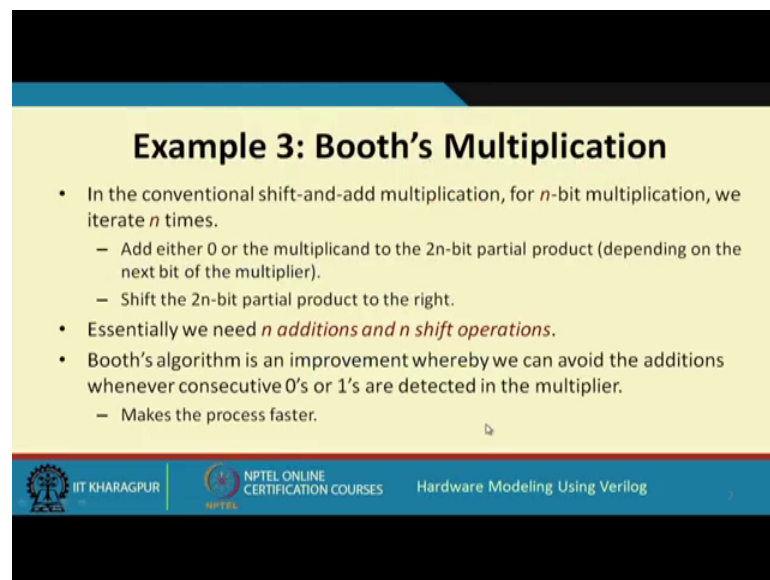


Hardware Modeling using Verilog
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 27
Datapath And Controller Design (Part 3)

So, we continue with our discussion now in this lecture. We shall be taking another example through which we shall be showing you the partitioning of the data path and the control path and how we can coat the 2 things in verilog. So, this is the part 3 of our lecture.

(Refer Slide Time: 00:38)



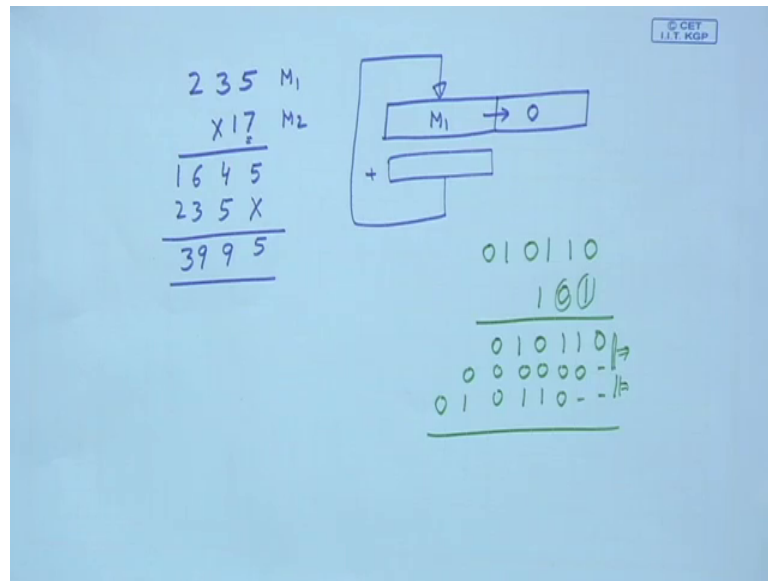
Example 3: Booth's Multiplication

- In the conventional shift-and-add multiplication, for n -bit multiplication, we iterate n times.
 - Add either 0 or the multiplicand to the $2n$ -bit partial product (depending on the next bit of the multiplier).
 - Shift the $2n$ -bit partial product to the right.
- Essentially we need n additions and n shift operations.
- Booth's algorithm is an improvement whereby we can avoid the additions whenever consecutive 0's or 1's are detected in the multiplier.
 - Makes the process faster.

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

Now, in this lecture we shall be taking an example a signed multiplication algorithm a well known algorithm called Booth's algorithm. Now you are just familiar with the conventional way we multiply numbers, like you think of the multiplication method that we learnt in schools.

