Digital VLSI System Design

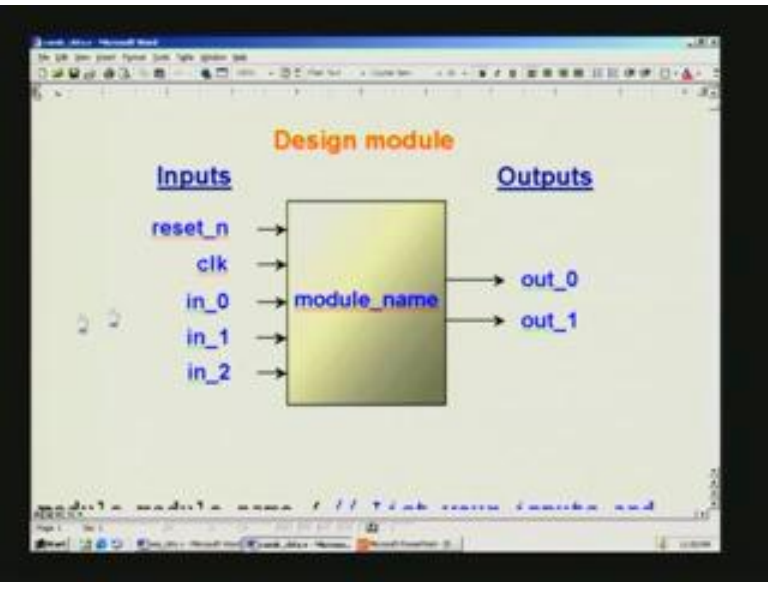
Prof. Dr. S. Ramachandran
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 14

Coding Organization – Complete Realization

We have seen how to design Verilog codes for combinational and sequential circuits. What we have seen is only piece meal codes; it is not a full-fledged code. Now, the present attempt is to make it full-fledged so that it is suitable for platform such as simulation and synthesis, and so on. We will call it by the name coding organization, because there are more statements to assign always and the like of it, which we have already seen earlier.

(Refer Slide Time: 03:25)

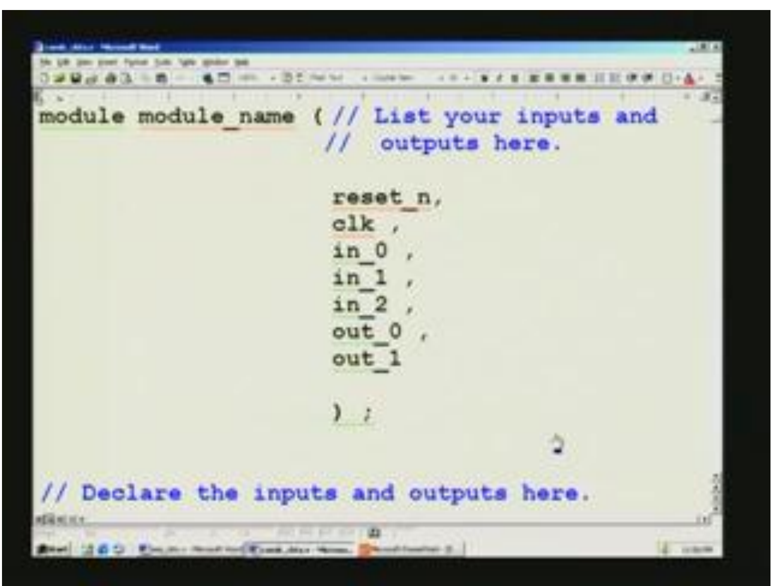


To start with, let us assume that your design is a module like this - just a block. To start with, we can regard it as a black box depending upon what functions you are going to integrate in your design, you can give a meaningful name to that. Let us say that we call it

module name; underscore is only to make the readability better. This particular module will naturally have some inputs and outputs, and these are all listed right here.

We have already seen the reset clock. In addition to that, we will have just three inputs for the sake of making it clear to you. This also has two outputs and they are out 0 and out 1. You can call this a design module or simply, the module. Now, how to code this in Verilog? You start writing on your paper with this block and then go on to code it.

(Refer Slide Time: 04:32)



The way to code is right here. First, we set the module name and that is what appears here; before that, we had to say it is a module. Naturally, the whole thing is identified by the module name which is particular to your design. This is a keyword in Verilog to identify there is a module here (Refer Slide Time: 00:04:56 to 00:05:12 min) and a corresponding end module at the end, which we will see towards the end of this piece. Here, note that there is a bracket. It starts there and ends here and is followed by a semicolon. Do not forget the semicolon or the braces mentioned here. We have already seen this line comment; so, you can freely use the line comments just to explain what you are doing in any part of the design.

Here, what you see is a list of some i/os that we have already seen in this block earlier. These things plus the outputs, all the i/os stand for inputs and outputs are listed here. Notice that each one is separated by a comma and also notice that there is no comma for the last one. If you violate any of this, naturally, the compiler will complain. After this, you have to declare first inputs and outputs. You have not mentioned here that you have not identified the actual inputs and outputs. So, you have to do the identification which is a logical sequence here.

(Refer Slide Time: 06:17)

```
// Declare the inputs and outputs here.

// Declare the registers and wire (or net).

// Describe your combinational and sequential

// circuits.

endmodule     // This indicates the end of
              //  design.
```

All these have been put in comments just for your clarity sake and this is not a full-fledged program code. The next item is registers and wire. What they are, we will see after a few minutes. They are basically signals used in always block for registers, and assign statements etc., will be wire or net; we will go into little more details, a little later. Then, you follow with your actual code. What we have already covered earlier in combinational and sequential circuits was precisely belonging to this category. As I said, wherever there is a module there must be an end module. So, in summary, you have nothing more than module, the module name, then followed by i/os, your declaration of inputs outputs, and then as well as registers and wires, then your actual circuit, and then finally do not forget to put an end module. Note that there is no gap here after end.

(Refer Slide Time: 07:32)

```
/*     COMBINATIONAL CIRCUIT REALIZATION

            USING VERILOG CODES

To start with, declare the module you wish to
design.

Note that the design file name is the same as
the module name, "comb_ckts".

.v is the extension to indicate that the
```

Here starts the actual code. We have already seen that in C we use this symbol for slash star here and a matching star slash at the end of the comments. If it is more than one line, you can use this as in C. What we are going to do is to realize a combinational circuit using Verilog codes. We have already seen these comments; we have to declare a module name.

(Refer Slide Time: 08:03)

```
To start with, declare the module you wish to
design.

Note that the design file name is the same as
the module name, "comb_ckts".

.v is the extension to indicate that the
design file is in Verilog.

*/


/*
  means multiple line comments like C.
```

What is new here now is that, whatever you have as modules in your design you will have to ultimately put it into a file. When you make it into a file, give the very same name that you have had earlier. For example, name of the module that you have given for your design. Make it a habit to use the very same notation, even for saving it as a file.

For example, the file: I wonder whether you can read it here; so, I will read it for you. It simply means com underscore circuits dot v. Do not forget the extension dot v to indicate that it is a Verilog and not v s t l. In v s t l, you say v s t etc., dot v is the extension that you have to give for any meaningful name that you have here. Once again, I repeat that circuits are nothing other than your module name and the filename should match with that.

You cannot nest multiple modules. In fact, I mentioned this in an earlier lecture also. If you have different modules, you have to put them in different files and call those modules here. What we refer to as an instantiation, we will cover while going in to the depths of those later on. When the design gets more and more complicated you have to resort to such methods. For simplicity sake, I have put all those that we have learnt in combinational and sequential, in just two files, and named them appropriately as sequential circuits for the other one. Here, we have already seen that this is the end of the comment; (Refer Slide Time: 00:09:43 min) here, there is more than a line and that is what is to be mentioned here and it means a line comment.

(Refer Slide Time: 09:47)

Taking the model, we have already put there instead of [.] here, we have used the module name. Now, we replace that by comb circuits, because our intension is to put all the combinational circuits that we have learned earlier into one file. Once again, we see there is a module here, and the code might be a long thing. So, you normally tend to forget an end module towards the end. If you forget it, the compiler will tell you. Initially, for the beginners it may be difficult. A good practice would be to put the module and straight away the end module, and keep on keying in between so that you will not forget.

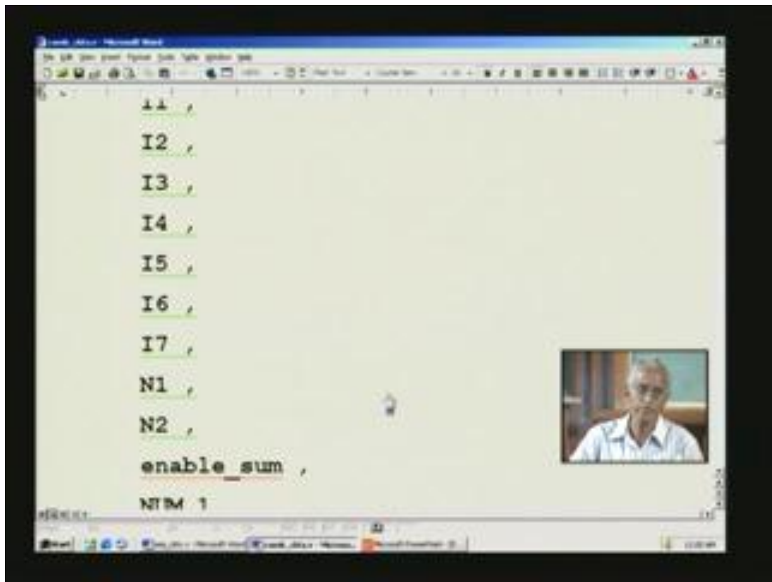
Once again, we have a pair of braces here. (Refer Slide Time: 00:10:40 min) There will be a start brace here and there will an end brace at the end. Towards the end, we will have to declare all the i/os here. Which order you mention them is left to you, and whatever is convenient to you can be done; it is a rather a free format in Verilog. A, B, C can be an output or 1 can be an input or output, or output and input; any combination that you prefer. It all depends upon your practice. Whatever start you have made, stick on to that so that it is readable for you, not only for yourself now and later on after years of usage, if you want to revise the design, but also for others who will be using your codes.

(Refer Slide Time: 09:56)

We have already seen here that we had to declare the design module; that is what is being done here. This is the module name that we gave and that is what we declare here. What appears here is that A, B and C. Note that every input is followed by a comma, just to indicate that this is one signal and another signal is this. Just a comma separates one signal from the other. Another thing - I wonder if I mentioned it to you earlier about why this style is adopted. I will explain this again. It depends upon the designer, and changes from designer to designer.

I prefer this style and the reason is that we can easily spot out where a particular signal has been used or not. By mistake, if you club all of them in a single line, it will be difficult to track. Of course, there are editors those who search, but this will be an easier thing. I will leave it your taste to handle it.

(Refer Slide Time: 12:40)



There is no restriction regarding that. All the inputs you have already seen in the course earlier. I may not go into the depths of the codes again, but I will do so during the simulation period. That will avoid unnecessary repetition. I may not explain today the actual code which we have already covered. These are all basically inputs here which we will have to define later on. Some of them are single bit and others are multi bits. These are all the outputs right up to F 11 and F 12.

