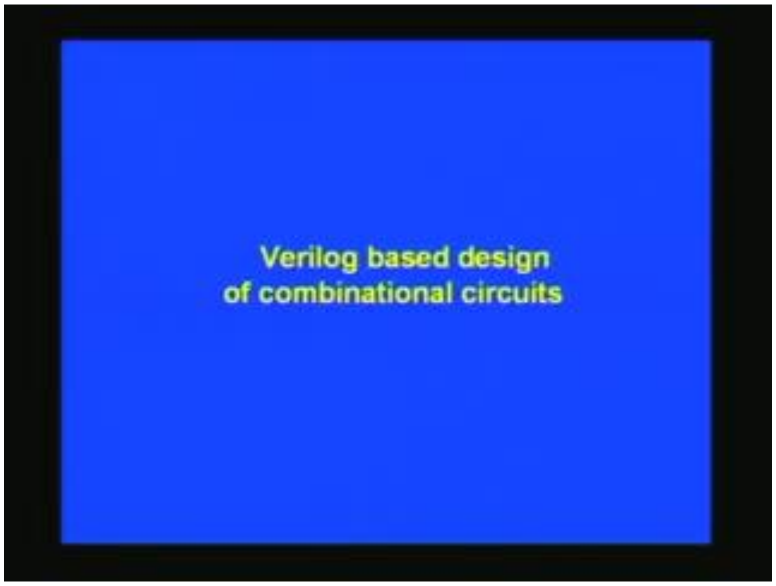
Behavioral, Data Flow and Structural Realizations

Realization of a Magnitude Comparator

Design Example

Welcome. You had a good review of digital circuit design and it is the right time for us to pickup HDL coding for the same. A digital system design is primarily a combinational circuit and a sequential circuit combined together in any mix as we will see, as we progress further. To start with, we will be learning about simple combinational circuits using Verilog and we will also cover little more complex sequential and combination circuits. Thereafter we will be going on to sequential circuits and ultimately proceed to digital VLSI system designer using Verilog.

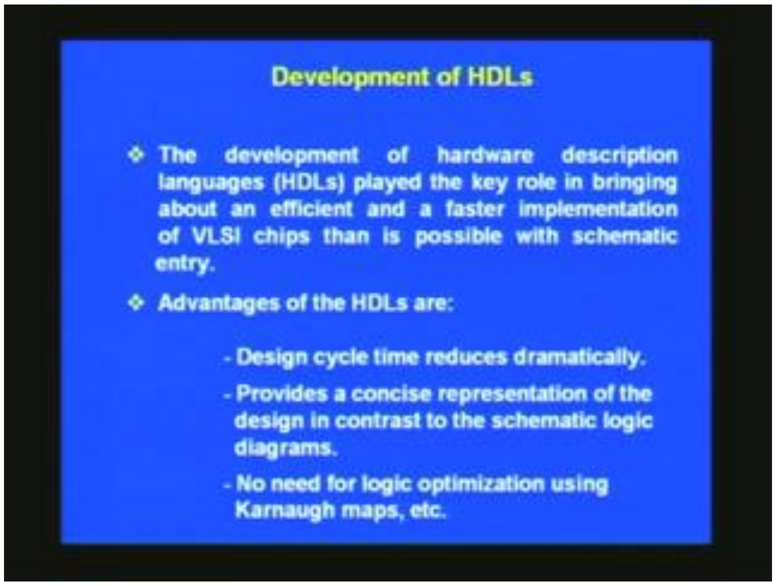
(Refer Slide Time: 03:16)



Verilog based design
of combinational circuits

Before we go into combinational circuits, we will just see the evaluation of HDL.

(Refer Slide Time: 05:50)



Development of HDLs

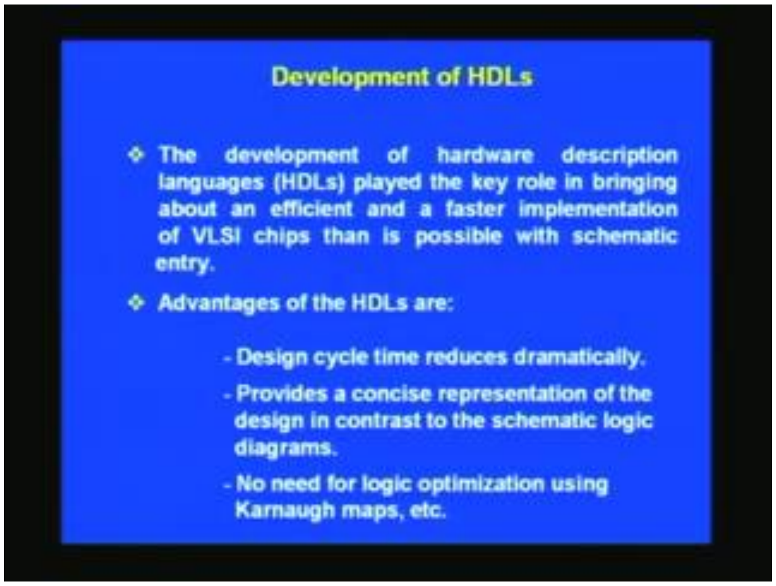
❖ The development of hardware description languages (HDLs) played the key role in bringing about an efficient and a faster implementation of VLSI chips than is possible with schematic entry.

❖ Advantages of the HDLs are:

- Design cycle time reduces dramatically.
- Provides a concise representation of the design in contrast to the schematic logic diagrams.
- No need for logic optimization using Karnaugh maps, etc.

Primarily, HDLs were used in order to speed up design cycle time. Hitherto, schematic circuit diagrams were used and it was very handy for designers who had been accustomed to design a system using discrete digital ICs which you know 74 series and so on. The problem with the schematics when it comes to VLSI design is that, we will have to struggle with hundreds of drawing sheets of the size A1; it has other disadvantages such as entry time is very high; you had to use Karnaugh, Mcluskey, Queen Mcluskey methods of optimization; you had to wait through hundreds of drawing sheets; readability is poor; these disadvantages outweighed the need for HDLs.

(Refer Slide Time: 04:39)

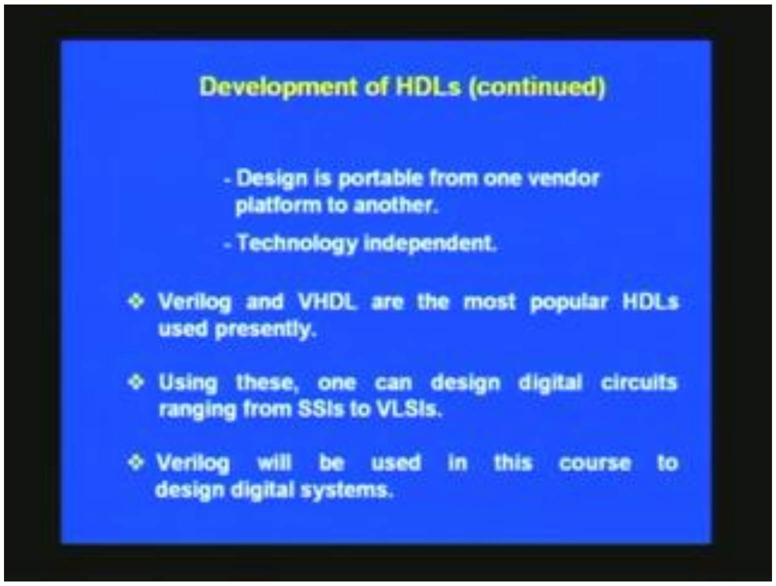


Development of HDLs

- The development of hardware description languages (HDLs) played the key role in bringing about an efficient and a faster implementation of VLSI chips than is possible with schematic entry.
- Advantages of the HDLs are:
  - Design cycle time reduces dramatically.
  - Provides a concise representation of the design in contrast to the schematic logic diagrams.
  - No need for logic optimization using Karnaugh maps, etc.

The HDL has brought about a revolution in a digital circuit design in the sense that the cycle time reduces very dramatically. From my experience, I can say, there is 3 to 5 times speeding up of a cycle time, I could see. It provides a very concise presentation in contrast to schematic logic diagrams. There is what is called behavioral statements and artial statements and all these can lead to very concise descriptions of the digital hardware that you are designing. In contrast to this, in schematic you had to really put gate by gate and it will take a quite an effort to achieve the same end result. And there is no need for queen map and queen mecholose way of reduction because synthesis tool is supposed to take up this role.

(Refer Slide Time: 08:01)



Development of HDLs (continued)

- Design is portable from one vendor platform to another.
- Technology independent.

❖ Verilog and VHDL are the most popular HDLs used presently.