(Refer Slide Time: 01:10)



Suppose we are multiplying a number 235 by say 17. So, you take the individual digits of the multiplier you multiply them 7 in to 5 35, 5 with a carry of 3 7 21, 24 with a carry of 2 7 to the 14 and 2 16. Then we shift one bit to the left we may keep a gap. Then multiply by 1 5 3 2 and then we add this is the product. So, our normal processes we keep our partial results same and whenever you look at the next digits we shift it left and add. But in a real multiplication we do slightly different.

We keep the partial result in a pair of registers, well let us say this is your M , let us say this is your M_1 and M_2 . Let us say suppose we load M_1 here and load this with 0. And depending on the next digit or next bit of M_2 we add something to M_1 . And after adding these result goes back to M_1 . Then we shift M_1 to the right 1 place. Instead of shifting left and adding we add in the same position, but the partial product will shift right it is same thing for example, 1 6 4 5 if you shift right 1 place and add 235 that is also the same thing right. So, this is how it works.

Now, in a normal shift and add multiplication this a example i showed for decimal, here you work in binary where the bits of the number are considered. Suppose when we add let us say 0 1 0 1 1 0 with let us say 1 0 1 we do the same thing like here i am showing this process, when you multiply one with this. So, it remains when you multiply 0. So, everything is 0 there is a gap when you multiply 1 again there are 2 gaps 0 1 1 0 1 0 then we add all of them up. This is how we do shift and add multiplication. So, what you do?

So, whenever you check the next bit you continuously carry out the addition here So that you do not have to store so many partial products. You store only one partial product here, and as you check these successive bits the addition is carried out and the partial result is getting stored here again.

So, at the end the final product will be stored here right. Now in the conventional method the point to notice that whenever we are multiplying $2n$ bit numbers. Then we have to check for all the n bits of the multiplier. So, we will have to repeat the process n times depending on whether the bit was 0 or 1. So, will be adding either 0 or the multiplicand to the partial products.

As I said the partial product is of size double that 2 registers we are putting side by side as the partial product. Because when you add let us say 2 8 bit numbers the result will be double the size the result can be 16 bits in size. So, multiplying $2n$ bit numbers will generate a result which will be $2n$ bits in size fine. So, this conventional shift and add method requires n additions and n shift operations. For every bit we have to add either 0 or the multiplicand depending on the next bit of the multiplier and you repeat it n times right.


Now, in this context Booth's algorithm is an improvement. Well, it requires n shift operations all right, but it requires less than n addition or subtraction operations. So, if there are consecutive 0s and 1s in the multiplier you skip the addition or subtraction step which makes the process faster.

(Refer Slide Time: 05:50)

Basic Idea Behind Booth's Algorithm

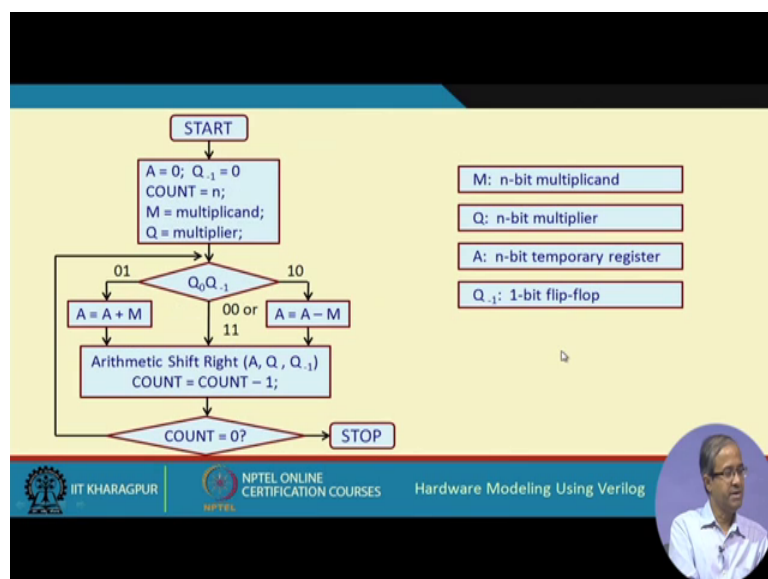
- We inspect two bits of the multiplier (Q_i, Q_{i-1}) at a time.
 - If the bits are same (00 or 11), we only shift the partial product.
 - If the bits are 01, we do an addition and then shift.
 - If the bits are 10, we do a subtraction and then shift.
- Q_{-1} is assumed to be equal to 0.
- Significantly reduces the number of additions / subtractions.