Note that these i/os are separated by commas which you have already seen except the last. This also, I think I have mentioned earlier. It is quite a long list of i/os. There must be a comma here. The compiler will complain here. (Refer Slide Time: 00:13:44 min) In fact, the compiler will not really throw full light on that; it might start complaining for, whatever follows later on also. That also may be there, but here in this case, it may not.

What we had done is just a listing of the i/os, but we will have to identify the particular signal for inputs. That is what has been done here. For example, A B C are all inputs here. Just put a comment saying that we declare all the i/os here. Coming to this statement, you notice that N1 is an 8 bit number which we already covered. We will touch up on it when we go to the actual assign statements that will follow suit here, and MSB as I mentioned is 7. That is the first thing that should be put here. This is a pre-

requisite here (Refer Slide Time: 00:14:41 min). If you interchange them there will be a lot of conversions and will interpret as MSB, and you will have lots of problems.

(Refer Slide Time: 14:00)

```
    LESS
);

input        A ;  // Declare the Inputs and
                  // Outputs of the module.

input        B ;

input        C ;

input        I0 ;
```

(Refer Slide Time: 14:23)

```
input        I6 ;

input        I7 ;

input   [7:0]  N1 ;

// Mention the size of the input -

// [7] is the MSB, and [0] is the LSB.

input   [7:0]  N2 ;

input   enable_sum ;
```

Stick on to this, and in this as usual N1 is the input. The only difference is that for a 1 bit there is no point in putting 0. You can dispense with it, but that is what is implied here. Here, you put seven to 0 for the 8 bits and the same inputs continue for different

dimensions and different signals. This is the 9 bits here, and these are all 1 bit outputs here (Refer Slide Time: 00:15:25). Just as you declare an input, you can also declare an output and once again list. As I mentioned earlier, you do not have to first list all the inputs and then go to outputs; if you feel like, you can say 1 input 1 output or in any mix you want.

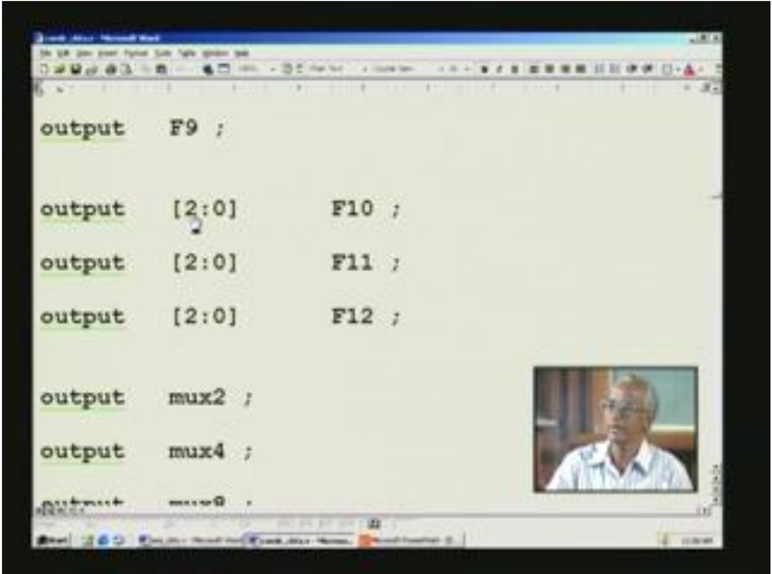
(Refer Slide Time: 15:21)

```
input   [7:0]  N2 ;
input   enable_sum ;
input   [7:0]  NUM_1 ;
input   [7:0]  NUM_2 ;
input   [8:0]  PRESET_VALUE ;

output   F1 ;
```

(Refer Slide Time: 15:25)

```
input   [8:0]  PRESET_VALUE ;

output   F1 ;
output   F2 ;
output   F3 ;
output   F4 ;
output   F5 ;
output   F6 ;
```

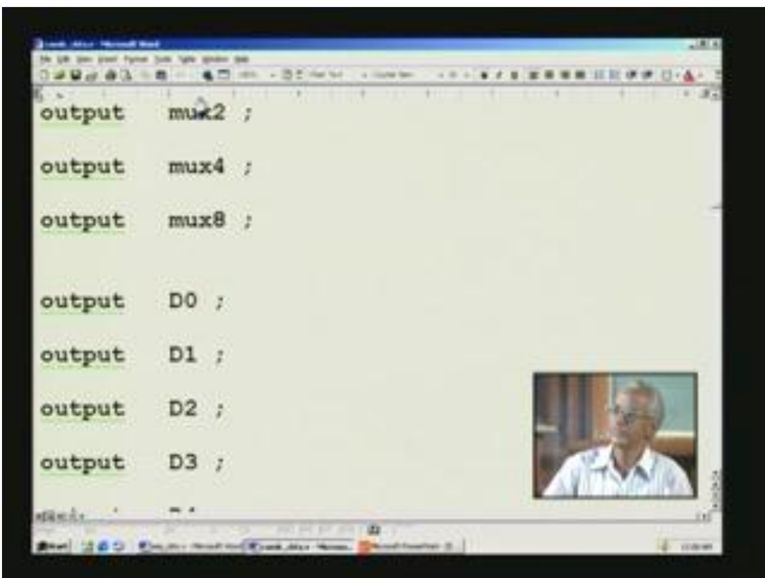
(Refer Slide Time: 15:45)



Once again, here you can see multi bit represented in a similar fashion.

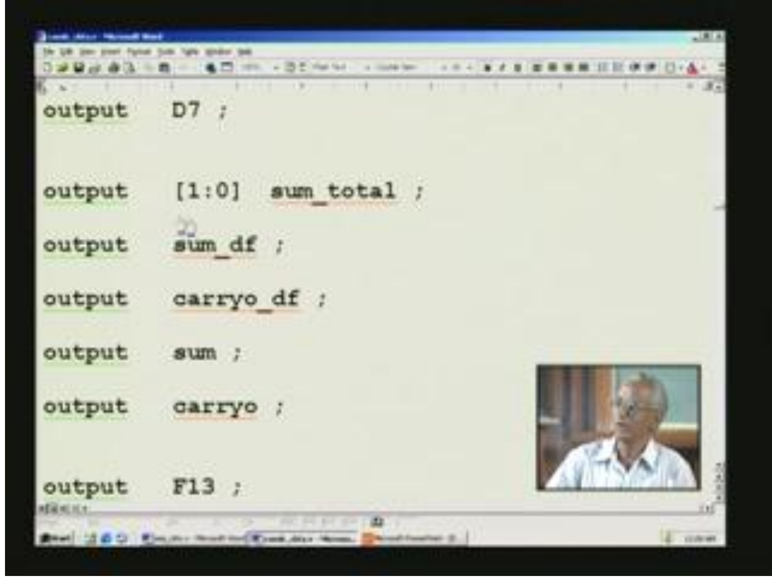
Conversation between student and professor – not audible (Refer Slide Time: 00:15:53 min). No.No. It can be a twisted jumble. Ya. It can. Semi-colon, actually separates one statement from another. You can put comma F11 and so on, but readability suffers.

(Refer Slide Time: 16:21)



Once again, we have had 2 input mux that outputs. You have a listing here - 2 input 4 input and 8 input mux, and once again they are all single bit. We also had a D mux that output is D 0 through D7. Once again, the output multi bit used in different places; for example, we had done a full adder; we have seen behavioral, structural etc.,; all those things are put here. The only difference between the coding explained earlier and here is that I might have used the very same nomenclature, but when you start putting it, it has a full-fledged code. You have to have different names; otherwise, the same signal will be overwritten. That is the reason I had made very slight changes. Just rename it; nothing more. Similarly, all outputs up to F18 and so on. Once again, it is precisely the same.

(Refer Slide Time: 16:23)



(Refer Slide Time: 17:18)

```
output    carryo_df ;
output    sum ;
output    carryo ;

output    F13 ;
output    F14 ;
output    F15 ;
output    F16 ;
```

(Refer Slide Time: 17:28)

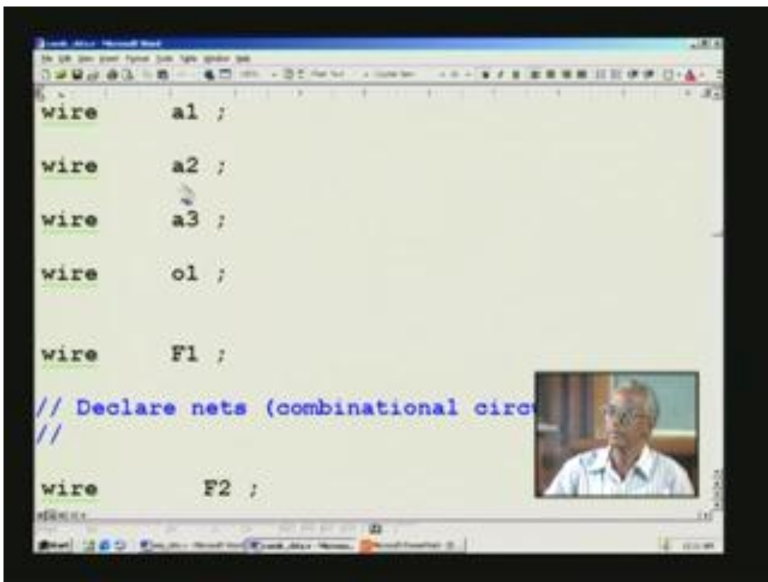
```
output    MORE ;
output    LESS ;

wire      [1:0]   sum_total ;
wire      sum_df ;
wire      carryo_df ;
wire      sum ;
wire      carryo ;
```

I will just go to the next. Here, we have used assigned statements earlier and assigned some outputs. So, whatever output appears in the assign statements, they will be wires. What is a wire? For example, you have a combinational circuit, the output of which can be declared as a wire. It is something like a physical wire that you have in your circuitry

which you had used earlier in schematic circuit diagrams. You refer to that as a wire or in some cases, as a net.

For example, a microprocessor bus signal or any signal that you want to have, you can treat that as a wire. It is a physical wire connection. That is why; they have retained this name wire. Once again, you see multi-bits represented in a similar fashion, followed by the actual signal name. These signals may be an output, straight away appearing at the output of your design module, or it can be intermediate outputs used in subsequent sequential circuits etc., and so is the case for other variables. This is an output. These are all intermediate outputs which we will see when we go to the code and that is what we have said here.