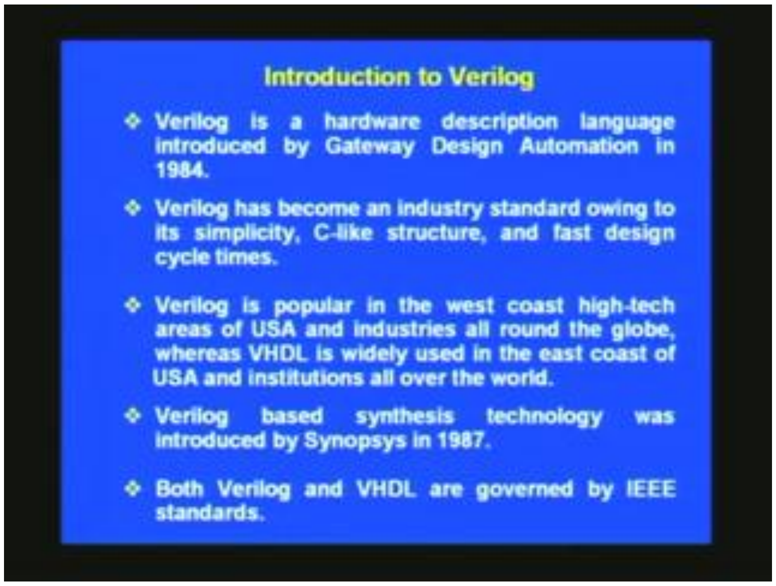
❖ Using these, one can design digital circuits ranging from SSIs to VLSIs.

❖ Verilog will be used in this course to design digital systems.

HDL designs are portable from one vendor platform to another. For example, if you have developed your HDL let us say, for Altera platform you can always migrate to another vendor such as Zylinks. It is also technology independent; in the sense a few years back, it was point 65 micron technology; thereafter point 5, then point 35, then point 135 and now, they are embarking on point 09 micron technology. All these technological changes had no effect on HDL and that is to clarify, whatever you have designed earlier when the technology was low, the same designs will precisely work with the present technology and most assuredly, on new technologies as well.

Two most popular HDLs used currently are: Verilog and VHDL. This is within the realms of digital circuit design. Recently, Cavins has come with mixed Verilog log and digital circuit design and they call it as Verilog AMS. You can implement A to D converter and D to A converter and PLLs and the rest of analog circuits and you can mix with digital circuit. It is needless to emphasize that Verilog and HDL can be used in a circuit design ranging from SSIs to VLSIs. You may regard up to 500 transistors to be falling in SSI category; beyond that and below 5000 for MSIs; beyond 5000 up to 50,000 for LSIs and beyond that they are termed as VLSIs. There are number of transistors on what I am talking.

(Refer Slide Time: 09:50)



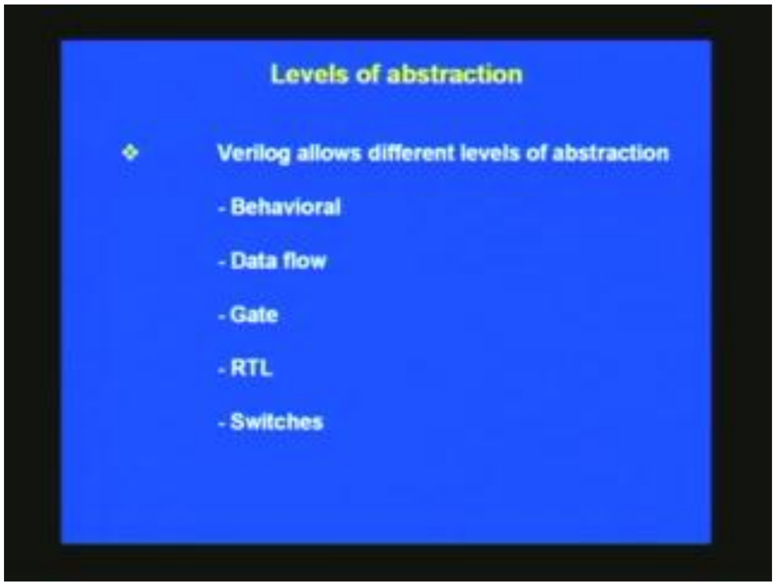
Introduction to Verilog

- Verilog is a hardware description language introduced by Gateway Design Automation in 1984.
- Verilog has become an industry standard owing to its simplicity, C-like structure, and fast design cycle times.
- Verilog is popular in the west coast high-tech areas of USA and industries all round the globe, whereas VHDL is widely used in the east coast of USA and institutions all over the world.
- Verilog based synthesis technology was introduced by Synopsys in 1987.
- Both Verilog and VHDL are governed by IEEE standards.

We will be using Verilog in this course for a digital VLSI system design. Verilog is a hardware description language; it was developed originally by Gateway Design Automation in 1984 and Cavins popularized it later on. In 1987 synthesis was introduced using Verilog by Synopsys - one of the leading vendors in ASIC platform. Verilog has become an industry standard as you know and because of its simplicity you can quickly learn Verilog. In fact it is easier than learning VHDL in my experience. It has a C like structure and very fast design cycle times. This C like structure is what I am saying you can use 'if', 'then', 'else' statements, case; 'for', 'while' and the rest of it and even I/O structures are more or less similar and of course, there are differences which we will be learning gradually.

This Verilog has been very popular in the hi-tech areas of USA and the west coast belt; whereas, VHDL in the east coast. The real reason may be that industries preferred Verilog and institutions preferred VHDL. But, it is only a general observation and it is not; there are always exceptions. As a designer you can start with Verilog first;; having mastered it you can switch over to VHDL later on. Because it in turn depends upon what the customer requirement is. In this connection, we will take Verilog in this course and specialize on it. Both Verilog and VHDL are confirmed to IEEE standards. We have already seen that Verilog is a hardware design language which is very much akin to a C but, you should bear in mind that Verilog is a hardware design tool and not a software design tool. C is clearly the programming language which basically runs sequentially unless you redo it by calls and the rest of it. Whereas, Verilog although it resembles C program, but, you should know that they are all concurrent statements. It is precisely the same as a digital circuit design making use of the conventional TTL gates.

(Refer Slide Time: 11:25)



Levels of abstraction

- Verilog allows different levels of abstraction
  - Behavioral
  - Data flow
  - Gate
  - RTL
  - Switches

Verilog allows different levels of abstraction one is behavioral what we have already seen 'else' I mean structure and 'while' for all those things I would form a behavioral. There is another structure called data flow structure. This is basically a concern with a flow of data from one place I mean one register to another and so on. We will see quite many examples as we progress. For those who are used to gate level simulation as in schematic entry earlier, they can still use gate level primitives. If you want get extra timing closures probably you may have to use gate level primitives as well. Next most important thing is Register Transfer Level and it is called RTL. This is the core of digital design; if you violate RTL synthesis tool will promptly reject.

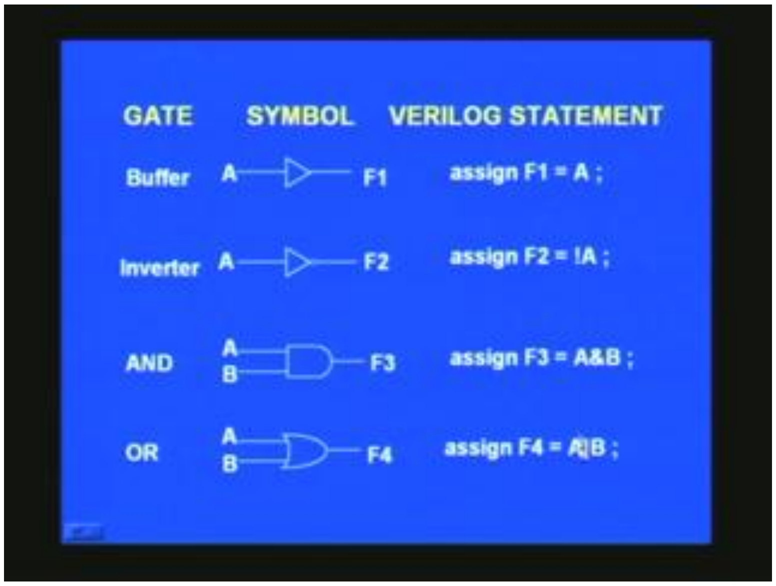
This is only to ensure that the final product that we are going to tap out will be really working on the hardware that you have is meant for. Finally we can use even switches like any NMOS or any MOS, in case you need. The problem in these switches is, it is technology dependent; the best what will be concentrate on this present course will be this artial type and occasionally going for data flow and behavioral type; at very rare occasion we will go on gate level as an illustration we may consider a gate later on.

(Refer Slide Time: 13:26)

Combinational circuit realization using "assign" statement

So we will start with the combination circuit using simple assigned statements. It is as simple as what you read there as assigned and some of the examples that we will go is for very primitive gates. To start with we would like to make a buffer; what all you have to do if you put the input here it is A, the output naturally after the buffer is once again A, so the output is F_1, A is the input. So if you see the corresponding Verilog statement is assigned this is the mandatory in order to assign any statement. This is the output on the left hand side equal to and it is almost like a C program with the exception that assigns a new thing here is because this is a hardware field. Similarly, if you want an inverter all you have do is instead of A just invert. This explanation mark is an inversion signal and this is a basically logical inversion; we can use it for multi bit precision also - this very same symbol. In this place you can use this symbol delta which is the bit wise negation; we want to do AND or OR with 2 inputs. It is once again as simple as that what you need is one more symbol we had known; it is an ampersand and this vertical slash here.