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



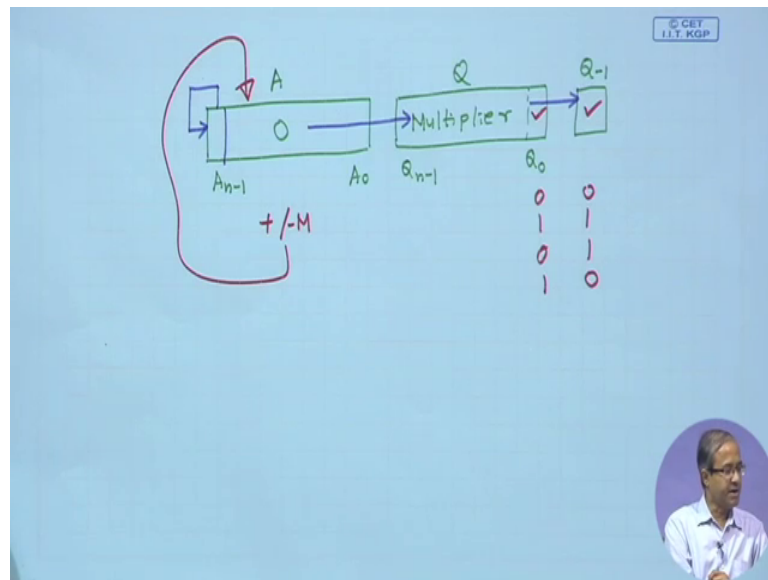
So, the basic idea behind the algorithm is that we inspect 2 bits of the multiplier at a time, i th and i minus 1th bit. So, the rule is very simple. It says if the 2 bits are either 00 or 11. There is no need to do any addition or subtraction. Then you do only shifting, but if you find that these 2 bits are 01, then you add the multiplicand and then shift if it is 10 then you subtract and then shift. And to start with Q minus 1; that means, the last bit after the least significant bit is assumed to be 0.

(Refer Slide Time: 06:49)



And because we are skipping through 0 0 and 1 1, this will reduce the number of addition subtractions in the process. This is the flow chart of the Booth's multiplier. So, you see here we have a register A initialize to 0, and Q which contains the multiplier. So, it is something like this.

(Refer Slide Time: 07:08)



There are 2 registers we are saying this is A and this is Q. Initially A is initialized to 0 and Q is loaded with the multiplier. And we check this bit and there is another flip flop we say call it Q minus 1. Because this Q will start from Q 0 up to Q let us say n minus 1 if it is n bit register. And a will start from a 0 up to a n minus 1. So, let us say this is Q minus 1; so will be checking 2 bits at a time like this. So, what we do? And there is a register count which will contain the number of bits that mean how many times you have to repeat, that is n. And M contains multiplicand and this Q contains multiplier. So, so in every step we check Q 0 and Q minus 1 we check this Q 0 and Q minus 1 whether they are 0 0, 1 1, 0 1 or 1 0.

So, we check the conditions. So, if it is 0 0 or 1 1 we skip the addition or subtraction step. If it is 0 1 we do A equal to a plus M if it is 1 0 we do A equal to A minus M. So, with this a we either add or subtract M, and the result we store back in to a right. And then we do a right shift we do arithmetic right shift of A Q and Q 1 M minus 1 all together. This means i right shift here in to multiplier will get right shifted in to Q minus

1. And arithmetic right shift means the most significant bit of a which is the sign bit that will be shifted back.

So, if A is negative it will remain negative right. And then a decrement count and repeat till count is not 0, right. So, these are the basic registers which are required in addition we need an adder subtractor. So, let us work out a couple of examples to have a feeling how the multiplication works. Let us take a simple example.