(Refer Slide Time: 18:44)



(Refer Slide Time: 18:56)

```
wire     F1 ;

// Declare nets (combinational circuit
//                                   outputs).

wire        F2 ;

// F1 thru' F9 are all single bit outputs.

wire        F3 ;

wire        F4 ;

wire        F5 ;
```

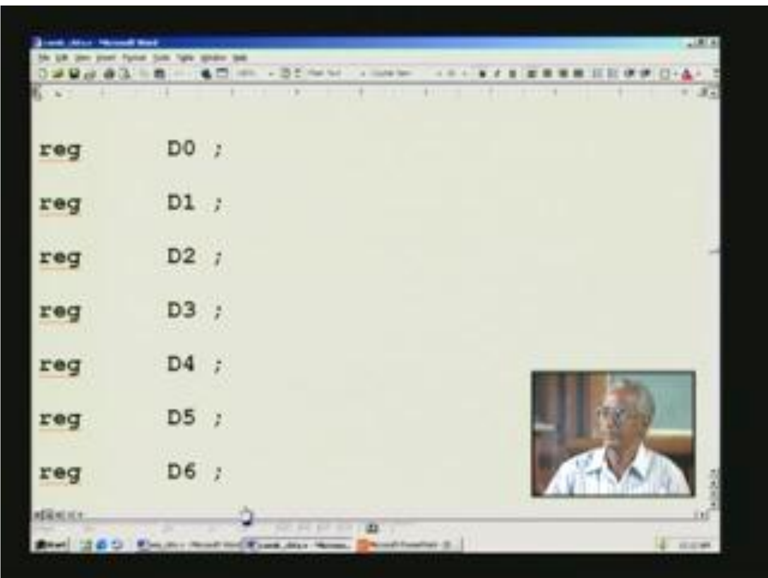
(Refer Slide Time: 19:05)

```
wire        F7 ;

wire        F8 ;


reg         F9 ;      // Declare registers.

reg  [2:0]  F10 ;     // F10 thru' F12 are all
                      // three bit outputs.

reg  [2:0]  F11 ;

reg  [2:0]  F12 ;
```

The declaration of net is declared as wire. Then comes always blocks: Whether it is a combinational or sequential. In sequential, you have a positive edge clock. so always at positive edge clock. in ordinary combinational circuit, you have it always when you select I0, I1, like that You list all inputs whenever there is a change, and only then that will be processed. In any always for that matter, you declare those outputs that you have

mentioned there in as a register. This is another easy rule to remember say for assign statements, you can say wire and for always block, whatever output, all are with reference to the outputs only and not the inputs. You can refer to it as a reg. In always block whatever signals that come in. Once again you see here for reg, multi bit through this.

(Refer Slide Time: 20:07)



For a single bit, earlier we have used D0 through D6 in an always block if you remember, and therefore they are declared as reg. This is a mandatory thing; you have to declare any signal that you use, either as a wire or reg, and even outputs would be normally a reg if it is coming from the sequential circuit. Once again, you see we did not use always block. We used assign statement for this. So, we use a wire here and we used an always block for mux. Therefore, it is a reg and these are all in always positive edges in sequential circuits. I think we have declared this as reg.

(Refer Slide Time: 20:34)

Once again, multi-bit here, we made a small design block which computes this sum and then reports whether a match has been found with a preset value and those outputs; this is a multi-bit, 9 bit output, and these are all single-bit.

(Refer Slide Time: 20:54)

(Refer Slide Time: 21:00)

```
reg     [8:0]    SUM ;
reg     MATCH ;
reg     MORE ;
reg     LESS ;

// Combinational circuits using
// statement.

assign F1 = A ;
```

Hence, this is the kind in the sequential block. I am not sure whether it was sequential; we will have a look when we come to that.

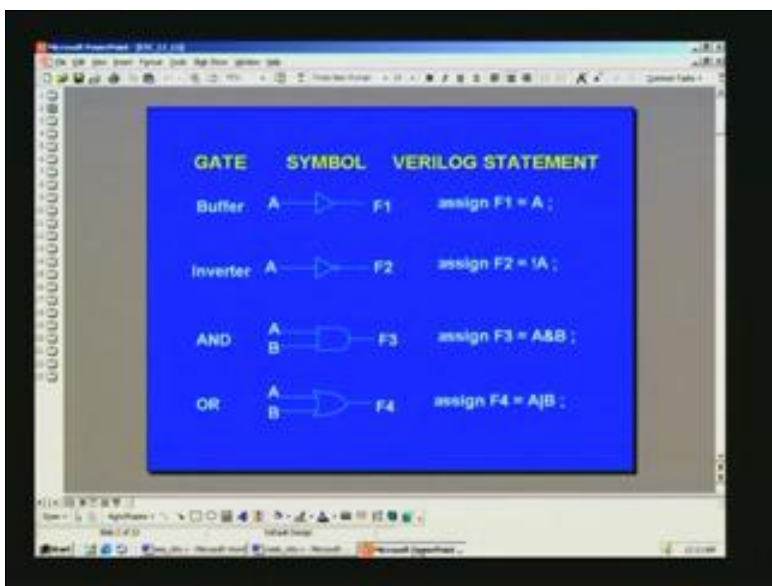
(Refer Slide Time: 21:30)

```
reg     LESS ;

// Combinational circuits using "assign"
// statement.

assign F1 = A ;        // Verilog code for
                       // buffer,

assign F2 = !A ;       // inverter,

assign F3 = A&B ;      // AND,

assign F4 = A|B ;      // OR,
```

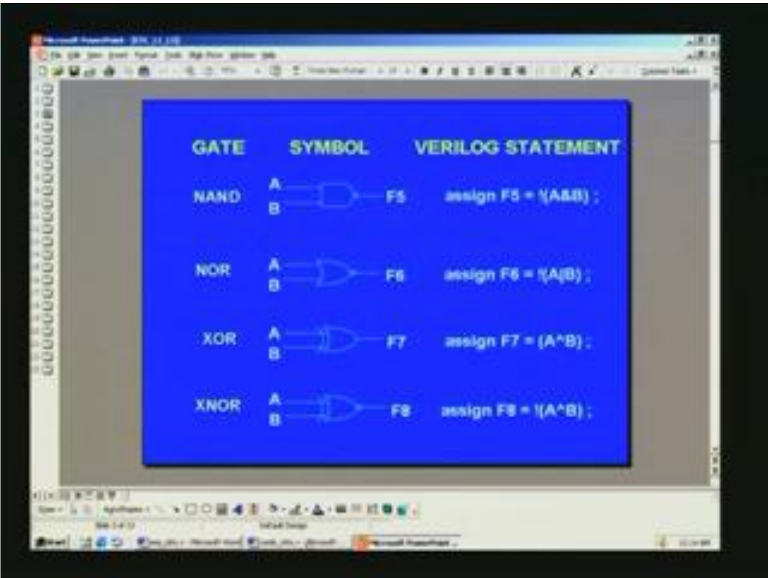
Combinational circuits using assign statement. To start with, we saw how to implement a buffer using an assign statement. Precisely that is what we have put here in the statements up to that. That is what is appearing here as it is. That is why we said before that we will

deal only with the core of the statements first so that you get a quick grasp of those fundas, and then go on to more details which we have already seen here, namely, identification of modules and how to put them together, then i/os and then declare a [.] (Refer Slide Time: 00:22:15 min) of wire and reg. So, you can put a meaningful comment here.

(Refer Slide Time: 21:37)



(Refer Slide Time: 21:48)



(Refer Slide Time: 22:34)

```
                      // buffer,
assign F2 = !A ;      // inverter,
assign F3 = A&B ;     // AND,
assign F4 = A|B ;     // OR,
assign F5 = !(A&B);   // NAND,
assign F6 = !(A|B);   // NOR,
assign F7 = (A^B);    // XOR, and
assign F8 = !(A^B);   // XNOR
```

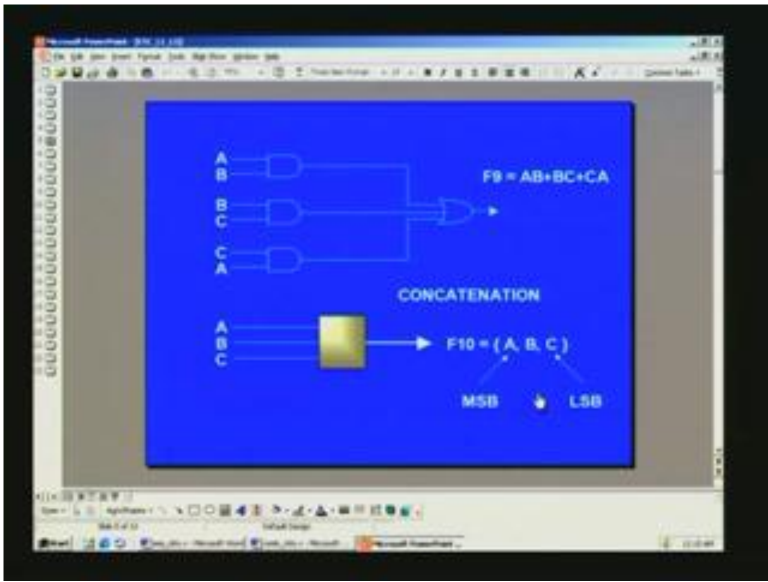
I will not go into details. Perhaps to just recollect, this is AND- IN operation; this is an OR operation. This is a NOT operation and you can have NAND etc., by the combination of this. That is what is listed here and for exclusive use, we can use this symbol.

(Refer Slide Time: 22:43)

```
// Combinational circuits using "always"
// statement.

always @ ( A or B or C or F10)

// F9 thru' F12 are computed only if there is
// change in any of the three inputs, A, B, C
// and F10.

  begin
```

We have also seen that using always block different combinations circuits. For example, we realize that this majority logic which is AB plus BC plus CA; this is precisely what we have already seen here in one of this.

(Refer Slide Time: 23:11)



This is a pictorial depiction. You have already seen this. The next example, was concatenation of three signals ABC. This will be the MSB which we have already seen.

These are all the inputs to this thing. What it does is, it puts them together in this order and you can regard that F10 as one signal which has 3 bits, A being the MSB.

(Refer Slide Time: 23:38)

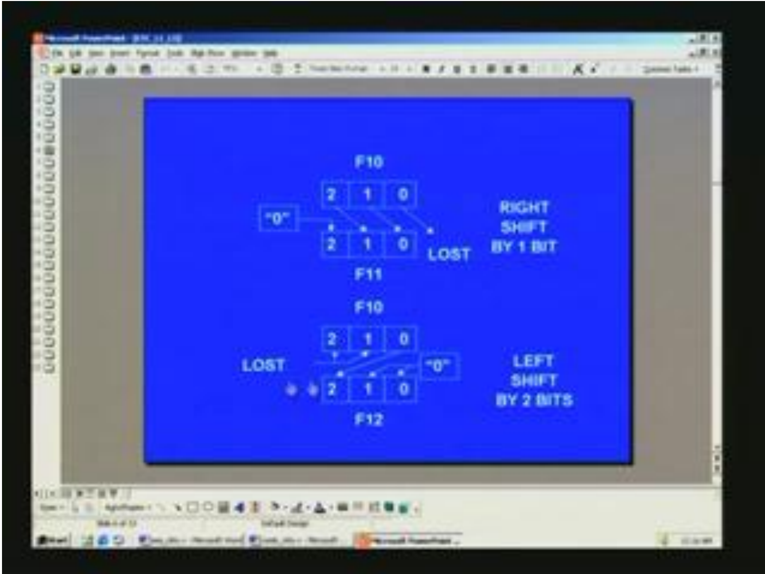
```
    F9  = (A&B) | (B&C) | (C&A)  ;

// Realize AB+BC+CA.

    F10 = {A, B, C} ;

// Concatenate A, B and C to get 3 bit result.

    F11 = F10 >> 1 ;
```

That is what we have put here in this statement, and you give once again a meaningful comment. Then, we saw shift registers shifted by 1 bit and 2 bits here.