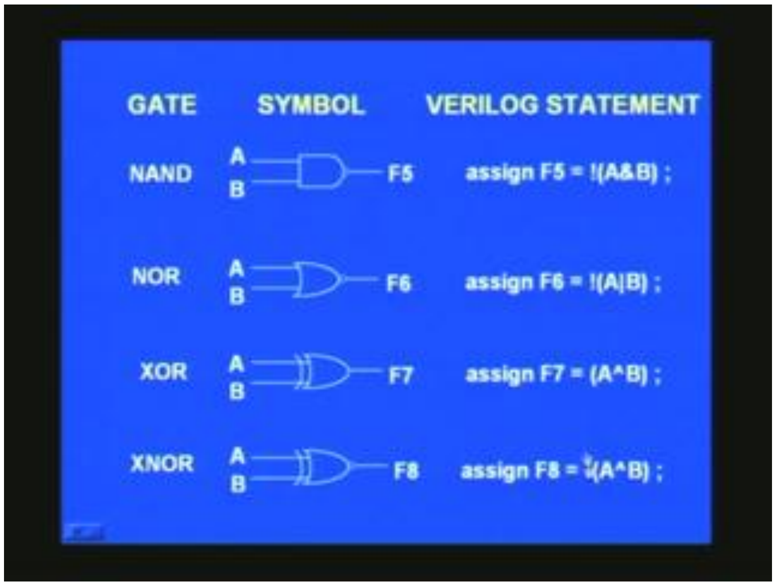
(Refer Slide Time: 15:00)



GATE	SYMBOL	VERILOG STATEMENT
Buffer	A F1	assign F1 = A ;
Inverter	A F2	assign F2 = !A ;
AND	A B F3	assign F3 = A&B ;
OR	A B F4	assign F4 = A|B ;

This is for OR, this is for AND (Refer Slide Time: 15:04); likewise whatever gates you have already derived you can get its inversion. For example, NAND which can be derived from complementing AND; so what you had A and B earlier use the same inversion symbol here. And so, is not the case for NOR once again use here inversion here and for exclusive OR exclusive NOR once again you see the inversion and for exclusive OR the symbol is the hat.

(Refer Slide Time: 15:31)



GATE	SYMBOL	VERILOG STATEMENT
NAND	A B F5	assign F5 = !(A&B) ;
NOR	A B F6	assign F6 = !(A|B) ;
XOR	A B F7	assign F7 = (A^B) ;
XNOR	A B F8	assign F8 = ~(A^B) ;

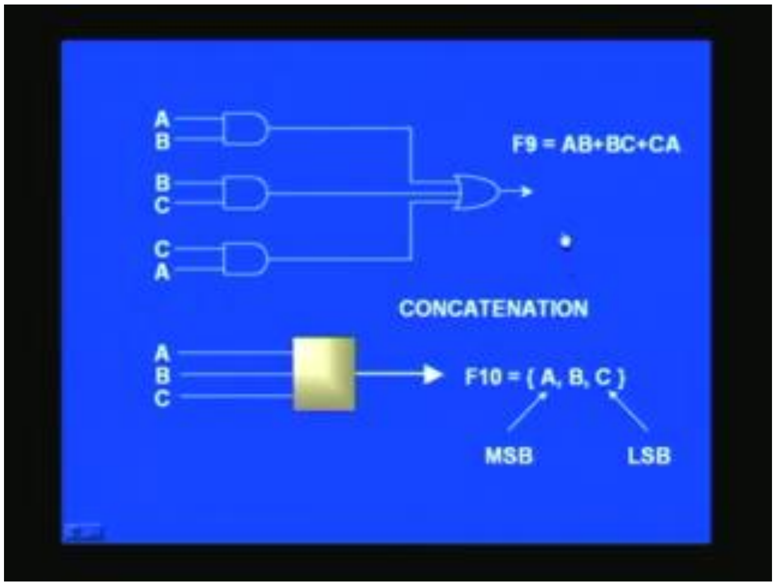
Now, that we have seen how to use assign statement for simple combinational circuit.

(Refer Slide Time: 15:48)

Combinational circuits using "always" statement

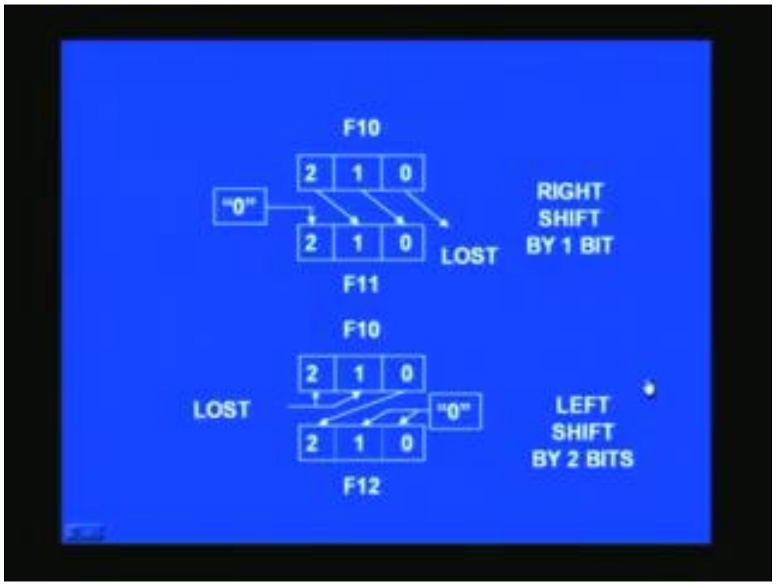
We can also implement the same using 'always' statement. Always is actually a block of instructions which will be seen from sum of these and a prior writing a code for this (Refer Slide Time: 15:58).

(Refer Slide Time: 16:01)



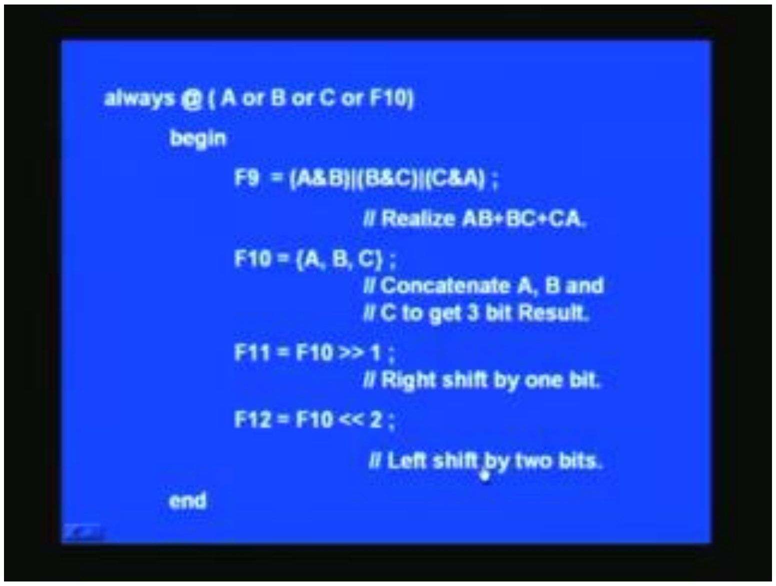
We will just take simple circuits such as the majority logic here. This is nothing but AB plus BC plus CA all single bit signal as such and you need 3, 2 input AND gates and 1, 3 input OR gate in order to achieve this. Next example we can consider how to concatenate different signals. For example, ABC is each 1 bit signal. You can put it concatenation just putting it together in the order that you want. This is the flower brackets which has signals. The compiler is a concatenation and you separate the signals and one may be the tempted to say it is a variable. That is because of our past habits linked to the C language. Here I do not speak in terms of variables but as signals; any digital node is a signal; whatever I/O you have in your system is a signal. As usual, you can assign this final output to any other signal that you want. Let us say when you do this it implies this A is in the MSB so always left side will be the MSB in all the treatments here. Naturally, it follows the last bit is the LSB.

(Refer Slide time: 17:36)



Next example will be considering is right shift and left shift. We will shift by just 1 bit in these case and 2 bits in this case. When you shift let us say we have a signal which is 3 bits wide; in fact this F10 (Refer Slide time: 17:36). If you notice it was derived from concatenating 3 signals. After right shift what happens is… first, these are all the weights you know that 8 4 2 1 codes. Probably, I should have put it or right on the top this is not a data as such. It is a bit position that I have what about here and once you right shift this one, what you are going do is just shift it here - this 0 bit position (Refer Slide Time: 18:25). Whatever data is in that position it will get lost because we are not storing it. If you want to store that you can use some other Verilog statement to store in case the need arises. Otherwise you do not really bother about that. Similarly 2 and 1 will occupy 1 and 0 of the target. Once this 2 has been used up this becomes vacant an unoccupied bit is 4 still 0. If it is left shift like this with a 2 shift here, then naturally this bit will come over 2 and these 2 bits will get lost and in such a case these bits vacated will be occupied by 0s.

