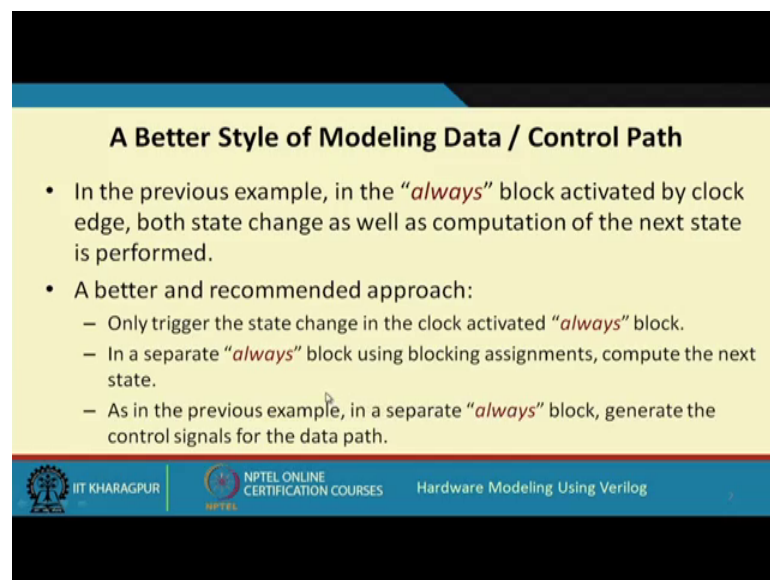


Hardware Modeling using Verilog
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 26
Datapath And Controller Design (Part 2)



So, we were discussing how data path and control paths can be together designed in complex digital systems. So, if you recall in our last lecture we gave a simple example of a multiplier and unsigned multiplication algorithm, which worked on the principle of repeated addition. So, we take a few more examples So that the methodology becomes more or less clear to you how do we do it.

(Refer Slide Time: 00:56)



A Better Style of Modeling Data / Control Path

- In the previous example, in the “*always*” block activated by clock edge, both state change as well as computation of the next state is performed.
- A better and recommended approach:
 - Only trigger the state change in the clock activated “*always*” block.
 - In a separate “*always*” block using blocking assignments, compute the next state.
 - As in the previous example, in a separate “*always*” block, generate the control signals for the data path.

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

So, we continue with our discussion data path and controller design part 2. Now just one thing you recall in the design that we discussed in the last lecture, we had designed the data path we had designed the control path. Now if you recall the way we had designed the verilog code model for the controller, there we had used one always block which was activated by the clock, which was carrying out this state changes depending on the input conditions it was checking the input conditions based on that it was going to the appropriate next state.

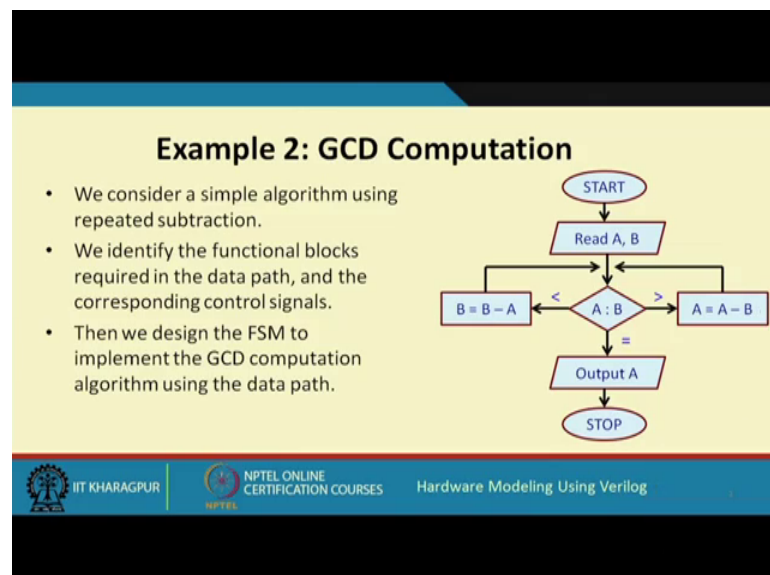
And in another always block which was based on blocking statements there we were generating the control signals for the data path. Now here we shall talk about an alternate

way of implementing the same thing. So, this in fact is a you can say a better style of modeling. So, the difference is as follows as I said in our earlier example the always block which was activated by the clock. So, the inside that always block. So, we were manipulating the calculation of the next state as well as we were assigning the next state. Both the things were performed in the same block.