(Refer Slide Time: 23:48)

```
// 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.
```

(Refer Slide Time: 23:57)



This picture depicted that you start with F10, and then you right shift by 1 bit. This is how it happens, and 0 is filled in the vacated bit position. If it is left shift, the other way happens and now in this case, two 0s are filled, because you are doing 2 bit shift here. What is shifted in these things is lost, unless you store them deliberately and that is what we have here.

(Refer Slide Time: 24:27)

```
// Verilog models for Multiplexers

// Two input MUX using "assign" statement.

assign mux2 = (A == 1) ? I1 : I0 ;

// mux2 = I1 if A = 1, otherwise mux2 = I0.

// Four input MUX using "assign" statement,
```

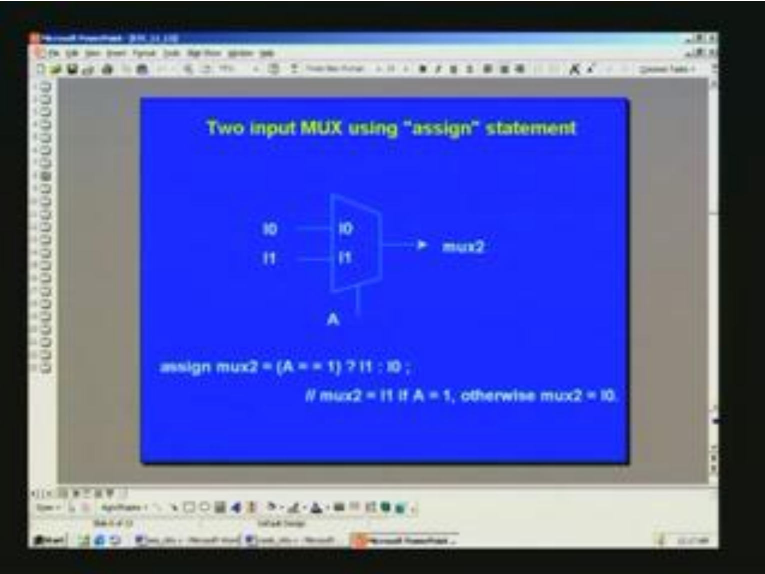
We then saw how to model a multiplexer. We can use an assign statement and this is a two input mux with A as a selector control. If A is 1, I 1 is routed to mux 2. Otherwise, I is not taken as mux 2 and mux 2 is the output, as such.

Similarly, we have seen a nesting of two such…(Refer Slide Time: 00:24:52 min). This is basically a mux is a question mark and you have a nesting of two mux here. In order to deal with four inputs, these also were trained and B and C are the select pins, B being the MSB and that is what you see here and in these two; that is, for the first 2 inputs mux and then followed by 4 input mux, it is pictorially depicted. We have also seen an 8 input mux, so 7 is the inputs mux, 8 is the output and selector pin ABC, MSB is A, and this is important.

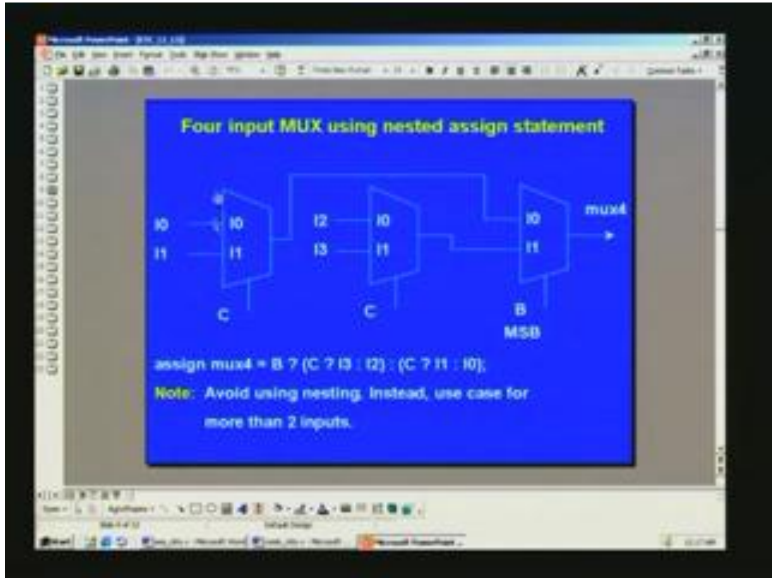
(Refer Slide Time: 24:50)

```
// mux2 = I1 if A = 1, otherwise mux2 = I0.

// Four input MUX using "assign" statement,
// nested.

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

// Avoid using nesting. Instead, use case for
// more than 2 inputs.
```

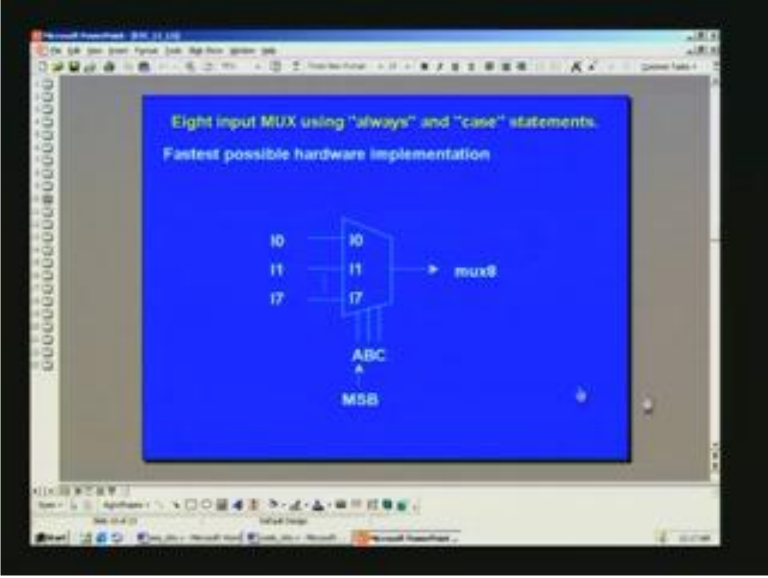
(Refer Slide Time: 25:10)



(Refer Slide Time: 25:12)



(Refer Slide Time: 25:18)



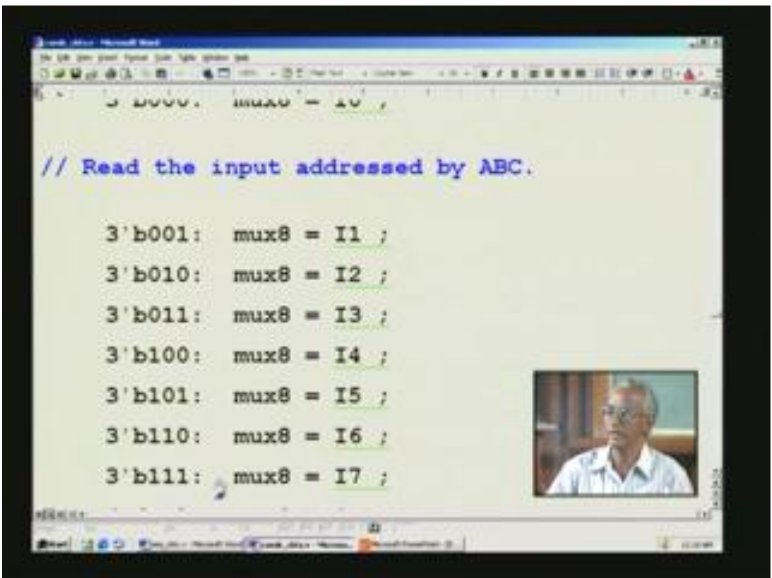
(Refer Slide Time: 25:43)

```
// Fastest possible hardware implementation.

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 ;
```

We have also mentioned that this results in the fastest possible hardware implementation and we have also said that always block has been used.

Whatever outputs that you have that must have been declared as reg earlier. You can go back and find out whether mux 8 has been declared as a reg; whereas, mux 2 and mux 4 where declared as a wire, because they do not appear in this Always block and we list all the inputs.

(Refer Slide Time: 26:40)



Whenever there is a change of input, only then this case statement will be processed. A case has concatenation of 3 numbers; what we have already seen earlier. If A B C is 0 0 0 respectively, then I0 is selected as output to the mux 8 signal. Otherwise, other signals are output.

We have seen in depth so the 1 is say 1 1 1 corresponds to I7 and you may encounter x, that is, do not care or tri-states say zee, and you have to account for such cases as well. That is why, we put a default here. (Refer Slide Time: 00:25:56 min) So it is a good practice to have a default and take it to a safe state. It might be according to your circuit design you may consider 0 as a safe state in some other condition, 1 or x or zee, depending upon your real need. You can just initialize to some safe condition, and once we have a case, we need to have an end case to complete the thing. After always 1 begin

an end is there. Notice that there is no semicolon either here or here. (Refer Slide Time: 00:27:27 min)

(Refer Slide Time: 27:38)

```
end

// Eight input DEMUX using "always" and "case"
   statements.

// This is a simple inverse of a MUX.

always @ (A or B or C or mux8)
```

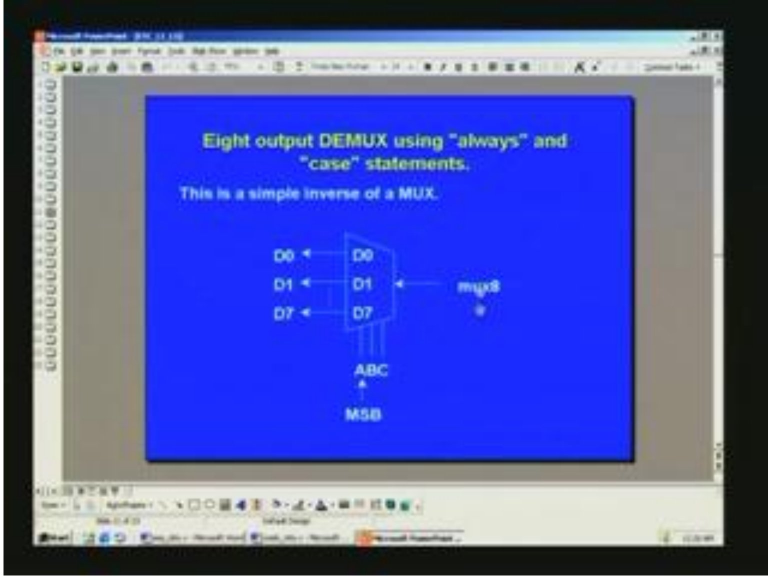
What we are doing here is that we have a missing 8 input mux. 8 outputs, I mean 1 output and that output we will feed to another D mux. It is a reverse process. So, see whether what we started with inputs for the mux appears at the output or not. That will be easy to check while doing the stimulation and that is why I fed it back. So, in this case, once again you use an always block and a case.

(Refer Slide Time: 28:01)

```
begin
  case ({A, B, C})
    3'b000:  begin D0 = mux8 ; D1 = 0; D2 = 0;
             D3 = 0; D4 = 0; D5 = 0; D6 = 0;
             D7 = 0; end

// Read the input into D1, etc., and clear
   other outputs.
    3'b001:  begin D1 = mux8 ; D0 = 0; D2 = 0;
```

These are all a case, exactly the same meaning as before, but the only difference is note that when this combination is met, mux 8 is output to D not and this pictorial shows this.