(Refer Slide Time: 19:16)



```
always @ ( A or B or C or F10)
    begin
        F9  = (A&B)||(B&C)||(C&A) ;
                    // Realize AB+BC+CA.
        F10 = {A, B, C} ;
                    // Concatenate A, B and
                    // C to get 3 bit Result.
        F11 = F10 >> 1 ;
                    // Right shift by one bit.
        F12 = F10 << 2 ;
                    // Left shift by two bits.
    end
```

Now what we have seen over 2, 3 pages we have contained the entire thing in a single page here - the program (Refer Slide Time: 19:24). In fact, we can recollect what we had done earlier is that we have realized the majority logic which is AB plus BC plus CA. So note that ampersand stands for AND, vertical slash for OR and thus does how you get AB BC plus CA. This is precisely part of a program as Verilog code and you would notice there is 1 'always' statement here and what are all it means is whenever inputs such as ABC and F10. If any these inputs changes only then this will become active otherwise not. This instruction will be executed only when any of these input changes from the previous value. If you have multiple statements like this, you need to put 'begin' and 'end' indicating the whole thing is block. In one block you can put on as many instructions as you want. Another thing is, when the statement is complete you just end up with a semicolon there.

If you forget this, compiler will promptly report there up as such and will be learning all those gradually. Next example was concatenation as they have already seen. It is precisely what we have seen pictorially there and these are the square brackets which concatenates ABC signal. Suppose you want C as MSB here, (Refer Slide Time: 21:05) usually I put C here and then A comma B in any order.

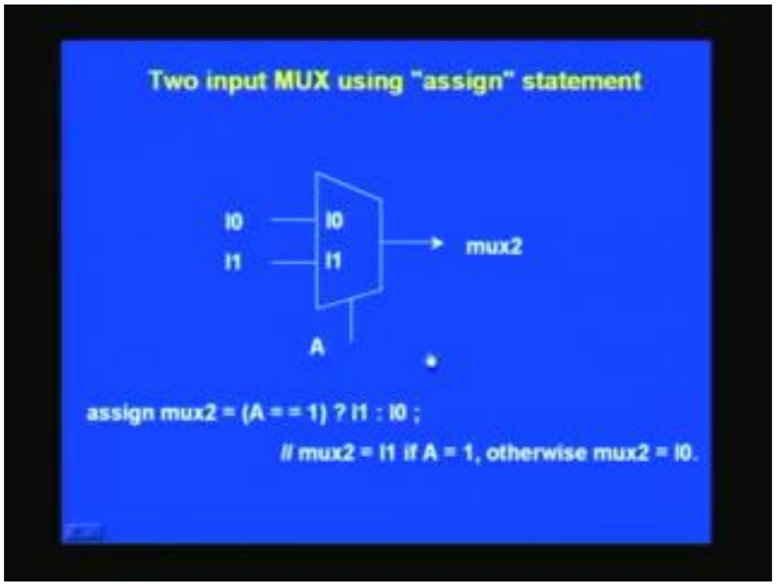
Suppose we want to add some 3, 0s at the LSB. What all we had to just add this put comma here and then say 3 stroke B 000 or any other data you want. If you put 111, 111 will be concatenated towards the LSB side. In this case, you can insert anywhere you can. Instead of that you can put any other signal which already is available here. We have also seen shift registers right shift by 1 bit here and 2 bits here. It is as simple as just one statement here. What happens here is after right shifting by 1 bit it does not affect this F10. It is the source. It can affect some other signal in this case, named as F11. That is the beauty of this statement here. In the case with left shift, you may also note that you can put any number of bits; all at one stroke it will do this shifting. Suppose you want 10 bits shifting, you just replace this 2 by 10. (Refer Slide Time: 22:10).

(Refer Slide Time: 22:25)

Verilog models for Multiplexers

Now, we will go into little more complex things such as multiplexer which you have already covered in the review.

(Refer Slide Time: 22:25)


Two input MUX using "assign" statement

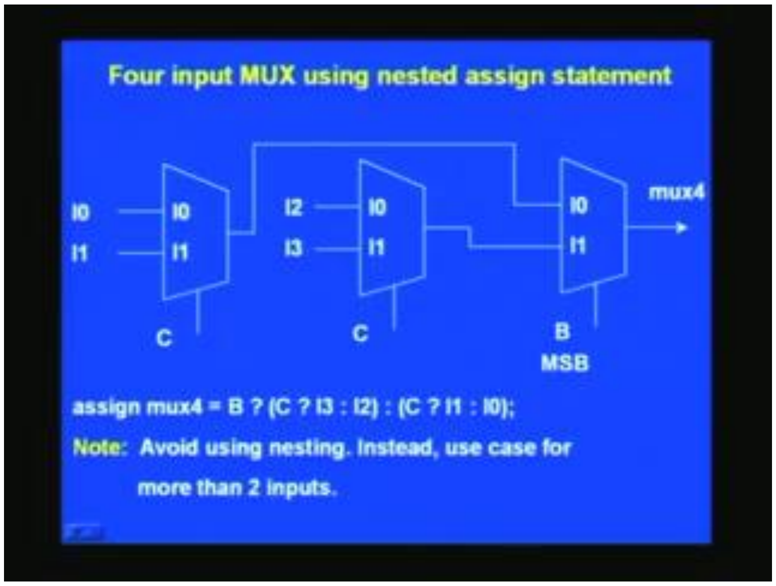
I0
I1
I0
I1
mux2
A

```
assign mux2 = (A = = 1) ? I1 : I0 ;
                // mux2 = I1 if A = 1, otherwise mux2 = I0.
```

Let us consider 2 input MUX with I0 and I1 as the 2 inputs and A is the selected input. If A equal to 0, it automatically selects I0; otherwise, it selects I1 and outputs on the MUX 2. In order to write this view, what all we need as a single line, once again, using assign statement? Only difference is that you notice that as far as the output is concerned it is precisely same up to the equal. Here, you have a select signal and this select signal must be first put here and a within brackets you can put here and this is not the only way of writing (Refer Slide Time: 23:07). In fact, you can dispense with all these here including the brackets.

I just have plain A; then also it will work or designers normally prefer putting one stroke B, 0 or 1, because, that will give an indication of how many bit precision this signal is. Here of course it does not arise and we can put it; need not be necessarily this alone. We can have one multi bit as an assignment here (Refer Slide Time: 23:51) and question mark is once again mandatory thing. For the MUX, this structure at once reveals there it is a MUX implementation and it has 2 inputs as we have already seen. The first input that you write is not this I0, please make a note of this, we had to write an I1 first and then followed by F colon which separates this input from this first input; this is clear I hope. Here, 2 slash - this is basically a comment field (Refer Slide Time: 24:25). Once you know this is a line comment, you can use a slash star and star slash for more than a group of lines. For a line you use this and you can either put it here for want of space, here I have just folded it here and shown here.

(Refer Slide Time: 24:51)



Four input MUX using nested assign statement

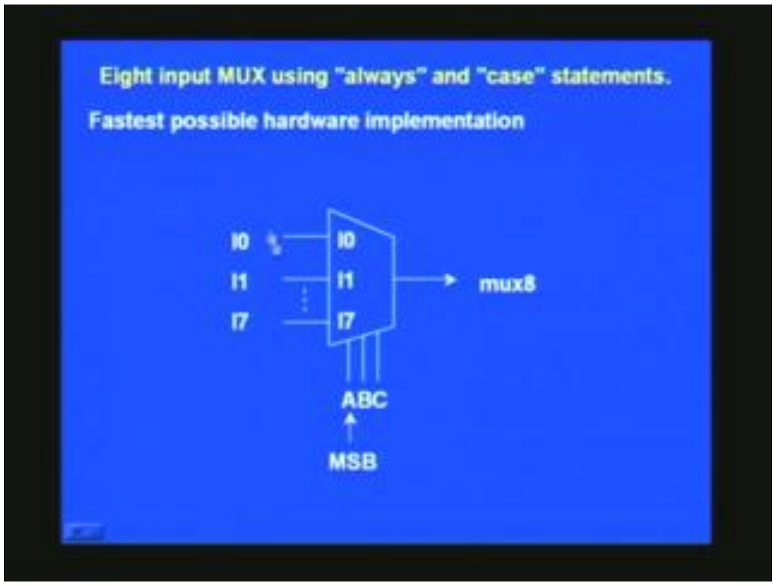
```
assign mux4 = B ? (C ? I3 : I2) : (C ? I1 : I0);
```

Note: Avoid using nesting. Instead, use case for more than 2 inputs.