Now what we say here is that a better and recommended approach may be. So, we use a simpler always block which will only trigger the state changes. So, it will not make any calculation or computation as to what the next state will be. And in one or more always block separate always blocks both of them will be using blocking statements blocking assignments.

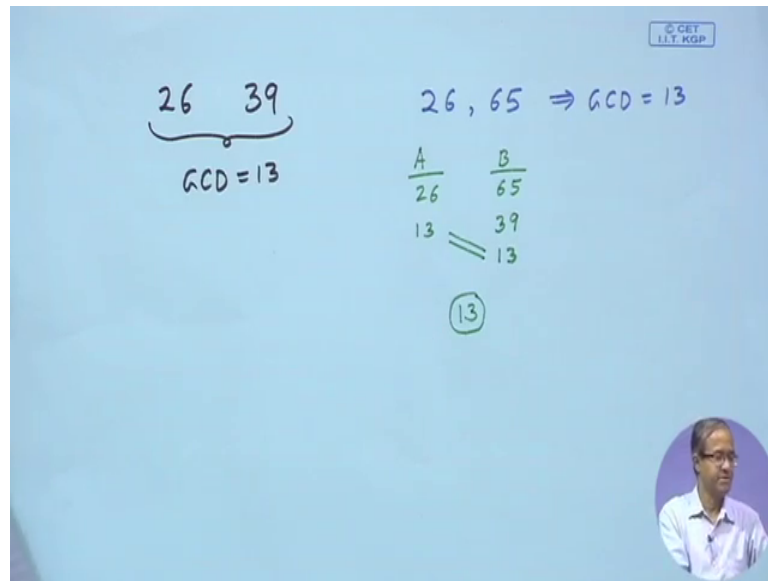
So, here you compute the next state as well as you generates the control signals for the data path. So, in this lecture we shall be taking another example, namely that of computing the GCD of 2 numbers greatest common divisor. And we shall first be following the previous approach for which what we did in the last example. Then we shall see how this new proposed method that we are talking about, how we can make the design modification to incorporate this right ok.

(Refer Slide Time: 03:34)



So, the problem that we consider here is GCD computation. Greatest common divisor, now you recall the GCD of 2 numbers is defined as the largest integer that divides both of them.

(Refer Slide Time: 03:56)



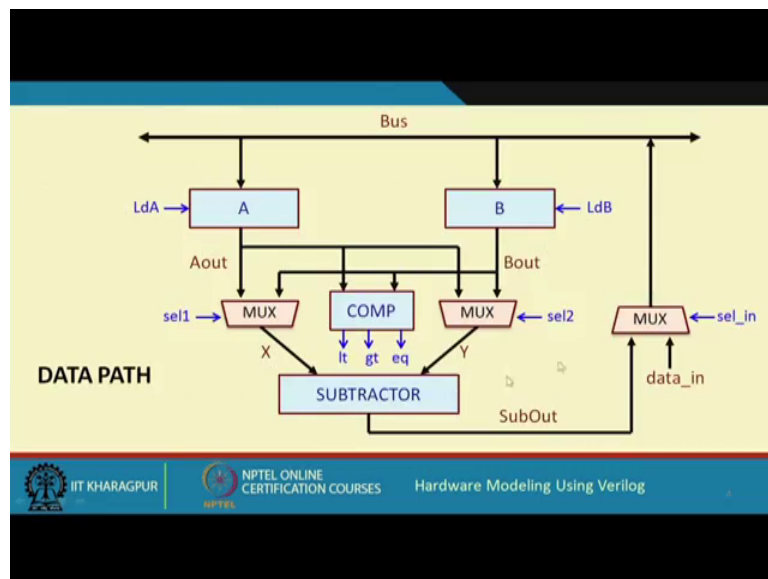
For example If you have one number as 26, one number as 39, and if you compute the GCD of them GCD will be 13, because 13 is the largest number which divides 26 as well as 39. Now the method we follow is the method of repeated subtraction. So, the algorithm is depicted in this flowchart you can see is very simple. So, you read the 2 numbers, for whom you are trying to compute the GCD then you compare A and B if A is less than B, then subtract A from B. Compute B minus A store the result in B. So, if you see A is greater than B then you subtract B from A, and if you find that equal then you have already got the result ok.