(Refer Slide Time: 28:15)



What we have done here, is, mux 8 was the output of earlier mux. Here, we feed it back. So, you expect a D0 through D7 precisely, what you have fed at the input here, and once again this has the same meaning. This is a de mux or you can even say decoder. Coming

to that, only this is applicable. So, what happens to the other outputs? You have to force them to 0. If you do not, you may land up in a problem later on, because it might have come from an earlier state with 1; so it latches on to that condition if you do not mention. This also, we have seen while dealing with RTL coding guidelines and similarly for other cases; all the eight cases are considered here.

(Refer Slide Time: 28:59)

```
// Read the input into D1, etc., and clear
   other outputs.

    3'b001:  begin D1 = mux8 ; D0 = 0; D2 = 0;
             D3 = 0; D4 = 0; D5 = 0; D6 = 0;
             D7 = 0; end

    3'b010:  begin D2 = mux8 ; D0 = 0; D1 = 0;
             D3 = 0; D4 = 0; D5 = 0; D6 = 0;
             D7 = 0; end
```

In each case, you see that mux 8 is assigned to the corresponding thing. This 4 corresponds to this 4, although it need not necessarily be.

(Refer Slide Time: 29:04)

```
          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
```

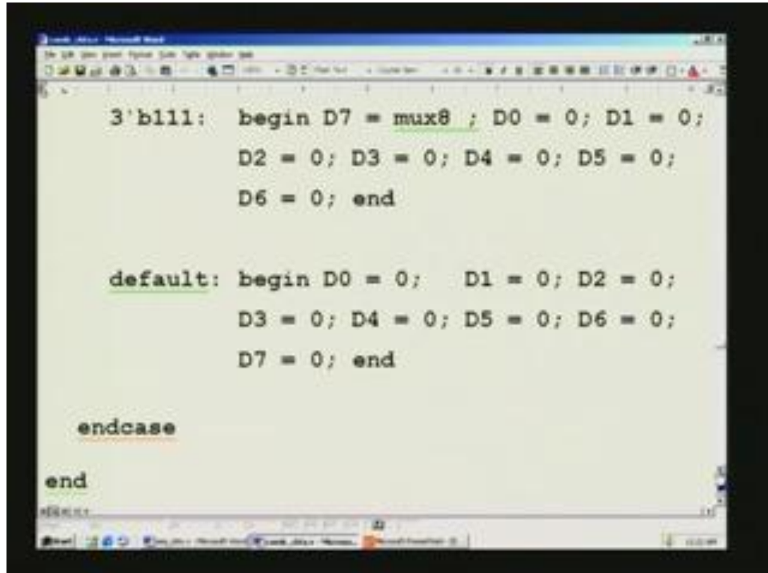
(Refer Slide Time: 29:16)

```
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;
```

Another point I might mention is that you have put everything in a sequential order say binary sequence. There is no need for you to do so. Also, it may be painful to write so many bits. If you have 64 inputs, we will have to write many and more the number, more you have to write.

Better thing would be to use a decimal. Here, you can just replace this B with D, but even in that case, this will mean only number of bits. So, you have to take the trouble of computing first as to how many bits it will run into maximum and then put this. So, this tick is a prerequisite here and this colon is also prerequisite here. (Refer Slide Time:00:30:03 min) We have once again covered a default here, and have taken all of them to 0 state. Once again, the end case end, matching end and then we have seen a full adder.

(Refer Slide Time: 30:03)



(Refer Slide Time: 30:20)

```
// Realization of a full adder
// Behavioral level realization
assign sum_total  = (A + B) + C ;
// Realize sum.
// Data flow level realization
assign sum_df     = (A^B) ^C ;
// Realize sum.
assign carryo_df  = (A&B) | (B&C) | (C&
```

We have seen three types of full adders; so, 1 is a behavioral level; this is the best, because you need to write just plain addition symbol. All of them are 1 bit and you add A and B; that is the bracket. I have explained earlier also that you should gate the synthesis tool, because it cannot make a decision all by itself. What you intend doing, you have to tell them, tell the synthesis tool and then do this in parallel with this; it means, to say that here it is redundant. If you had 1 more here - C plus D, you can put another parenthesis then C plus D and A plus B will be done simultaneously.

We have also seen earlier while in RTL coding, precisely that example. There is the same end result that can be achieved in two other ways. Another way is what is called the data flow realization, in which case you straight away deal with the actual gate, but in a concise manner, as this is nothing, but exclusive of 3 inputs. So, we do this here once again. To guide the synthesis tool, we put this. (Refer Slide Time: 00:31:30 min) That is what the sum was, and then here a carryout was nothing, but a majority logic which we have already realized earlier.

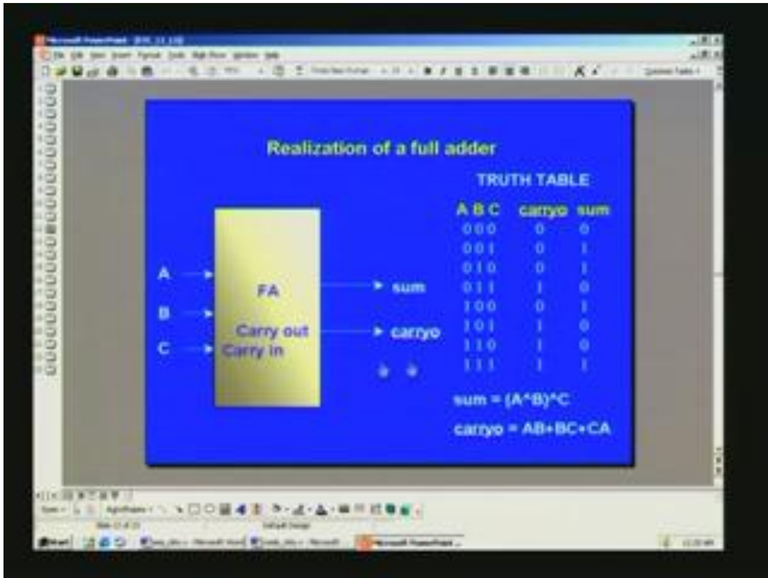
(Refer Slide Time: 31: 39)

```
// Realize sum.

// Data flow level realization

assign sum_df      = (A^B) ^C ;

// Realize sum.

assign carryo_df  = (A&B)|(B&C)|(C&A)  ;

// Realize carry out, AB+BC+CA.

// Structural level realization
```

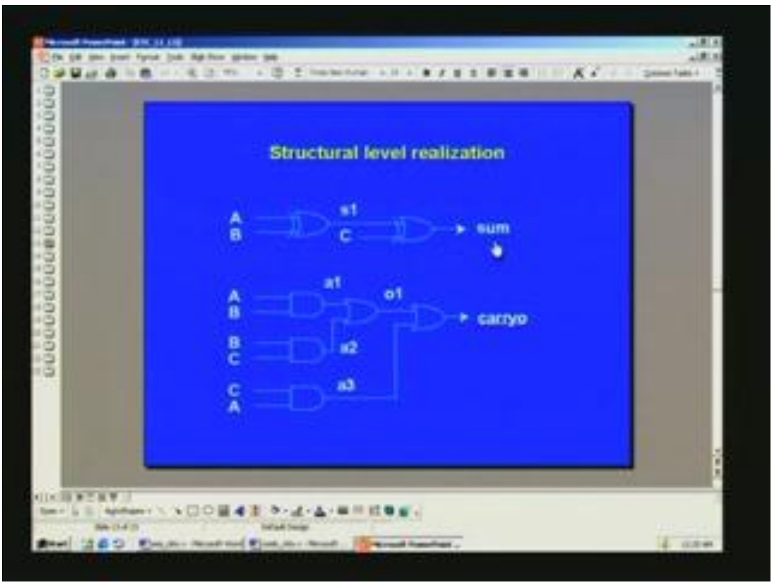
Notice that, as an example, in order to show all these features, I have interconnected them using the very same inputs outputs as far as possible. So, the only thing is that outputs have been changed as I mentioned before, because two results will be contradictory. So, it will take the latest and it will overwrite the previous result. That is why the name has been changed in some cases.

In fact, I have used the very same name. Now, I have marked a difference here; I had just extended a carry out underscore df. It is used to connected data flow, because that is what we are dealing here and data flow is very much [.] (Refer Slide Time: 00:32:27 min) to your RTL coding, although RTL coding has graduated from this level to even behavioral level. Another way of implementation is using the gate level logic. You can use a primitive gate and this is how you use and you have seen this here. So, full adder - I do not have to say some carry two table and this is the final thing.

(Refer Slide Time: 32:53)



(Refer Slide Time: 33:01)



In order to implement using gate primitives, whatever you see as a circuits you can straight away implement. This also has been explained here. So, that is what we are doing here.

(Refer Slide Time: 33:14)

```
// Structural level realization

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.
```

Note that this is an output and these two are inputs for an exclusive, or if you want you can say 1, which will instantiate. You can use that and there is no need for you to put separately. Here, I think you can put a comma and add 1 more, 2, like that and this is what is realized here exclusive of three inputs, and then AB BC CA are the majority logic for the output here.

(Refer Slide Time: 33:41)

```
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.
```

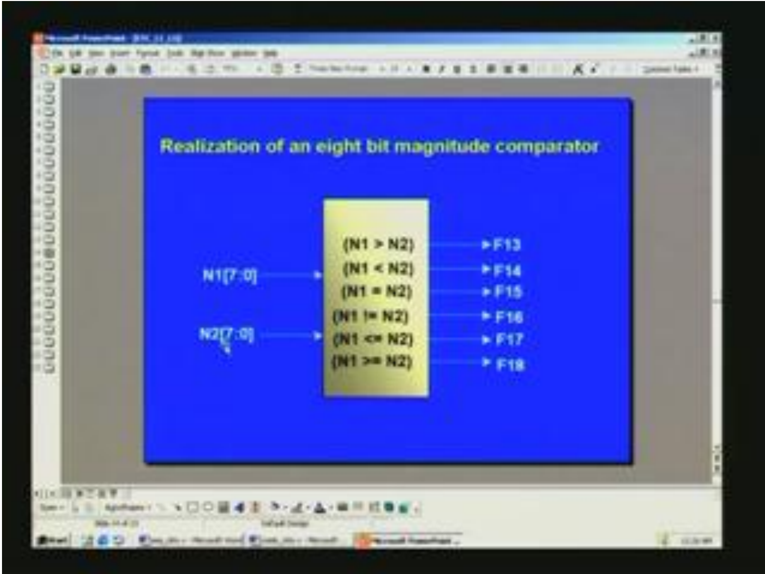
These are all the intermediate signals s1 a1 a2 a3 o1 etc., here. These have exactly the same meaning as exclusive or so always, output followed by the inputs and operation for 2 input and operation 2 inputs or operation and that is what you had here. See, these are all s1 a1 a2 a3 o1 that is precisely the intermediate outputs that we have used.

(Refer Slide Time: 34:12)

```
// 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.
```

Next, we went to implementing a magnitude comparator and this always block and the two numbers here are compared.

(Refer Slide Time: 34: 26)



Accordingly the outputs are set if it is greater this would be equal to this and so on.