Now we will make a little more complex by going for 4 inputs MUX. Let us consider I0 I1 I2 I3 is the 4 inputs that you want and naturally this calls for 2 selector pin. They are C and note that both have used the same C; here is B and this happens to be the MSB. We should be clear, which is MSB and LSB; otherwise, the whole thing will become chaotic because they only treat in that particular fashion. For example, this is always a higher order input and this one inside, precisely, the same 2 input MUX that we have used earlier, except that inputs and outputs are different. Once you say this one, whatever is the output will be actually be in this position and that particular thing will go as the second input.

For example, to make it more clear, B selects this MSB we have changed and if you notice here I0 I1 is coming over here. As the output of first MUX and second MUX output is derived from I2 and I3 here. So by selecting B for example, you select I mean make it 0, then this will be selected. When this is selected which of these will be selected depends upon C here (Refer Slide Time: 26:20). Naturally, you can very easily select one out of the 4 signals. In this case is this and for 1 its takes this, either it selects I0 I1 or I2, I3 and assigns it to the very final output MUX here. And a word of caution here, you should avoid nesting for the simple reason that you notice that there is a gate delay here and this ploughed to another MUX and that in turn will add more delay and if you keep this 1. Finally, we get a very delayed output and if you are to use this in between sequential circuits in the data path, naturally, your speed of operation will get hit this is not a good practice of nesting, so the designers have unwritten law saying that they will not go beyond two inputs. You use only 2 inputs MUX in this form with assign statement in this form. What is the remedy for this in real life situation? It is not merely 2; it may be 20 or 40 or even 100s. What you do for that? You had to make an efficient hardware design and the way out is to use case statements which will be seeing right now.

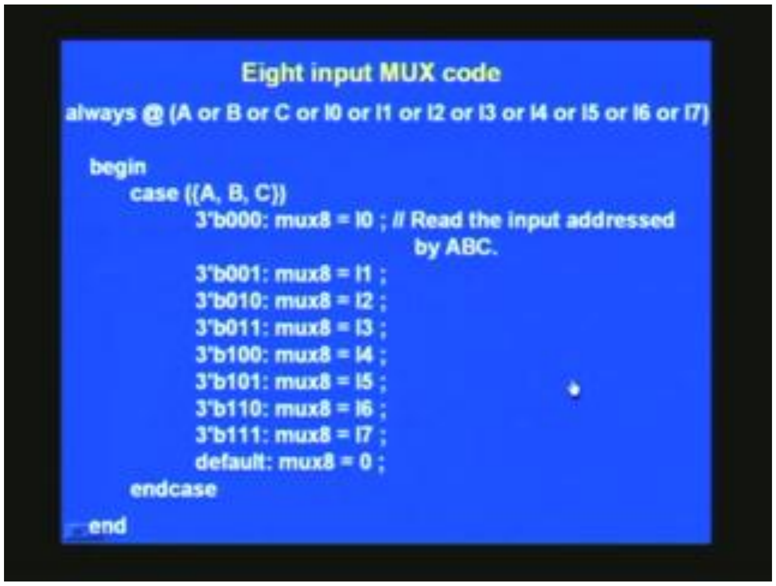
(Refer Slide Time: 27:46)


Eight input MUX using "always" and "case" statements.

Fastest possible hardware implementation

For example, we have gone to 8 input MUX as such I0 through I7 are the inputs; once again ABC you need 2 to power of 3 here for selection; once again ABC is the MSB which will have to keep track all the time; MUX 8 is the final output as and we will be using always case statements together in order to achieve this and notice that suppose I want 64 inputs. It is all child’s play we have only to just add few more statements which will be seeing.

(Refer Slide Time: 29:10)



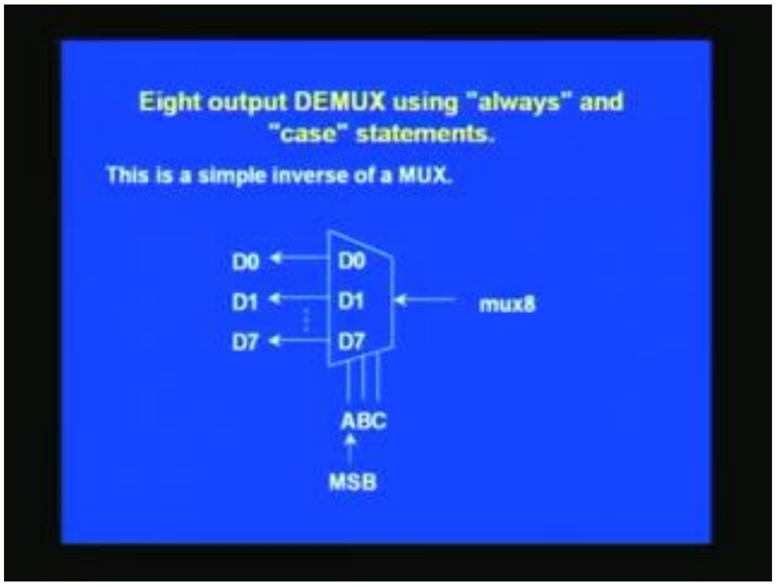
Eight input MUX code

```
always @ (A or B or C or I0 or I1 or I2 or I3 or I4 or I5 or I6 or I7)
  begin
     case ({A, B, C})
          3'b000: mux8 = I0 ; // Read the input addressed
                                   by ABC.
          3'b001: mux8 = I1 ;
          3'b010: mux8 = I2 ;
          3'b011: mux8 = I3 ;
          3'b100: mux8 = I4 ;
          3'b101: mux8 = I5 ;
          3'b110: mux8 = I6 ;
          3'b111: mux8 = I7 ;
          default: mux8 = 0 ;
     endcase
  end
```

What you have seen earlier is a MUX circuit; this you are already familiar. It is an always block with always a 'begin' here and 'end' here (Refer Slide Time: 28:33); now we have added one more statement called case here which is very much close to C case except that this actually hardware design; you should bear in mind. We have also seen a concatenation of 3 numbers here and that we put with in brackets.

This also is a mandatory thing that means single signal can be put here or any other multiple 'even' statement can be put here; what are all this means is when ABC each of them is 0 that is corresponding to this we should get is I0 at the output and similarly if it changes for different values let us say 111 naturally I7 must be selected output to the MUX 8 it is as simple as that. Notice that we have only put one statement here; occasionally you may have to put multiple statements. In which case you start do not fail to put begin and end towards end of that statement. When we will be going for a DEMUX implementation which is exact counter part of this then we can see more details on that.

(Refer Slide Time: 29:55)



Eight output DEMUX using "always" and "case" statements.

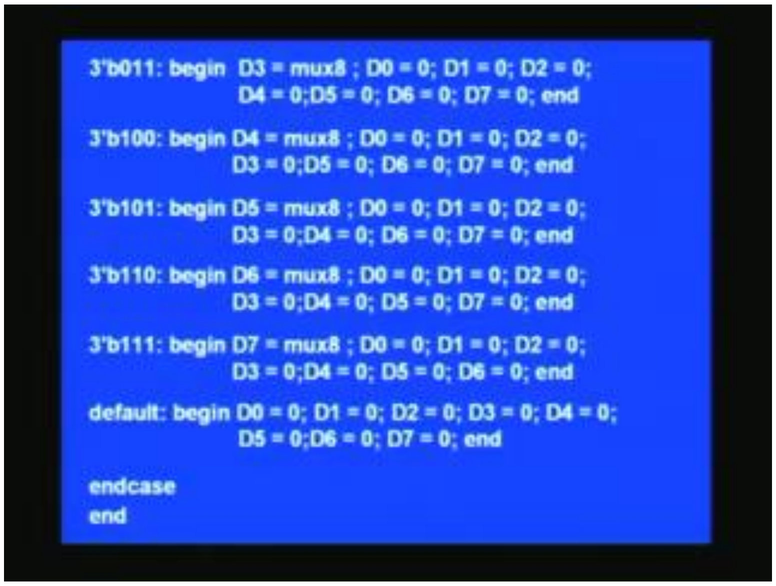
This is a simple inverse of a MUX.

As I said it is a inverse of MUX 8 that is the reason why I am putting MUX 8 which is the output in the earlier case as input here. Naturally after this process you naturally get back what you have ploughed in as inputs. This D0 should correspond to I0 so on I7 to D7. Once again A is the MSB here; this appears to be little more complicated because (Refer Slide Time: 30:30) this is once again using same always statement whatever the inputs there are being used here and note that MUX 8 is also an input here; you had to list all of them and notice that it is only OR no AND is valid here. It only says this signal or that signals and so on.

It is only an OR case there is no such thing as AND in there and there is no logic. This is not a logic thing just plain English statement. What it does here is, it reads MUX 8 as the input and depending upon ABC inputs, it assigns to that particular output that you really design. For example, if it is 000 we notice that MUX 8 must be routed to D0 output. That is precisely what is being done here in the statement. So MUX 8 which was the output of the multiplexer earlier 8 input multiplexer we have seen and assign it to this D0 here (Refer Slide Time: 31:39). Notice that D1 through D7 are all made 0s here because, when it enters it can enter from any other state. Unless you make them 0s here, it may hamper with the performance. It may come from latch and it may latch on to the old signals and it is better to give this here and you may notice that this is done deliberately for every value here but this need not be done.