(Refer Slide Time: 09:53)

Example 1: $(-10) \times (13)$
 Assume 5-bit numbers.
 M: $(10110)_2$
 -M: $(01010)_2$
 Q: $(01101)_2$
 Product $\rightarrow -130$
 $= (110111110)_2$

A	Q	Q ₋₁	Operation	Step
00000	01101	10	Initialization	
01010	01101	0	$A = A - M$	Step 1
00101	00110	1	Shift	
11011	00110	1	$A = A + M$	Step 2
11101	10011	0	Shift	
00111	10011	0	$A = A - M$	Step 3
00011	11001	1	Shift	
00001	11110	1	Shift	Step 4
10111	11100	1	$A = A + M$	Step 5
11011	11110	0	Shift	

NPTEL ONLINE CERTIFICATION COURSES
 IIT KHARAGPUR
 Hardware Modeling Using Verilog

Let us say we multiply minus 10 with 13 we assume these are 5 bit numbers. M minus 10 is 1 0 1 1 0 in (Refer Time: 09:57) complement representation. Well, just for convenience i am also showing minus M minus M will mean plus 10 plus 10 in binaries 0 1 0 1 0. And Q the multiplier is 13. So, we load A with 0, Q with the multiplier and Q minus 1 is 0. Then we check these 2 bits Q 0 and Q minus 1. So, it is 1 0 means we have to subtract M. Well, just for convenience subtracting M means adding minus M. So, it will be easier to understand that is why i have shown minus M. So, we subtract M means we add 0 1 0 1 0 minus M we add, So A becomes 0 1 0 1 0. Then we do a right shift whole right shift this 0 gets shifted and everything gets shifted right 1 place and this one gets shifted here.

So, this process will repeat next we say this is 1 0 1 which means we have to add to this a will have to add M. So, if you 0 0 1 0 1 plus 1 0 1 1 0 if you do you will see you will get the result 1 1 0 1 1 this you can check if you do it. Then you shift in the same place this is arithmetic shift. So, this one gets shifted. Now the 2 bits are 1 0. So, it is again

subtraction. So, we again do a subtraction; that means, we add minus M to this 0 1 0 1 0 add to this you get to 0 0 1 1 1 then do a right shift. Now we have 1 1, 1 1 means no need to do add or subtract just shift. Simply shift, simply shift now you get 0 1; that means, addition. So, it is 5 bits 5 steps are over and your final result minus 1 30 which is in close complement is this you see the same result is obtained. This is our final result ok.

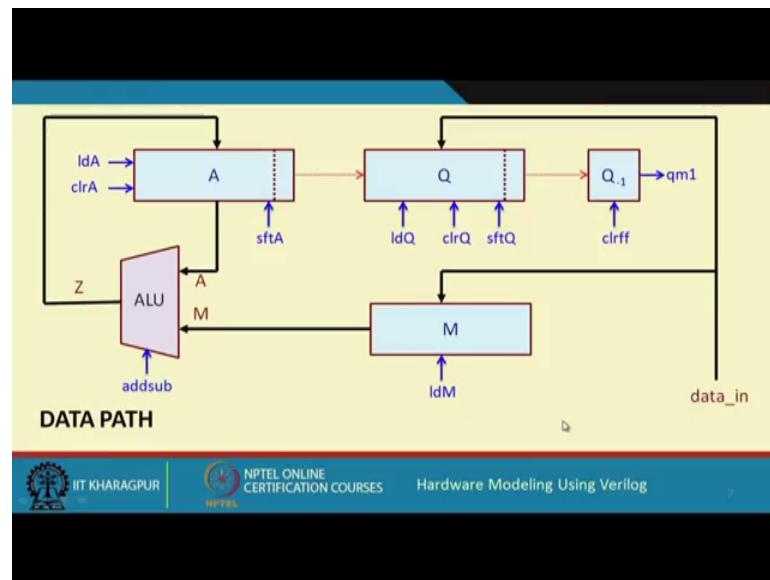
(Refer Slide Time: 12:09)