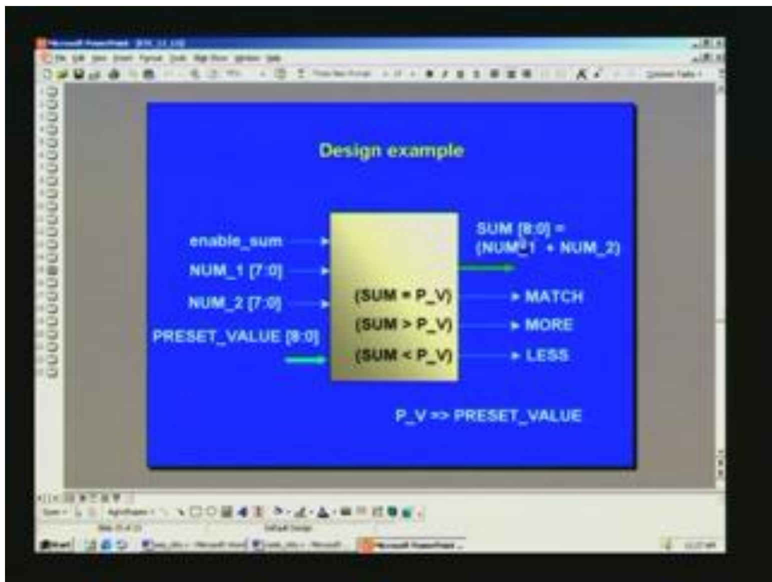
(Refer Slide Time: 34: 56)

```
    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);
```

This code is given over here. It is plain 1 to 1 correspondence. This is the 1. Just when it is greater you just assign. Braces are always easy to read, so you make it a habit to use braces freely.

Again not equal to, and, less than or equal to and in always block, you use that one. You should distinguish the difference between that. Here it has a different meaning - it means precisely less or equal to. In an always block, I mean non-blocking statement - it is called that is near equal to.

(Refer Slide Time: 35:23)



(Refer Slide Time: 35:44)

```
//                A Design example

always @ ( enable_sum or NUM_1 or NUM_2 or
                             PRESET_VALUE )

  begin

    SUM[8:0] = enable_sum ?
        ( NUM_1[7:0] + NUM_2[7:0]) : 9'd0 ;

// Compute sum if enabled, otherwise output 0.

    MATCH    = enable_sum ? (SUM ==
```

We also consider 1 design example. This involved computation of a sum between two numbers- 8 bit numbers only when the enabled sum is energized and compared with a preset value and if the final sum is equal to this, you output a match signal, otherwise, more or less that is how it is here.

Precisely, the same thing is being done here. It is a mux again which you have already seen and the summation is assigned to sum or 0 is assigned to the sum depending upon the enabled sum. If it is 1 that is active or not if it not active we simply make it 0, so as not to distinguish that no action has been taken there.

(Refer Slide Time: 36: 11)

```
( NUM_1[7:0] + NUM_2[7:0]) : 9'd0 ;

// Compute sum if enabled, otherwise output 0.

MATCH      = enable_sum ? (SUM ==
                        PRESET_VALUE) : 1'b0 ;

// Set output if SUM is equal to PRESET_VALUE,
// only if enabled.

MORE       = enable_sum ? (SUM >
```

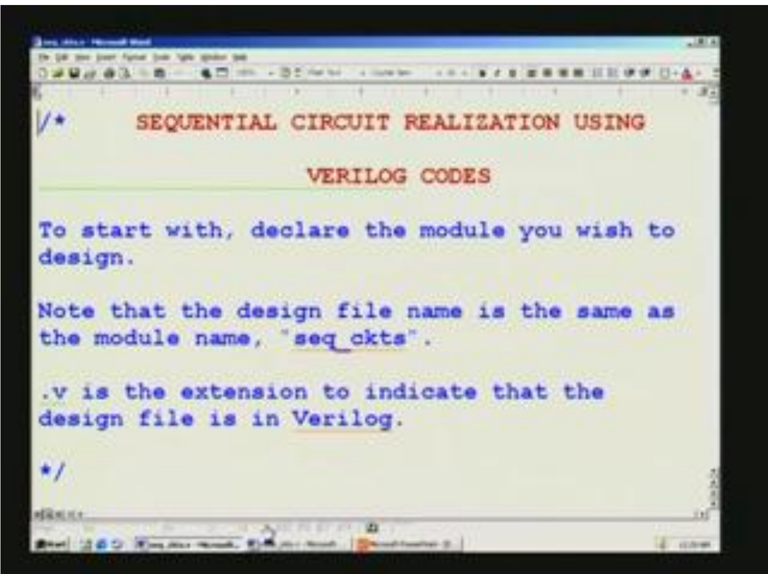
Similarly, for MATCH it is precisely the same thing. Just use the sum equal to preset value in the select pin of the mux, and assign in a similar fashion. So is the case for more and then less and then end.

(Refer Slide Time: 36:20)

```
LESS       = enable_sum ? (SUM <
                        PRESET_VALUE) : 1'b0 ;

// Set output if SUM is less than
// PRESET_VALUE, only if enabled.

  end
endmodule
// This indicates the end of the design.
```

We have come to the end of our combinational circuit. So, we finally do not forget to put the end module here. This is the indication that we have ended here.

(Refer Slide Time: 36:43)



Now, we will go on to the sequential circuits and exactly repeat the same thing probably little faster, because most of them are going to be a repetition and once again, we see that we had to give a name meaningful name here, we are going to do only sequential circuits. So, we will use this one and the file that you have for this module is dot v.

(Refer Slide Time: 37:08)

```
*/

`define  S0   3'd0   //Defines the state of
                     // the controller.

`define  S1   3'd1

`define  S2   3'd2

`define  S3   3'd3

module seq_ckts( // Declare the design module.
```

That is why it comes right on the top here and dot v for the Verilog execution. There is another new thing which we have not seen before. There is a tick here, followed by a define. Suppose you have a variable, you can define those variables. You have also another counterpart called another parameter.

We will not deal with it right now, but probably will see it later on. You say define and here it indicates the state machine. Finally, it will be seeing state machine model and in that we have s0 to s3 for state. We can list all this here. While writing the code, we can deal straight away with s0 and not the actual number. It is a compulsion to use a number at every point. So, it is always easy to use a symbol like this. This is 0 and this is only for our readability. This decimal is what will ultimately reside. It will be in binary fashion, but while we deal with it, it is easier for us to resort to decimal.

Once again 3 bits binary, because maximum we can have is 7. You can correct it to 2, because if it is going to be 3 2, I will check it up later on, but you can straight away correct, because I have listed only this, in which case you might have to use a default. If you use this one, ultimately a case statement you will be using for machines. There you have to take care using a default.

Once again sequential circuit is the module name. This is the keyword as we have seen and declared all the i/os. We list all the i/os. As I mentioned now, output is coming very close to this. This time I have put whatever function that we have. Those functions I have put, instead of listing all the inputs and wondering to which part of the code it belongs to. It may be a good practice; if you like this, you can adapt this. I have shown both and once again a b c they are all different i/os.

(Refer Slide Time: 39:11)

```
module seq_ckts( // Declare the design module.
        D ,      // List I/O s.
        Q ,
        Q_n ,
        A ,      // I/O s can be written in any
                    order.
        B ,
        C ,
        clk ,
```

(Refer Slide Time: 39: 49)

```
        reset_n,
        hold,
        shift ,
        in1 ,
        in2 ,
        trigger , // Monoshot trigger input.

        pixelout_valid , // Register output.
        cnt_reg ,
        delay_out ,      // Monoshot trigger
                            output.
```

There is a clock reset and in addition to that we have a hold pin also that is similar to a microprocessor hold signal. There are other i/os. We have shift register; so we use shift in 1 and so on. A mono-shot trigger input is required, and for that mono-shot and the corresponding outputs, one internal counter is also required. That is what we have put here.

(Refer Slide Time: 40: 13)

The final output is what is called the delay output and we have already seen this in detail. That is why I am going little fast here. Anyway, we will revisit when we do a simulation and this precisely why i/o listing for different ones.

(Refer Slide Time: 40: 28)

We have also done a parallel to serial converter. So, this i/os are pertaining to this. From here to this, up to end of conversion, which part it belongs to- you can probably comment

it like this, or if you require, you can put more comments. For a state machine, we had sum to indicate every state. In some, we had put a lamp there as output, zee not, through zee three, and that is also listed here. We had finally a pattern detector. A pattern sequence detector and its i/os are here. Both of them are 1 bit. Finally, bracket completed with a semi colon. Usually, people forget; especially, beginners either forget this or this or no comma here or a comma there and so on.

(Refer Slide Time: 40: 48)

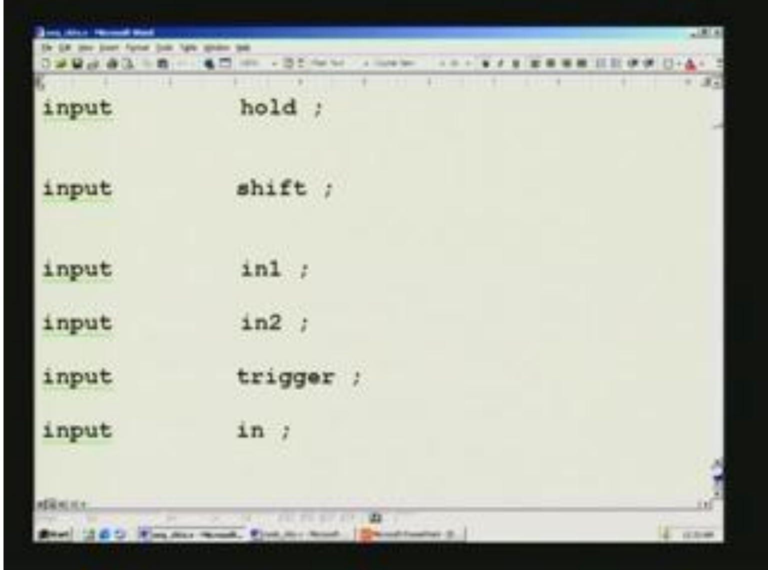
```
        data_out ,      // outputs of Parallel
        data_valid ,
        eoc ,           // to serial converter.
        Z0 ,
        Z1 ,
        Z2 ,
        Z3 ,
        in ,           // Pattern detector I/O,
        out            // 1 bit each.
);
```

(Refer Slide Time: 41: 23)

```
        out           // 1 bit each.
);

input         A ;   //  Declare the Inputs and
                        Outputs of the module.

input         B ;

input         C ;

input         D ;

input         clk ;
```

Once again, you identify what the inputs are and as you can see, I am not going to details.