You can even put all these from D0 through D7 right before this case just after begin and put as a single statement and depends with the use here. You just cannot do whatever output that you want; only that need to be listed. I have made it deliberate in order to drive home the point that we should not forget whatever it is to be forced to 0. They will have to be deliberately forced to 0; otherwise tool will not do that on its own.

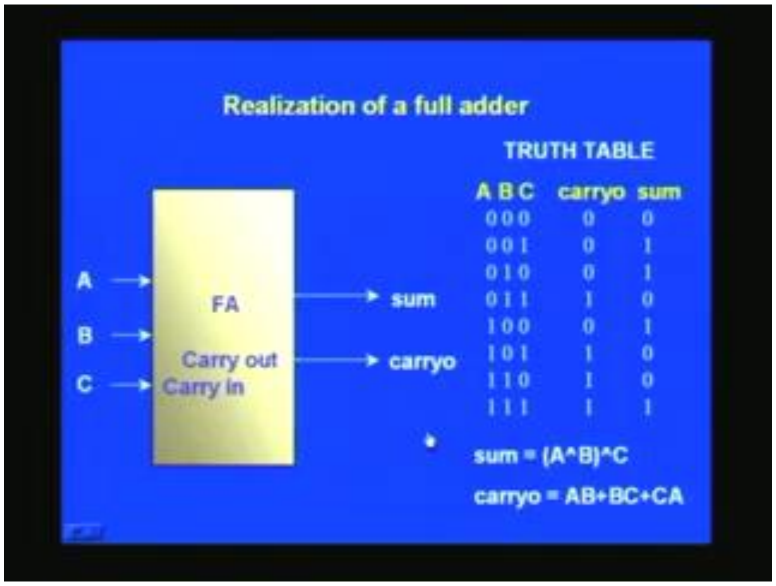
(Refer Slide Time: 32:46)



```
3'b011: begin  D3 = mux8 ; D0 = 0; D1 = 0; D2 = 0;
               D4 = 0;D5 = 0; D6 = 0; D7 = 0; end
3'b100: begin D4 = mux8 ; D0 = 0; D1 = 0; D2 = 0;
              D3 = 0;D5 = 0; D6 = 0; D7 = 0; end
3'b101: begin D5 = mux8 ; D0 = 0; D1 = 0; D2 = 0;
              D3 = 0;D4 = 0; D6 = 0; D7 = 0; end
3'b110: begin D6 = mux8 ; D0 = 0; D1 = 0; D2 = 0;
              D3 = 0;D4 = 0; D5 = 0; D7 = 0; end
3'b111: begin D7 = mux8 ; D0 = 0; D1 = 0; D2 = 0;
              D3 = 0;D4 = 0; D5 = 0; D6 = 0; end
default: begin D0 = 0; D1 = 0; D2 = 0; D3 = 0; D4 = 0;
               D5 = 0;D6 = 0; D7 = 0; end
endcase
end
```

This is a continuation of the same thing assigning different things. Finally, if you see for 111 input you have MUX 8 assigned to D7 and we have one default here. There is no guarantee these inputs are exactly X, valid number here and it may be an X that is I do not care or even a Z, a tri state condition. In other cases, you may not have use all these conditions. In such a case this default is a mandatory thing otherwise it infer a latch and you will run it to problems later on. Finally, when there was a case corresponding end case must be there and since we put the begin of the block there must also be an end. If you forget any of these including semicolons etc., prompt the error message comes while compiling.

(Refer Slide Time: 33:49)


A
B
C
FA
Carry out
Carry in
sum
carryo

Realization of a full adder

TRUTH TABLE

A B C	carryo	sum
0 0 0	0	0
0 0 1	0	1
0 1 0	0	1
0 1 1	1	0
1 0 0	0	1
1 0 1	1	0
1 1 0	1	0
1 1 1	1	1

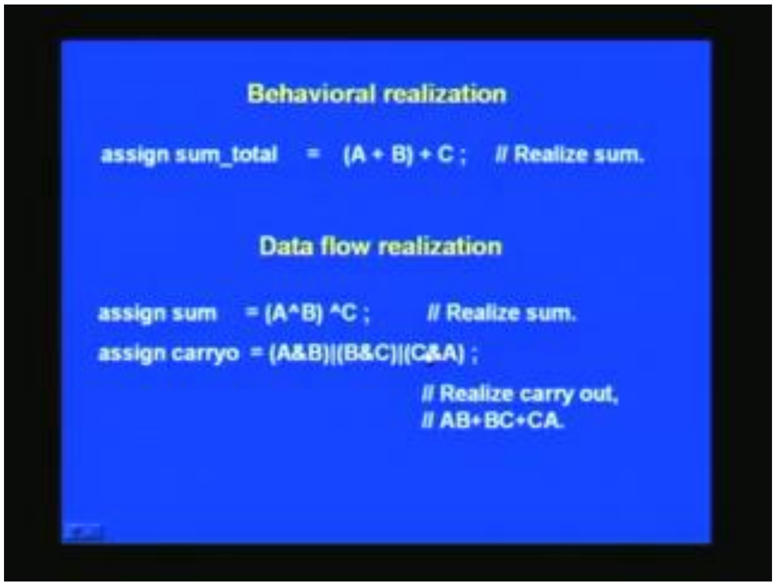
sum = (A^B)^C

carryo = AB+BC+CA

We have also seen full adder in our review; this is duplicated here for ease of understanding. You have ABC as 3 inputs; this may be regarded as a carry in as I mentioned here (Refer Slide Time: 34:05); there will be a carry out and a sum. FA stands for Full Adder; corresponding truth table is given here (Refer Slide Time: 34:16). This is nothing but exclusive OR of 3 input exclusive OR the inputs we just add up all these things what you get 00 that we have got here; here in add up it will be 0, 1 and so on it goes if you add up last two it means 1 0 and if you add this 1 to that 1 what you get is 1 1 and as simple as that; hence this expression is nothing but why this bracket is put in a Verilog we can implement this using gate level primitives also.

They are restricted with 2 inputs basically; if I0 to have realized a 3 input exclusive OR you will have to use 2 numbers of 2 inputs exclusive OR and carry out is simply majority logic and you can easily infer the majority logic. Wherever more than 2, 1s are there you will see the corresponding 1 here, and then see here (Refer Slide Time: 35:25).

(Refer Slide Time: 35:26)


Behavioral realization

```
assign sum_total   =   (A + B) + C ;    // Realize sum.
```

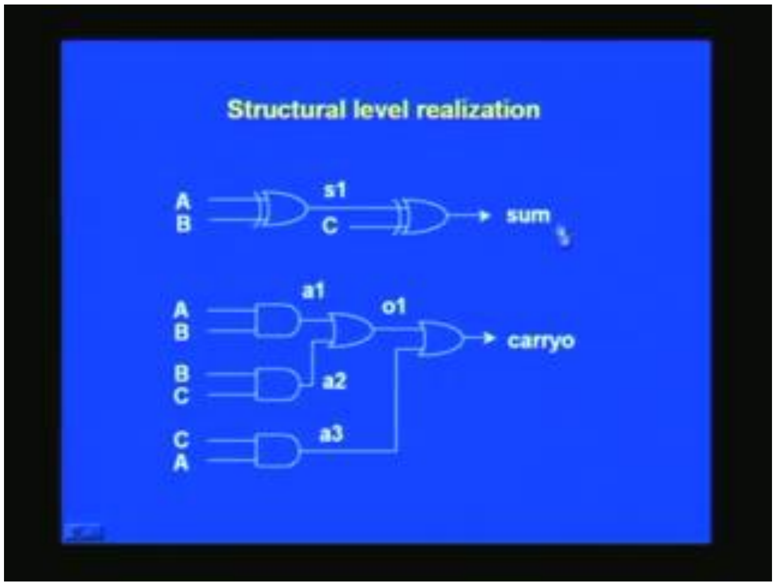
Data flow realization

```
assign sum    = (A^B) ^C ;        // Realize sum.
assign carryo = (A&B)|(B&C)|(C&A) ;
                                   // Realize carry out,
                                   // AB+BC+CA.
```

There are 3 ways of implementation of the same full adder: one is we have already seen like the earlier stage that we have behavioral type of level of realization. Suppose you can just put an simple arithmetic expression as such A plus B and C and this parenthesis is only to guide the synthesis tool to make it very efficient and if you do not put this 1 it may infer as internally 2 MUX and so on. Instead of that only 1 MUX will be put if you guide that synthesis tool, that is why this parenthesis. It is a good habit to use parenthesis and you will notice that what we have in a single line we can realize a full adder. Another beauty of this is, irrespective of the bit a precision as such, suppose you want 16 bit precision for A and B you can still use the very same statement as such. Only thing used to will have to declare A B to be of that precision on the top of a program that you will be covering it later on; another method of realizing the same thing full adder is to use 2 more statements.