So, let us take an example well 26 and 31 may be too simple, an example let us take an example of 26, and let us say how much? Fine 65. 26 65 here also GCD is expected to be 13 because both 26 and 35 is divisible by 13. Now the method we follow. So, we use the same method a let us say it is 26 B is 65. So, in the first step we see B is greater than a subtract A from B. So, if you subtract 26 from 65 it will become 39. Then again compare again B is greater than a subtract A from B, it is now 20 39 minus 26 is 13. Now again compare A and B now A is greater subtract B from a it becomes 13. So now, you have a situation where both A and B are equal. If A and B are equal then either A or B will be the required GCD this is the algorithm right. So, we have actually used a simple algorithm like that.

So, as before here also we shall be identifying the functional block that we required in the data path and identify the control signal and design the FSM to execute this in sequence. Now if you look at this flow chart you can identify what are the elements in the data path that are required. First you will have to read the 2 numbers A and B for them you will be needing 2 registers. Then you need subtraction. So, you need a subtractor, but one thing you see sometimes you are doing be my sometimes you are doing A minus B.

So, even if you have a single subtractor the 2 inputs of the subtractor must come through a multiplexer. So, you will have to select either A or B either A or B depending on whether A is less than B or greater than B one of them will be coming to the input of the subtractor. And finally, to carry out the comparison you need some kind of a comparator, you need a comparator circuit also and of course, lastly left output the value.

(Refer Slide Time: 07:27)

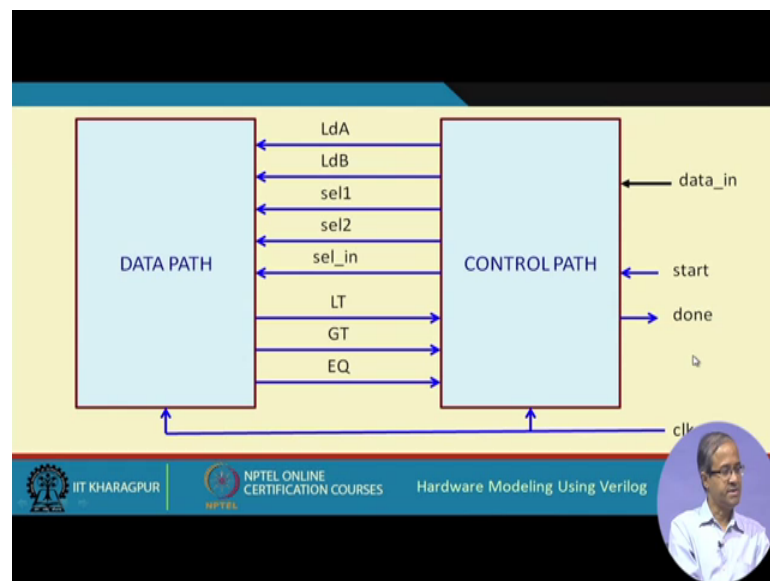


So, let us see the overall data path as I have said. So, we will need 2 registers A and B to store our initial numbers. Then as I said we need a subtractor. The first input conveyor of the subtractor can be either A or B that is why we have used a multiplexer with inputs coming from A and also from B for the other input also there is another multiplexer input coming from A as well as B. There are select inputs of this mux's cell 1 and cell 2 they will select which one of A and B are selected. If select 1 is 0 and select 2 is 1, then here a will be selected from here B will be selected.

And here there is a comparator which will be comparing the values of A and B and will be generating the 3 status signals less than greater than or equal to. And there is another multiplexer. This multiplexer will be means either selecting an external data signal data in you see initially when you load the values in A and B, we need the data in to come to the bus and be loaded to B or to A. And for loading we have this load A and load B control signals.

And this selecting control signal of the multiplexer will select which of the inputs are selected. And during the normal computation after subtraction the result will have to be stored back in to either A or B. For that I will have to select this input of the multiplexer this sub out will be coming to the bus and will be selecting again either load A or load B. So, this is how the data path functional components look like, and you can also see the control signals. Load A load B select 1 select 2 select in and these are the status signal which are generated by the data path less than greater than and equal to. So, the outputs of the multiplexer we are calling as X and Y fine.