(Refer Slide Time: 41:33)

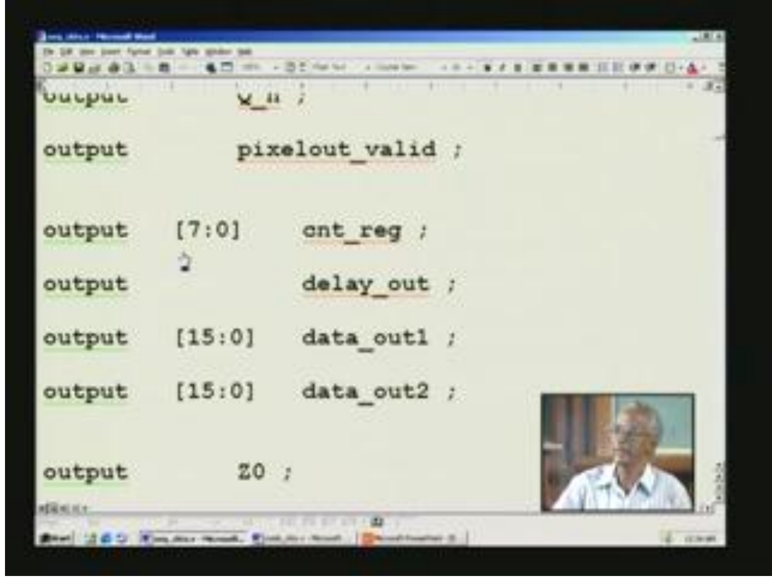


(Refer Slide Time: 40: 15)

```
input       trigger ;
input       in ;

output      Q ;
output      Q_n ;
output      pixelout_valid ;

output  [7:0]   cnt_reg ;
output          delay_out ;
```

Once again, multi precision outputs and single precision output.

(Refer Slide Time: 41: 39)



(Refer Slide Time: 41:52)

Once again, here are multi precision and single.

(Refer Slide Time: 41:57)

You see that input has been brought again and here write, input output. We can mix it any fashion that we like.

(Refer Slide Time: 42:04)

```
output     eoc ;    // serial converter.

reg        Q ;

reg        Q_n ;

wire       set_pixout ;

// Declare nets(combinational circuit
// outputs).

wire       reset_pixout ;
```

After that, you had to identify reg which we have already seen here.

(Refer Slide Time: 42:13)

```
// Declare nets(combinational circuit
// outputs).

wire       reset_pixout ;

reg        pixeloutp_valid ;

// Declare as registers

reg        pixelout_valid ;

wire       res_cnt ;
```

You list wire and reg and this we have already explained.

(Refer Slide Time: 42:20)

```
reg          pixeloutp_valid ;
// Declare as registers
reg          pixelout_valid ;

wire         res_cnt ;
wire         adv_cnt ;
wire   [7:0]  cnt_next ;

reg    [7:0]  cnt_reg ;
```

Wire is an assign output. We can use wire and always block outputs. We can use reg and that is precisely what we have.

(Refer Slide Time: 42:33)

```
reg          data_out ;
reg          data_valid ;
reg          eoc ;
reg    [15:0] sr ;
reg    [4:0]  cnt_ps_reg ;

// Realization of a D flip-flop with a reset
// control using "always" block.
```

Finally, it ends up in the pattern detector sequence detector i/os here.

(Refer Slide Time: 42:43)

```
reg        Z3 ;
reg        out ;
reg   [1:0]  psd_state ;
reg        data_out ;
reg        data_valid ;
reg        eoc ;
reg   [15:0] sr ;
```

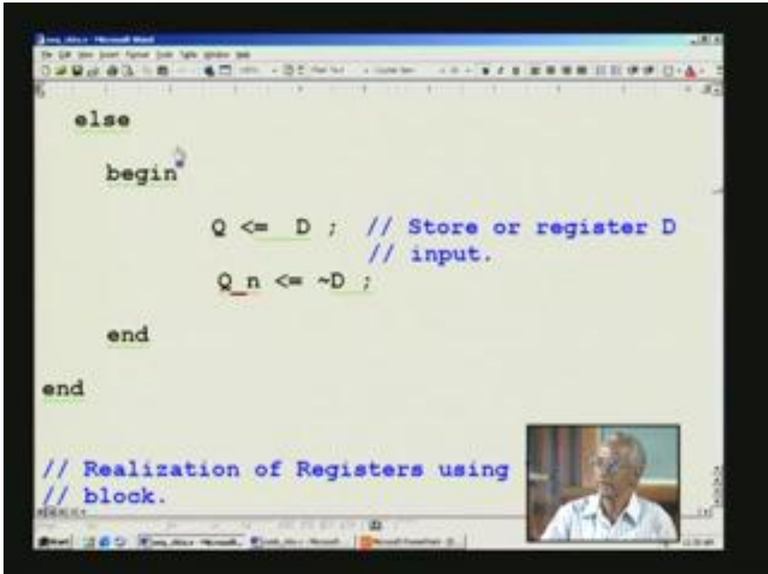
(Refer Slide Time: 43:01)

```
always @ (posedge clk or negedge reset_n)
begin
    if (reset_n == 1'b0)
    begin
        Q  <= 0 ;   // Clear register when
                    // system is reset.
        Q_n <= 1 ;
    end
```

Let us start the code for each of this. You see here, of course, it is nothing but a d flip-flop which was put on the board. If you remember and you have an always block and with positive edge of clock or negative edge of reset, take the action. You can reset the flip-flops. Whenever reset is applied and this is done right at the power on stage, there are

two outputs q and q underscore n, which you have you know as a d flip-flop and you can initialize this here, q to 0 and 1 here.

(Refer Slide Time: 43:36)

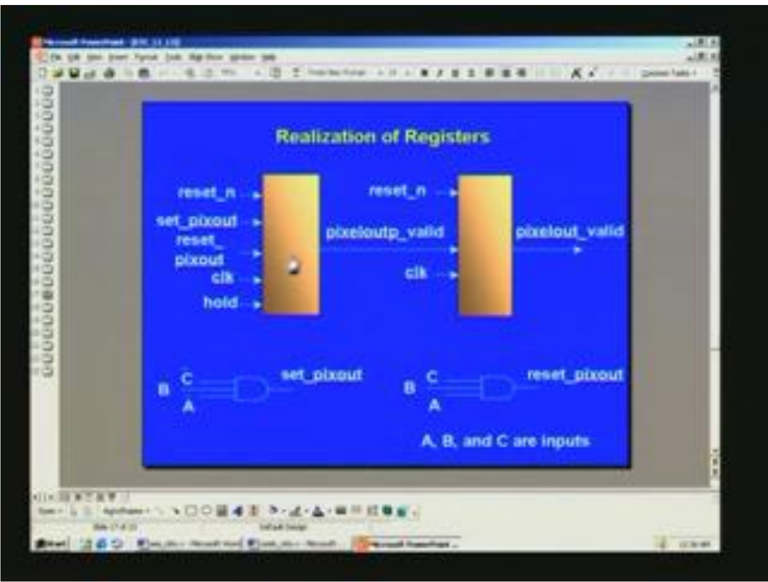


The course of normal working of d flip-flop, you just assign whatever is in d flip-flop; d input to q and inverted d to q underscore n, but unfortunately this creates two flip-flops. When you look into the synthesis tool later on, to decrease the count, you can put this as an assign statement outside and have the same result. Only an inverter will be used and that may be better approach.

(Refer Slide Time: 44:07)

```
assign set_pixout   = (A == 1'b0) && (B ==
1'b1) && (C == 1'b0);

// Pre-compute (not A) and (not  B) and (C).

assign reset_pixout = (A == 1'b1) && (B ==
1'b1) && (C == 1'b1)  ;

// Pre-check ABC status.
```

(Refer Slide Time: 44:14)



Here, we have two registers. We realize in this fashion that when you have just a plain register which is nothing but a collection of flip-flops, say d flip-flop, and depending upon on how much bit you want, there are reset set controls and block. Hold for this and whatever it stores. Here, we want to use it only with the arrival of the next clock edge. In other words we want to delay one particular signal in this case is what is called a pixel

out valid with a p standing for the previous value. Actually, this is the real value that we are interested in, but we want it at the next clock pulse. So, what we do is we put another register and then register it. That is what you have here. We have already seen this before; whatever set and reset conditions are all applied here. The corresponding code is what is shown here.

(Refer Slide Time: 45:30)

```
assign set_pixout   = (A == 1'b0) && (B ==
1'b1) && (C == 1'b0);

// Pre-compute (not A) and (not  B) and (C).

assign reset_pixout = (A == 1'b1) && (B ==
1'b1) && (C == 1'b1) ;

// Pre-check ABC status.
```

What you see here is set pixel out and reset pixel out for A B C and A bar B C bar here and always block to realize the first register is what you have here and that is with p pixel out p valid. The same thing, if there is a hold like a microprocessor signal, it will just write back itself.

(Refer Slide Time: 45:36)

```
always @ (posedge clk or negedge reset_n)
begin
   if (reset_n == 1'b0)
               pixeloutp_valid <= 1'b0 ;
// Clear register when system is reset.

  else if (hold == 1'b1)
               pixeloutp_valid <=
                              pixeloutp_valid ;
```

That means to say it does nothing but just idles. It does not do anything. So, long as the hold is asserted, it keeps on going in this. So, to say block look in here. That is how it preserves at the time of hold.

(Refer Slide Time: 45:51)

```
  else if (hold == 1'b1)
               pixeloutp_valid <=
                              pixeloutp_valid ;
// Retain the value if the system is in hold.

 else if (set_pixout == 1'b1)
               pixeloutp_valid <= 1'b1 ;
// Set or reset when the conditions
// are satisfied.

  else if (reset_pixout == 1'b1)
```

When hold is removed, it continues from where it left off. It does not start from square 1 again. That is the beauty of using a hold. This we have seen in if statement and else if statements, to realize, when such and such condition is met do this; otherwise do that.

(Refer Slide Time: 46:33)

```
// Set or reset when the conditions
// are satisfied.

  else if (reset_pixout == 1'b1)
              pixeloutp_valid <= 1'b0;
  else
              pixeloutp_valid <=
                         pixeloutp_valid ;

// Otherwise, don't disturb.

end
```

So, the otherwise is this final otherwise is with an else that will mark the end of this entire always block. With an if else if notation, you see that once again it is v- saving back the same thing, because we do not want to lose its contents.

(Refer Slide Time: 46: 50)

```
end

always @ (posedge clk or negedge reset_n)
begin
   if (reset_n == 1'b0)
          pixelout_valid <= 1'b0 ;
// Clear register when system is reset.

   else if (hold == 1'b1)
```

The next register takes this as the input and precisely reset etc., is same in this next register here.

(Refer Slide Time: 46: 57)

```
   else if (hold == 1'b1)
           pixelout_valid <= pixelout_valid ;
// Retain the value if the system is in hold.

   else
          pixelout_valid <= pixeloutp_valid ;
// Assign previous (clk)value

end
```

The previous content is assigned here, so that means to say, in the earlier clock this might have matured. It was one. Let us say that one cannot be assigned right at the same clock. So, what we do is, we assign it only at the next clock. That is how you get a one clock

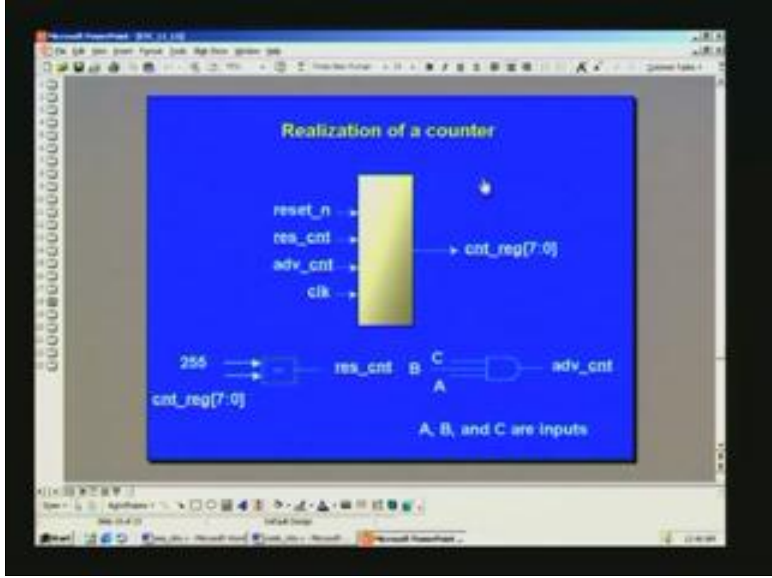
delay. If you want more clock delay, you can run a counter when a particular condition is satisfied and then when set value is reached at the counter, then you can take the action. So, you can delay as much as you want.