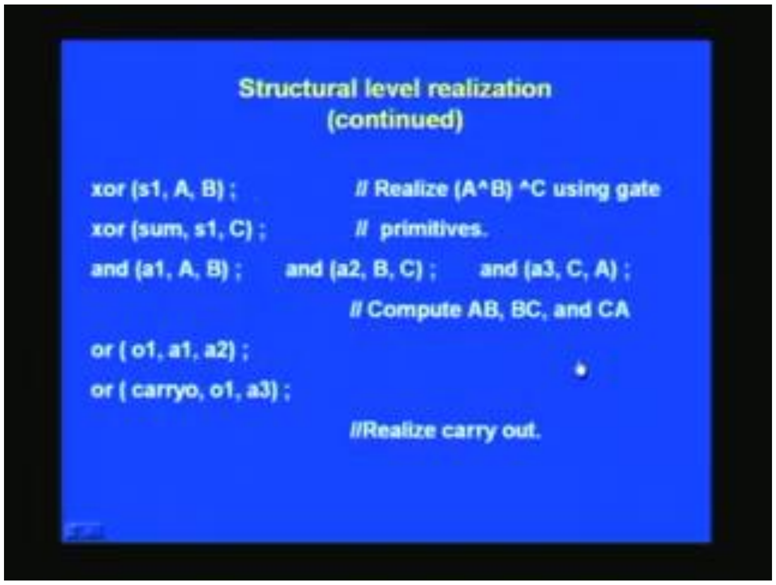
One is for the majority logic expression which you have already seen earlier, so adding of AB BC and CA. Then assigning it to carry out and some is nothing but as a said 2 input MUX, I mean 2 input exclusive OR is being used here and two times, that is the implication there and called as data flow realization and this will be helpful while pipelining your design and we will see more about pipelining later on.

(Refer Slide Time: 37:21)


Structural level realization

The very same full adder debited using gates is this and with an NB of primitive gate implementation it has been put in the slash on and note that s1 is an intermediate signal which will be using and similarly A1 through A3 are intermediate ANDed operation here and OR this 2 and once again O1 is an intermediate signal we will be using; final result is sum and carry.

(Refer Slide Time: 37:58)

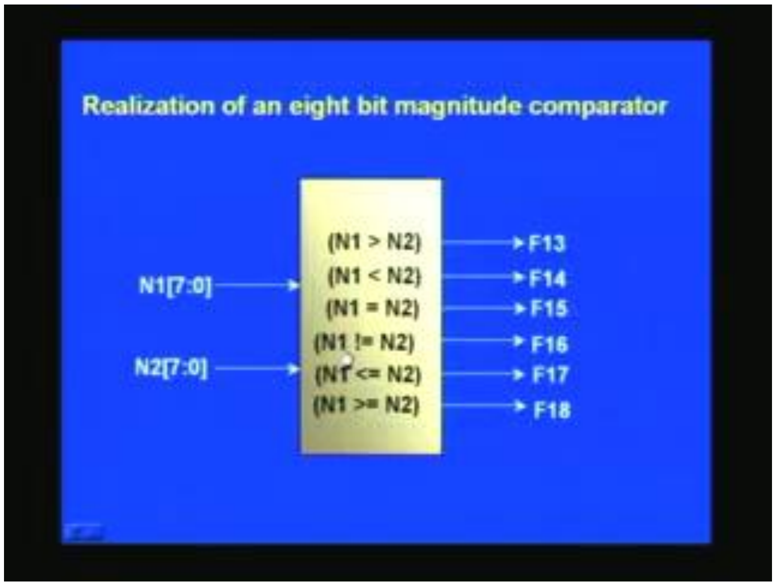

Structural level realization (continued)

```
xor (s1, A, B) ;              // Realize (A^B) ^C using gate
xor (sum, s1, C) ;            //  primitives.
and (a1, A, B) ;    and (a2, B, C) ;    and (a3, C, A) ;
                              // Compute AB, BC, and CA
or ( o1, a1, a2) ;
or ( carryo, o1, a3) ;
                              //Realize carry out.
```

In the next slide you will see the gate level implementation for the same and notice that you made to go for 3 5 7 lines here and this is a very much close into your standard schematic based editing, the entry right. So here what you see is that intermediate signal what you have seen; this is to invoke primitive which is available in the libraries of Verilog. This is exclusive OR which you are already familiar; similarly this is AND and OR (Refer Slide Time: 38:36) and first one stands for the output of that gate; whereas rest will be the inputs.

As I mentioned there all restricted 2 inputs normally and some offer higher inputs as well but that is not guaranteed on all the platforms. Finally, this provides the next exclusive OR along with the third signal which you have not used earlier to get the final sum output here. Similarly, you can AND AB BC CA and OR this A1 and A2 here; assign to an intermediate signal here and once again you can use this and continue ORing this A3 which you had not used earlier. Thus, we have got carry out as well as sum here.

(Refer Slide Time: 39:28)

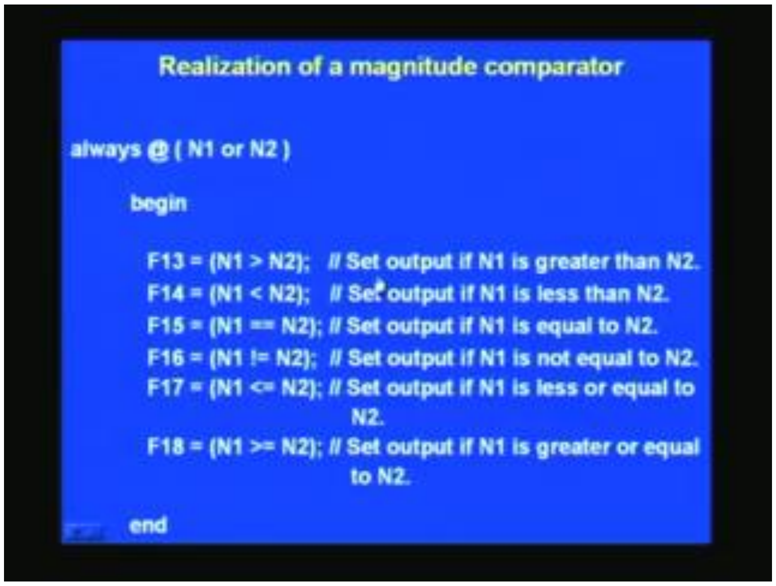


Realization of an eight bit magnitude comparator

Before we close this session, we will see how to make magnitude comparators. Let us consider we have 2, 8 bit numbers N1 and N2. If N1 is greater than N2, we would like to have one output and you can have here; you just label them when we show the program we will be using the very same nomenclature and these are all other possibilities.

Suppose you want N1 less than N2, then activate this output and so on; this is for equality this is for not equal to. You are already familiar with exclamation mark for inverting and this is symbol for less or equal to and this is greater or equal to and you may get confuse with this symbol later on and they are called non blocking statement which will be used in always block. We will see more details later on. Let us see how the program goes.

(Refer Slide Time: 40:43)



Realization of a magnitude comparator

```
always @ ( N1 or N2 )
     begin
       F13 = (N1 > N2);   // Set output if N1 is greater than N2.
       F14 = (N1 < N2);   // Set output if N1 is less than N2.
       F15 = (N1 == N2); // Set output if N1 is equal to N2.
       F16 = (N1 != N2);  // Set output if N1 is not equal to N2.
       F17 = (N1 <= N2); // Set output if N1 is less or equal to
                             N2.
       F18 = (N1 >= N2); // Set output if N1 is greater or equal
                             to N2.
     end
```

Once again we have used an always block and you have to list all the inputs either N1 or N2 here and there will be begin and end; all the outputs that you have already seen are appearing here on the left hand side here (Refer Slide Time: 41:03) Precisely the same statement which is put here as the output only needs to be inserted here in this as you has seen here. N1 greater than N2, you assign into F13; this is the comment what you have to put.

For all other things, this is logical equivalence you had to put 2 equals; whereas in the ordinary else where other than this you need to have only 1 ‘equal to’. This is a ‘not equal to’ and ‘less’ or ‘equal to’ and ‘greater’ or ‘equal to’; I think may have covered all the possibilities. Let us consider a design example using some of the methods that we have adopted earlier and this example is a simple summation of 2 numbers and then compare it with preset value and then output different results.

For example, you have a number 1, number 2, numbers each of 8 bits and we want to take the sum of these 2 and output here provided enable sum is active; that is logic high. Further, we wish to compare the sum with preset value, which is also another user input just like number 1 and number 2 and then output; if they two match, that is sum and preset value matches then you say activate one match signal.

Similarly, if sum is greater you activate more if it is less, activate less. It is such a simple example and we will see how to write the code for these. We can use assign statements for this but however, always statement is better thing because we need to take action only when there is a change in the inputs.