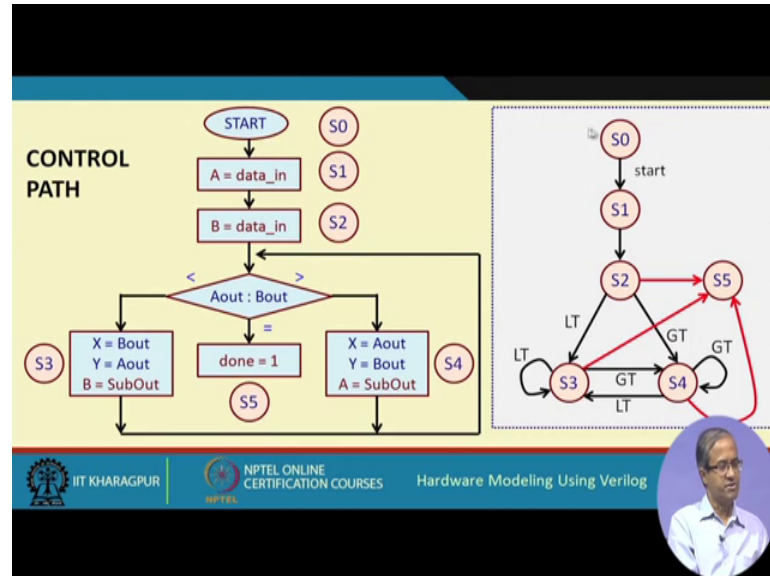
(Refer Slide Time: 09:36)



So, schematically data path treated as a black box as I said. These are the 5 control signals which are required, and these are the 3 status signals which it is generating. So, we will now have to design the control path in a similar way as earlier. So, for the control path there will be a clock of course, the data in which will also come to the data path, and

there will be a start signal, which will initiate the computation of the GCD and after it is done the done signal will be activated. This is the signals for the interface.

(Refer Slide Time: 10:20)



Now, let us come to the control path. Now that same flow chart we are just showing in a slightly different way stepwise from the beginning. Let us say we load the data in signal which is coming from outside in to A first then data in to B first. Next we are calling them as states s 0s initial state this is state 1 this is state 2. Then we compare the outputs of the A or B register which you have called as A out or B out.

If it is less then B out goes to X and A out goes to Y, what does this mean? You see B out goes to X and A out to goes to Y; that means, I am selecting this multiplexers appropriately. Similarly if it is greater, the other inputs are selected A out goes to X B out to Y and the result goes back to be or to A. And if they are equal the result is already in either A or B and you are activating done, this state is s 3 s 4 and s 5; so in terms of the state transition. So, you can depict this like this starting from s 0. So, whenever start is activated we go to s 1, then to s 2 then from s 2 depending on the condition 3 conditions are there, if it is less than we go to state s 3 if it is greater than 2 s 4 if it is equal we go to you see this red arrows indicate the equal to conditions equal to it goes to s 5.

Similarly when you are in s 3 see after s 3 you again go back like this. So, you can again come to s 3 or you can come to s 4 or you can come to s 5. So, from s 3 again if it still remains less than you remain in s 3 if it is greater than you go to s 4 if it is equal you

finally, go to s 5. Similarly for is 4 if it is greater than you remain here if it is less than you go to s 3 if it is equal you go to s 5.

(Refer Slide Time: 12:32)

```
module GCD_datapath (gt, lt, eq, ldA, ldB, sel1, sel2, sel_in,
data_in, clk);
input ldA, ldB, sel1, sel2, sel_in, clk;
input [15:0] data_in;
output gt, lt, eq;
wire [15:0] Aout, Bout, X, Y, Bus, SubOut;

PIFO A (Aout, Bus, ldA, clk);
PIFO B (Bout, Bus, ldB, clk);
MUX MUX_in1 (X, Aout, Bout, sel1);
MUX MUX_in2 (Y, Aout, Bout, sel2);
MUX MUX_load (Bus, SubOut, data_in, sel_in);
SUB SB (SubOut, X, Y);
COMPARE COMP (lt, gt, eq, Aout, Bout);
endmodule
```

So, this FSM we have to implement. So, here we are showing the FSM implementation using the approach which we showed with the previous example first. So, we are first defining the data path where these are these signals gt, lt, eq are the output signals. And all the others these 5 are the select signals and of course, data in and clock out there. These are the control signals. And load A load B they multiplex a select and clock at the input signals data in I am assuming that is a 16 bit data and these are outputs. And this intermediate signals A out B out X Y bus and the output of the subtractor, these are all assumed to be 16 bit buses of type y.

Now, just in this schematic we have seen we use 2 registers A and B, 3 multiplexers one subtractor and a comparator. So, we have just instantiated that many functional blocks here. Parallel in parallel out is a register we have already defined I will show it. So, here A out is the output versus the input load and clock are the control signals. Similarly for pipo B for the multiplexer this X is the output A out B out are the inputs and select 1 is the select. Similarly for the second multiplexer; for the third multiplexer versus the output sub out and data in at the inputs and selecting is the select. And there is a subtractor where sub out is the output X and Y are the inputs and there is a comparator finally, less than greater than equal to the outputs and these are the 2 input numbers right.