(Refer Slide Time: 47: 38)

```
// Realization of a counter using "always"
// block.

assign res_cnt = (cnt_reg  == 255) ;

// Condition for resetting the counter.

assign adv_cnt =(A == 1'b1)&(B == 1)&(C == 1);

// Condition for Pre-incrementing the counter.

assign cnt next = cnt reg + 1 ;
```

This is a counter realization.

(Refer Slide Time: 47:48)



In counter, we have declared it as counter reg; so, it is only to facilitate. It is a register - all flip-flops are basically registers and it is eight bit MSB once again here. If you want to advance the counter, then you base it on ABC status. If all of them are 1, then only advance here. That is what is applied here.

If you want to reset when the counter reaches say 255 in this case, it may sound a redundant, because next value will be automatically 0. It will wrap around being eight bits, but any other number you want here you can also use. That is why it has been put in a general fashion like this. There must be a clock and reset also, because all flip-flops normally need a reset power on reset.

(Refer Slide Time: 49:01)

```
assign res_cnt = (cnt_reg  == 255) ;
// Condition for resetting the counter.

assign adv_cnt =(A == 1'b1)&(B == 1)&(C == 1);
// Condition for Pre-incrementing the counter.

assign cnt_next = cnt_reg + 1 ;
// Pre-increment the counter.
```

This is the code for the counter. If you had already seen the advance counter, the counter is actually pre-incremented here by using assign statement in order to save computation time and thereby speed up the process.

(Refer Slide Time: 49:14)

```
assign cnt_next = cnt_reg + 1 ;

// Pre-increment the counter.

always @ (posedge clk or negedge reset_n)
begin
   if (reset_n == 1'b0)
            cnt_reg <= 8'd0 ;

// Initialize when the system is reset.
```

Here, as usual, with a reset you reset the register.

(Refer Slide Time: 49: 22)

```
            cnt_reg <= 8'd0 ;

// Initialize when the system is reset.

   else if (res_cnt == 1'b1)
// Reset if terminal count is reached.

            cnt_reg <= 8'd0 ;
   else if (adv_cnt == 1'b1)
            cnt_reg <= cnt_next ;
```

If not with a reset condition met, this will be met when you want to physically reset with an external reset input or with a count value which we have already seen here.

This reset condition happens to be only with counter equal to 255. Only then you reset here. That is what is covered here, and if you want to increment the count counter, then you have to see that ABC is 1. So that will be 1, and then this will be asserted here. That counter which was pre-incremented earlier will be assigned only here.

You may think that it is not going beyond the time; it would not, because whatever is pre-computed here, again depends upon this register only. So, it would not keep on incrementing, because it is assigned in an assign statement which is a combination. It depends upon the counter reg. So, this will change only with the arrival of the positive edge of the clock. So, you can be rest assured that this also will change only with subsequent to the arrival of the positive edge clock.

(Refer Slide Time: 50:43)

```
// Realization of a Non-retriggerable Monoshot
// delay using a counter.

// This produces 255 clock cycles delay for
// the preset value,

// cntd_reg = 255.

// For longer delays, change the cntd_reg
   width, and its preset value.

assign res cntd = (reset n == 1'b0) ||
```

We have one retriggerable and non-retriggerable mono-shot and earlier it was mentioned that we needed 256 clock cycles for delay. I have subsequently amended it, because it is easy to keep track. That set value is what you really get. So, I have amended that one. Although the previous code will work for two digits, that has been amended and that amended statement will appear in this case.

(Refer Slide Time: 51:15)

```
assign res_cntd = (reset_n == 1'b0) ||
(cntd_reg  == 255) ;

// Condition for resetting the counter.

assign run_delay = (triggerp == 0)&&(trigger
                                     == 1) ;

// Detect the positive edge of trigger.

assign cntd_next = cntd_reg + 1 ;
```

I will tell you, in a minute you can reset based on this condition; when reset is applied as well as a counter here, and this run delay is to detect 0 1 raising edge of a trigger p which is the input to the mono here.

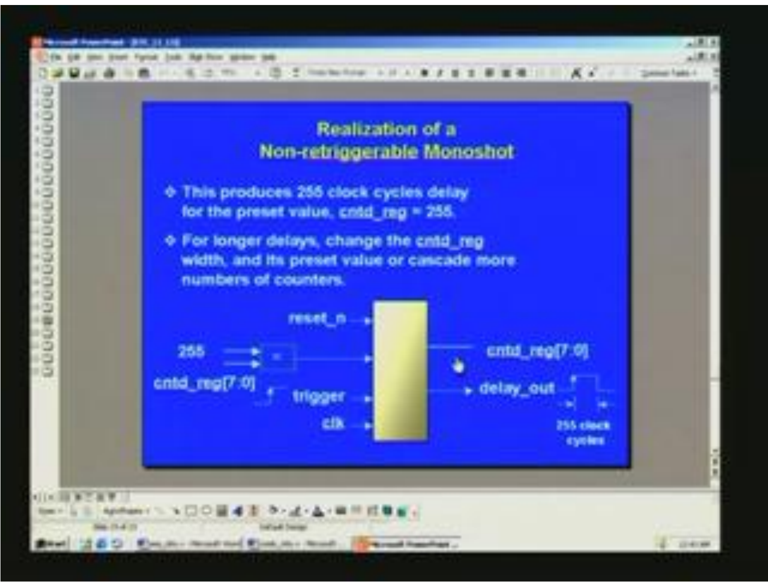
(Refer Slide Time: 51:15)

```
// Condition for resetting the counter.

assign run_delay = (triggerp == 0)&&(trigger
                                     == 1) ;

// Detect the positive edge of trigger.

assign cntd_next = cntd_reg + 1 ;

// Pre-increment the counter.

always @ (posedge clk or posedge reset_cntd)
```

(Refer Slide Time: 52:35)

```
// Pre-increment the counter.

always @ (posedge clk or posedge reset_cntd)

// Because of posedge res_cntd, delay_out
// will be 255 clock cycles instead of 256.

begin
   if (res_cntd == 1)

// Initialize when the system is reset
```

(Refer Slide Time: 51:37)



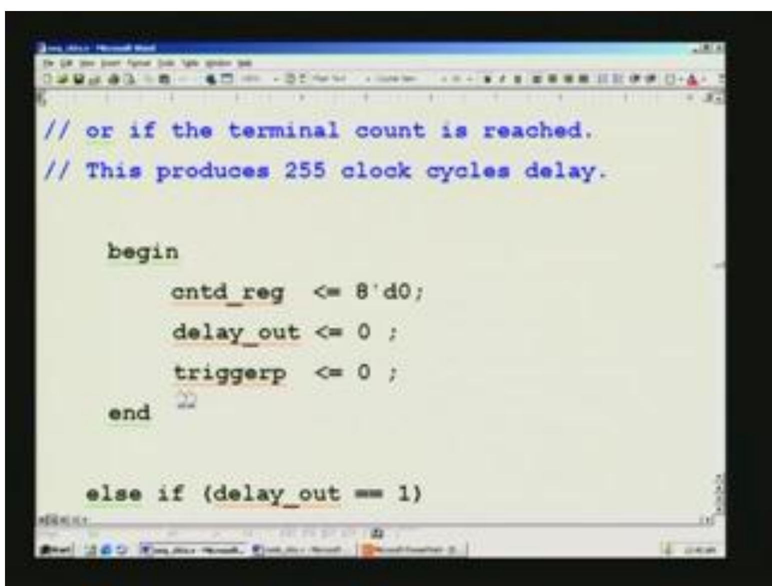
This is based upon like being equal to 255 and with a trigger you start the timing action and outcome the delay here. This is the delay out and this is the internal counter that runs here. If you wish to ease, you can use that, otherwise you can just ignore that. Now, you might have noticed this is the change I have made here, also here and that is precisely

what is written here for run delay we sense the raising edge here. As usual, we prerequisite increment the register.

Earlier I had used just cnt. I have renamed, it has d; to indicate that it is a delay that we really want to do. So, I had just renamed it, because otherwise that earlier counter will be overwritten by this, or this will be overwritten by the other one- we cannot predict that.

We have an assignment of always block which does the actual sequential work. That is the counting; so, that is what is here.

(Refer Slide Time: 52:48)



With the reset condition being satisfied, you reset the system. All these are different counters, delay output and there is an internal storage for trigger condition to store the previous status. That is what is here.

(Refer Slide Time: 53: 03)

```
        delay_out <= 0 ;
        triggerp  <= 0 ;
    end

  else if (delay_out == 1)
    begin
              cntd_reg <= cntd_next ;

// Advance the count by one if the
// timer is still running.
              triggerp <= trigger ;
```

If the timer is already running, it would have been forced to 1. So, as long as it is running then only the increment is to be done and that is why this increment is here.

(Refer Slide Time: 53:15)

```
// Advance the count by one if the
// timer is still running.
              triggerp <= trigger ;

      end

  else if (run_delay== 1'b1)
    begin
        delay_out <= 1 ;
```

We do not forget to preserve the current trigger value as a previous value so that we use it in the clock cycle.

(Refer Slide Time: 53:27)

```
    end

  else if (run_delay== 1'b1)
    begin
        delay_out <= 1 ;

// Start the delay if the positive edge of
// trigger is detected.

          triggerp <= trigger ;
// Preserve the current state of trigger.
```

If a run delay is actually the sensing of the raising edge; if it is 1, then you have to stop the timer action, thereby making delay out as 1.

(Refer Slide Time: 53: 41)

```
// Start the delay if the positive edge of
// trigger is detected.

          triggerp <= trigger ;
// Preserve the current state of trigger.

      end

  else
    begin
```

Once again, you preserve this here.

(Refer Slide Time: 53:46)

```
else
  begin
        cntd_reg <= cntd_reg ;
        delay_out <= delay_out ;
        triggerp <= trigger ;
// Otherwise, don't disturb.

  end

end
```

The whole thing ends there. This is if else you do not do anything, but just preserve what you have here. We have some more circuits say shift register etc. I think, we will cover this in your next class. Thank You.

Summary of Lecture 14

Next Lecture

Coding Organization – Complete Realization Continued

Complete Code for Sequential Circuits

- Right Shift Register
- Parallel to Serial Converter
- Model State Machine
- Pattern Sequence Detector

Test Bench for Combinational Circuits