So we will use the always statement. First these always block it says always ‘at’, this is mandatory. List all the input variables; sorry, you should always speak in terms of signals not variables - that is the C habit right? I will just list this enable sum and all the inputs. Then next one is number 1 and underscore, then that is 1 or is here then finally, PRESET VALUE. Now in order to create sum, we have already seen earlier 1 bit adders, in fact full adder and we have also mentioned in a behavioral statement that we can use even multi precision in that same add. We will see right now that particular thing is being done here .For example, you want to have a sum output so what all we have to do is just write sum then equal to that. We have also learnt earlier, how to model a MUX? We will combine that MUX in this ‘always’ loop. We should not use an assign for a MUX inside the ‘always’ block; that is the difference here.

What we want to do is we have to take enable sum into account while writing the sum. So let us say in the MUX the selector control is enable underscore sum and a MUX is identified by this question mark symbol. Now what you want is there are 2 inputs for the MUX. So the higher order input will be put here that is what we want here because when enable sum is 1 we want to add the 2 numbers. What will do is, we will put (Refer Slide Time: 45:39). Do you think the statement will work? We have not put the precision, it will still work and we can also make the precision here which is normally done by all designers because we do not have to go to the previous way through previous codes and find out what precision it is. We can write there in this statement we can identify that by just mentioning the range for example 7 to 0 and so on. Will this statement still work? It will not because we have not terminated base semicolon here and similarly many people especially now a days is, make the mistake of putting a logical OR; this is not a valid thing, What we have do is put OR.

Similarly, will write for the other 3 outputs that is match. Here also we need to put design signal I am not writing it here and now this time what we have to be if it is 1 then this is corresponding to the 1 input and that is sum equal to. (Refer Slide Time: 47:15) There is actually one more mistake here in this sentence (Refer Slide Time: 47:33). We have forgotten another input for the MUX. We will had to put a colon and then put what we want is just 0 at the output. If the enable is not valid, I mean it is not asserted then we need to put just 0 because we just want to make a distinction whether it has gone through the actual process or not. Similarly, here we need to put 0 here see this will had to put in with in parenthesis and note that this is a logical statement. For logical statement, just 1 equal if you put at the compiler stage it will reject, what we need to put is 2 equal statements.

For example you want to put an inversion, not equal to what all we have to do is just replace this one by an exclamation mark. That exclamation mark is for the logical and similarly, we can put ‘less’ as output, here I think I have missed one thing (Refer Slide Time: 48:50) that I can put here as same. Here it is less and here it is more (Refer Slide Time: 49:00); precisely the same statement will appear and likewise we have a 0 if the condition is not satisfied; that is to say as a further explanation what we say is the sum is equal to preset value both of which are multi precision bits and what the value it returns is actually 1. Are we clear on this? Are we together on this? Then I think that is all as simple as this code. We have forgotten one more thing, can you point out what it is? This is a multi statement; that is block we need to add ‘begin’ and ‘end’ to that, so we will put ‘begin’ here and ‘end’ here.

This completes the code for this. Now an assignment to you, I will suggest that we add one more input here saying that sum stroke difference (Refer Slide Time: 50:15). This is 1 bit input as such the sum is 0; what I want this output is sum otherwise I want difference to appear over that, so modify this accordingly. In a different case let us call it by a generic name as some D. So the same thing will become mod of 1 N num 2, this is clear (50:44) Conversation between professor and student this is 0, (51:15). Conversation between professor and student. Yes, this is this one, no these are all single precision. Therefore you do not have to mention it specifically as such to know it is a single signal as such right?

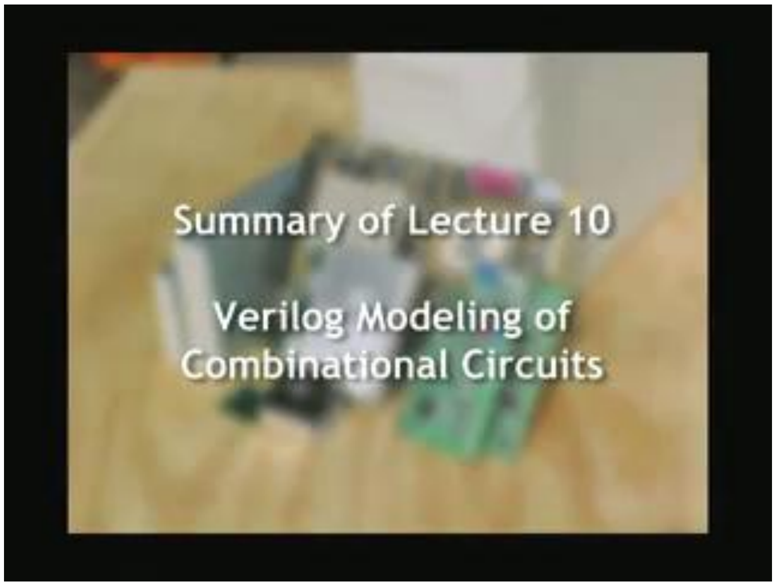
To summarize what we had done so for is we have used multi precision in the place of single precision because of full adder seen earlier. Next is, we have used precisely the same comparators and that is how we have got. We have also combined an 'always' block and same as a thing by putting assign statements. You can just make that one but what will happen when you put assign statement it keeps on turning continuously because it is precisely the gate that is going to be mapped on to the device.

So it will be working all the time. If the circuit is working it consumes more power, so the thing always is preferable because this will work only when change is encountered there, that is all the difference. To summarize the whole thing we have covered simple gates and the symbol for which is ampersand and OR that the vertical slope and exclusive OR cap so on; then we have seen how to use assign and 'always' block in order to achieve any combinational circuits. This way you can make any complicated combination circuits. The next class will be seeing how to model a sequential circuit.

Thank you!

Summary of Lecture 10

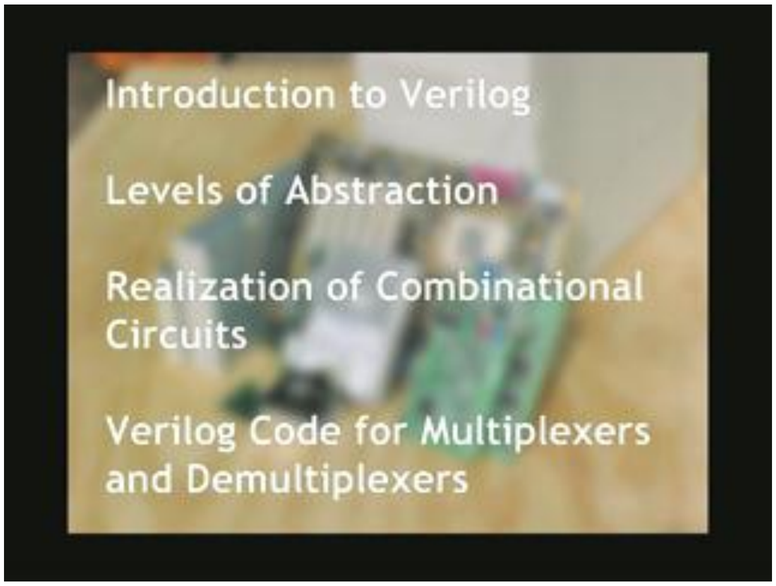
(Refer Slide Time: 52:48)



Summary of Lecture 10

Verilog Modeling of Combinational Circuits

(Refer Slide Time: 53:11)



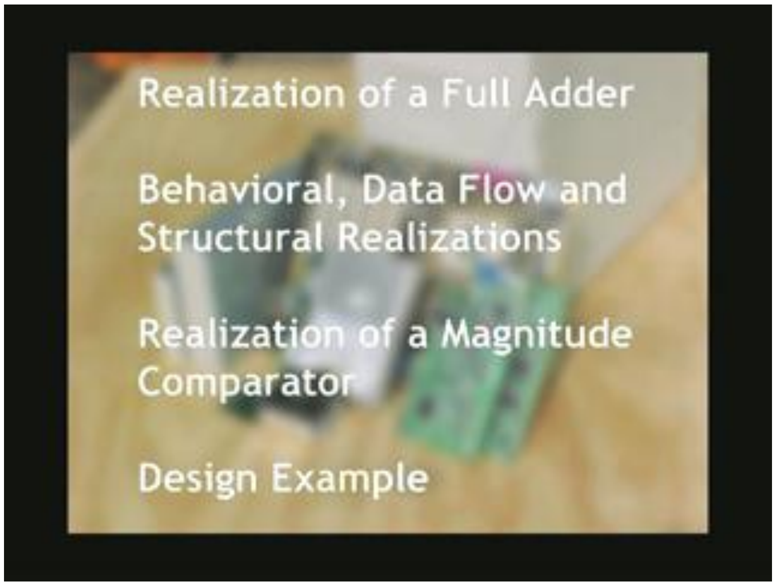
Introduction to Verilog

Levels of Abstraction

Realization of Combinational Circuits

Verilog Code for Multiplexers and Demultiplexers

(Refer Slide Time: 53:31)



Realization of a Full Adder

Behavioral, Data Flow and Structural Realizations

Realization of a Magnitude Comparator

Design Example