(Refer Slide Time: 14:30)

```
module PIPO (data_out, data_in,
            load, clk);
    input [15:0] data_in;
    input load, clk;
    output reg [15:0] data_out;
    always @(posedge clk)
        if (load) data_out <= data_in;
endmodule

module SUB (out, in1, in2);
    input [15:0] in1, in2;
    output reg [15:0] out;
    always @(*)
        out = in1 - in2;
endmodule

module COMPARE (lt, gt, eq, data1,
               data2);
    input [15:0] data1, data2;
    output lt, gt, eq;
    assign lt = data1 < data2;
    assign gt = data1 > data2;
    assign eq = data1 == data2;
endmodule

module MUX (out, in0, in1, sel);
    input [15:0] in0, in1;
    input sel;
    output [15:0] out;
    assign out = sel ? in1 : in0;
endmodule
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

Now, a very quick look at these descriptions; pipo here for the sake of convenience I am shown all of them in a behavioral fashion, but in a real design when your target is to achieve higher performance, we often design many of the modules in a structural way. So, that we have entire control over the way the circuit is designed, because here when you design using behavioral fashion, will leave it entirely to the synthesis tool to decide what kind of circuit it will be generating right.


So, you see for the register the parallel in parallel out data out data in load clock it is very simple, data in is the input load clock are single bit control signals. So, always that posedge of the clock if the load signal is active then data in goes to data out it is loaded. Subtractor is a combinational circuit very simple this n one n 2 out or 16 bit numbers; so the whenever anything changes always at start out equal to n 1 minus n 2.

Comparator is also very simple, data 1 and date 2 are 2 16 bit numbers, and these are the outputs. Where was 3 assign statements assign lt equal to data 1 less than data 2 which means, if this condition is true that condition will be stored in lt condition true means it is stored as 1. So, 1 will be stored in lt. If it is false 0 will be stored. Similarly for this one if this condition is true one will be stored otherwise 0 similarly equal right. Mux also is very similar this is actually a set of 16 2 to 1 multiplexers because we are multiplexing 2 inputs n 0 and n one each of them have 16 bits. So, actually these are vectors out n one n 2 depending on the select line we are either selecting n 0 or n 1 ok.

(Refer Slide Time: 16:40)

```
module controller (ldA, ldB, sel1, sel2, sel_in, done, clk, lt, gt, eq, start);
input clk, lt, gt, eq, start;
output reg ldA, ldB, sel1, sel2, sel_in, done;
reg [2:0] state;
parameter S0=3'b000, S1=3'b001, S2=3'b010, S3=3'b011, S4=3'b100, S5=3'b101;
always @(posedge clk)
begin
case (state)
S0: if (start) state <= S1;
S1: state <= S2;
S2: #2 if (eq) state <= S5;
    else if (lt) state <= S3;
    else if (gt) state <= S4;
S3: #2 if (eq) state <= S5;
    else if (lt) state <= S3;
    else if (gt) state <= S4;
S4: #2 if (eq) state <= S5;
    else if (lt) state <= S3;
    else if (gt) state <= S4;
S5: state <= S5;
default: state <= S0;
endcase
end
```

THE CONTROL PATH




Now the controller, the controller again it will be generating all the control signals for the data path done clock and less than greater than equal to and start are the signals which it is taking as input these are inputs. And the signals which it is generating for the data path these are outputs, and because they are using a procedural block you have declared them of type reg. Now in this example because there are 6 states we need a 3 bit state vector. So, we define it as a 3 bit state vector called state, and using parameter we call this state are s 0 s 1 s 2 s 3 and s 4 and s 5. Now in the first always block triggered by the clock we are just implementing that flow chart whatever I had shown. So, initially we are in state s 0, if start is active this state becomes s 1.

Well, if you are in state s 1 irrespective of anything else next state will be s 2. Well if we are in s 2 then if it is equal go to s 5 if less than s 3 greater than s 4. Same thing is done for this states s 3 and s 4. And whenever you are in state s 5; that means you are done you remain in state s 5. Well and initially if this state is not initialized. So, there is a default case which initializes the state to state s 0. This is for the state calculation. Then there is another always block which checks this state.

(Refer Slide Time: 18:18)

```
always @(state)
begin
case (state)
S0:   begin sel_in = 1; ldA = 1; ldB = 0; done = 0; end
S1:   begin sel_in = 1; ldA = 0; ldB = 1; end
S2:   if (eq) done = 1;
      else if (lt) begin
                sel1 = 1; sel2 = 0; sel_in = 0;
                #1 ldA = 0; ldB = 1;
            end
      else if (gt) begin
                sel1 = 0; sel2 = 1; sel_in = 0;
                #1 ldA = 1; ldB = 0;
            end
S3:   if (eq) done = 1;
      else if (lt) begin
                sel1 = 1; sel2 = 0; sel_in = 0;
                #1 ldA = 0; ldB = 1;
            end
      else if (gt) begin
                sel1 = 0; sel2 = 1; sel_in = 0;
                #1 ldA = 1; ldB = 0;
            end
end
end
```



Whenever the state changes there is a case statement and it selects the appropriate control signals you can actually verify.


Well, in s 0 well. So, while you are going from s 0 to s 1 you will have to load A. So, that is why load A is active you are selecting the other input of the multiplexer select in So that data in gets in to the bus while load B and done are 0. And when you are in s 1 going to s 2 you have to load B. So, load B is one again select 1 is 1, and here when you are in s 2 s 3 or s 4 it is the same you check the conditions.

If it is equal you activate the done signal, else if it is less than then you select B in the first one and A in the second multiplexer load B result will go back to B. And else you select A in the first multiplexer subtractor the multiplexer B in the second multiplexer you activate a result will go back to A. Similarly s 3 this is same, and in the same way s 4 is also same ok.


(Refer Slide Time: 19:36)

```
S4:   if (eq) done = 1;
      else if (lt) begin
            sel1 = 1; sel2 = 0; sel_in = 0;
            #1 ldA = 0; ldB = 1;
            end
      else if (gt) begin
            sel1 = 0; sel2 = 1; sel_in = 0;
            #1 ldA = 1; ldB = 0;
            end

S5:   begin
      done = 1; sel1 = 0; sel2 = 0; ldA = 0;
      ldB = 0;
      end
      default: begin ldA = 0; ldB = 0; end
    endcase
  end
endmodule
```




IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES

Hardware Modeling Using Verilog



And finally, in s 5 when you are done you activate the signal done equal to 1 and deactivate all the select lines and loads. So, that nothing else will proceed after that. And for default we are just setting load f 0 load B 0. So, you see in this example the FSM we are actually generating all the control signals, as per the flow chart whatever you require.

Well, whenever you want to select a multiplexer one input of the multiplexer we are activating cell one and the select 2 control signals accordingly. When you have to load the result in to A or B we are activating either load A or load B. So, you can just compare that flow chart with this means this means FSM description and you can see the correlation they are the same fine.

(Refer Slide Time: 20:32)

THE TEST BENCH



```
module GCD_test;
  reg [15:0] data_in;
  reg clk, start;
  wire done;

  reg [15:0] A, B;

  GCD_datapath DP (gt, lt, eq, ldA, ldB, sel1, sel2, sel_in, data_in, clk);
  controller CON (ldA, ldB, sel1, sel2, sel_in, done, clk, lt, gt, eq, start);

  initial
  begin
    clk = 1'b0;
    #3 start = 1'b1;
    #1000 $finish;
  end

  always #5 clk = ~clk;
endmodule
```

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

So, to test it out we wrote a very simple test bench. So, again we instantiated the 2 modules GCD data path and the controller, this is the clock generation initially clock was 0, and at time 3 we activated start and simulation continued till some time 1000 and after every 5 the clock was toggling.



(Refer Slide Time: 21:04)


```
initial
begin
  #12 data_in = 143;
  #10 data_in = 78;
end

initial
begin
  $monitor ($time, " %d %b", DP.Aout, done);
  $dumpfile ("gcd.vcd"); $dumpvars (0, GCD_test);
end

endmodule
```

0	x	x
5	x	0
15	143	0
35	65	0
55	52	0
65	39	0
75	26	0
85	13	0
87	13	1

 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

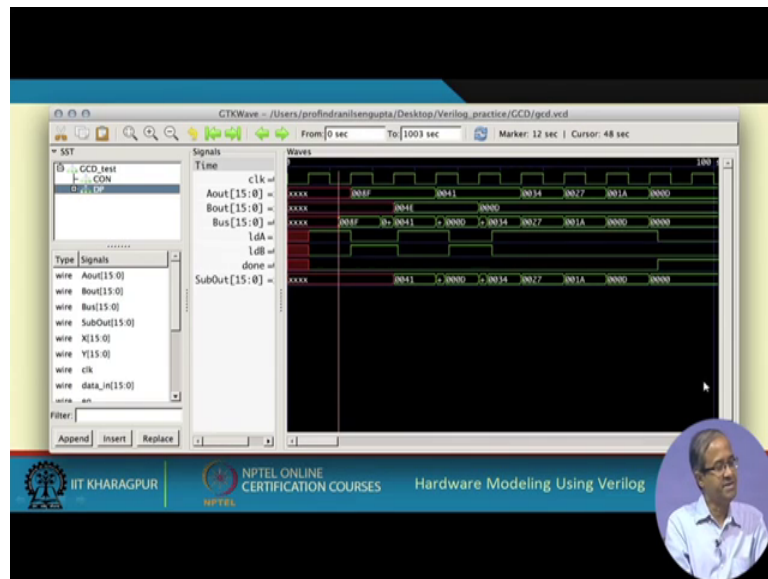


So, the clock period was 10. And these are the 2 input data we are applying at 12 and a little later after one clock 143 and 78. Well, if you just do it manually the result is supposed to be 13. And we are monitoring the time, as well as the A outs you see this

output of the A register is not available here directly, but we have instantiated this modules DP and con. Well, we can write like this DP dot A out means this A out is a signal which is defined inside the DP module. So, we want to see this because here the result will be there and of course, done and this dumb file and dumb variables we are generating here.

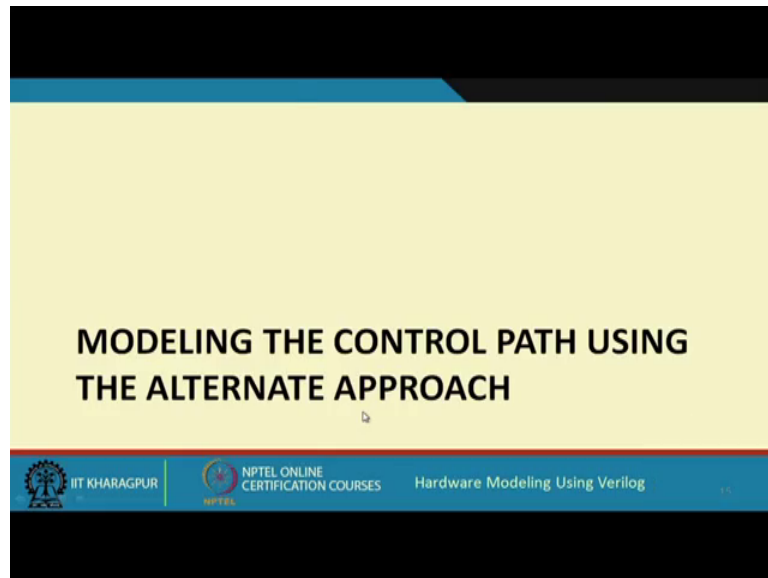
So, if you just simulate this, the result comes like this. So, the 3 things that are printed are time the value of A output of A and done. So, a changes and finally, it stops at 13 which is the final result when done also becomes 1 right. And just a simulation snap shot also is shown here.

(Refer Slide Time: 22:14)



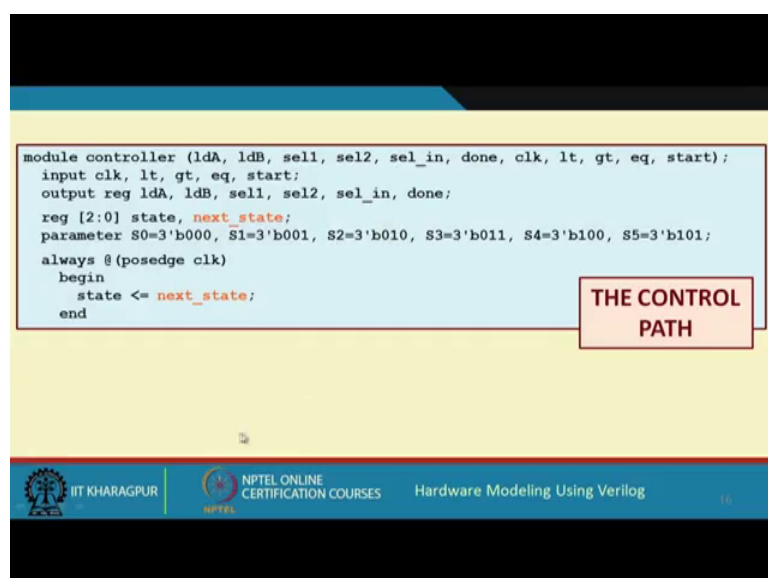
So, here also you can see the differ and signal values clock A out B out bus load A load B. So, you can see the values of A out and B out are changing. You see initially the values at 143 and 78. So, 143 means 8 f 78 means 4 e. So, 8 f minus 4 e it becomes 4 1, 4 e minus 4 1 it becomes 0 d 4 1 minus 0 d becomes 3 4. 3 4 minus 0 d is 2 7. 2 7 minus 0 d is 1 A 1 A minus 0 d is 0 d they are equal; now you stop done gets activated here when they are equal. So, the result is 0 0 0 d which is 13 right fine.

(Refer Slide Time: 23:03)



Now, the alternate approach that we were talking about let us quickly see that how our design will look like, if we use the alternate approach. There is not much difference really the 2 things we have to do. So, in the earlier case we have defined only one variable that was storing the state vector, but now we will be storing 2 different state variables. One will be the present state and other will be the next state. Now in the clock activated always block we will only write present state next present state equal to next state whatever was the next state in the last time it will become the present state now. And in the other always block we will do all other computations.

(Refer Slide Time: 23:55)



Let us see, so our main always block becomes very simple. You see here as I said we have defined another variable called next state the other part is same the header does not change, and in the first always block is very simple.

(Refer Slide Time: 24:19)

```
always @(state)
begin
case (state)
S0:   begin sel_in = 1; ldA = 1; ldB = 0; done = 0; end
S1:   begin sel_in = 1; ldA = 0; ldB = 1; end
S2:   if (eq) begin done = 1; next_state = S5; end
      else if (lt) begin
          sel1 = 1; sel2 = 0; sel_in = 0; next_state = S3;
          #1 ldA = 0; ldB = 1;
        end
      else if (gt) begin
          sel1 = 0; sel2 = 1; sel_in = 0; next_state = S4;
          #1 ldA = 1; ldB = 0;
        end
S3:   if (eq) begin done = 1; next_state = S5; end
      else if (lt) begin
          sel1 = 1; sel2 = 0; sel_in = 0; next_state = S3;
          #1 ldA = 0; ldB = 1;
        end
      else if (gt) begin
          sel1 = 0; sel2 = 1; sel_in = 0; next_state = S4;
          #1 ldA = 1; ldB = 0;
        end
end
end
```


Now, here it only says state equal to next state. And the next state is calculated in the second always block.

So, here earlier these red things were not there. The remaining control signals were there now we have made a little modification here, we have also added the statements which compute or calculate the next state. So, where whenever in s 2 and it is equal your next state will be s 5, when you are in s 2 it is less than your next state will be s 3, if it is greater than next it will be s 4.


(Refer Slide Time: 24:55)

```
S4:   if (eq) begin done = 1; next_state = S5; end
      else if (lt) begin
          sel1 = 1; sel2 = 0; sel_in = 0; next_state = S3;
          #1 ldA = 0; ldB = 1;
        end
      else if (gt) begin
          sel1 = 0; sel2 = 1; sel_in = 0; next_state = S4;
          #1 ldA = 1; ldB = 0;
        end

S5:   begin
      done = 1; sel1 = 0; sel2 = 0; ldA = 0;
      ldB = 0; next_state = S5;
    end
default: begin ldA = 0; ldB = 0; next_state = S0; end
endcase
end
endmodule
```



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES

Hardware Modeling Using Verilog

18

So, in a combinational block using blocking assignments you are doing this; so same thing in all the cases. So, you see there are 2 always blocks which are running in parallel. One will be a clock controlled thing which is whenever clock comes it will just put the next state in to state. And in the other block which is not clock controlled which is controlled by state whenever state changes you do it. So, whenever there is a change in state you make some modifications there generate the control signals, and also assign the proper value to the next state; so that whenever the clock comes the first always block will go to the correct next state right fine.

So, actually this is how this design is carried out. So, with this actually we come to the end of this lecture. Well, if you see this example was quite similar to the previous example. So, we deliberately we are taking a few examples So that you become familiar with the process of design given any problem. You see this problems are not very trivial a little complex of course, real digital systems can be much, much more complex. But still just to have a flavor of complexity we are taking some reasonable size problems which can be discussed and put on the slides right.

So, in the next lecture also I mean here next we shall be talking about another example. There we shall see that how we can carry out signed multiplication using a well-known algorithm so that we shall see in the next lecture.

Thank you.