	A	Q	Q ₋₁	
Example 2:				
$(-31) \times (28)$				
Assume 6-bit numbers.				
M: $(100001)_2$				
-M: $(011111)_2$				
Q: $(011100)_2$				
Product = -868				
$= (110010011100)_2$				
	0 0 0 0 0 0	0 1 1 1 0	0	Initialization
	0 0 0 0 0 0	0 0 1 1 1	0	Shift Step 1
	0 0 0 0 0 0	0 0 0 1 1	1	Shift Step 2
	0 1 1 1 1 1	0 0 0 1 1 1	0	A = A - M Step 3
	0 0 1 1 1 1	1 0 0 0 1	1	Shift Step 3
	0 0 0 1 1 1	1 1 0 0 0	1	Shift Step 4
	0 0 0 0 1 1	1 1 1 0 0	0	Shift Step 5
	1 0 0 1 0 0	1 1 1 0 0 0	1	A = A + M Step 6
	1 1 0 0 1 0	0 1 1 1 0 0	0	Shift Step 6

Let us take another example. Suppose we are multiplying minus 31 with 28 and in 6 bit representation let us say number have 6 bits, multiplier Q 28 is in 6 bits A is also 6 bits. So, M minus 31 and also minus M is shown. So, you see 0 0 just shifting no addition subtraction only shift right. So, again 0 0 shift again now it is 1 0 subtract. So, minus M you add to this it becomes this, then shift then again 1 1 just only shift; so again 1 1 just shift again now 0 1. So, you will have to add to this you add M this will be the result and then shift and this will be your final result. So, you see here you need only 2 additions or subtractions rest of the time you are only shifting.

So, in general when in the multiplier you have consecutive 0s are 1s you can go on. Shifting this whole thing a Q and Q minus 1, without doing any addition or subtraction right ok.

(Refer Slide Time: 13:33)

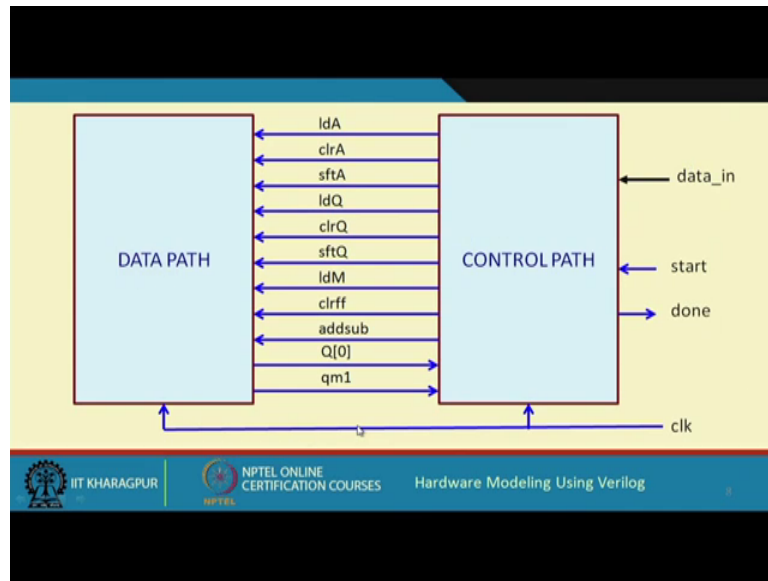


So now let us come to the data path now i have understood what we are doing. First thing is that a Q and Q minus 1 must be connected as a shift register. So, A is an n bit register Q is an n bit register Q minus 1 is a flip flop. So, an M contains the multiplicand. So, from outside we will have to load the multiplicand, as well as will have to load the multiplier. So, for that we need a load M control signal and a load Q control signal right.

Well of course, we do not require this clear Q, but because we are using 2 registers of the same type, this a needs to be cleared. So, there is a clear control signal, and there is a load control signal also A because the output of the addition or subtraction will get stored back in to a for that there is a load A. And there is a shift control signal shift A shift Q this is the shift register mode control signal. And for the flip flop there is a signal to clear it reset clear flip flop and the output of the flip flop we are calling it Q M 1.

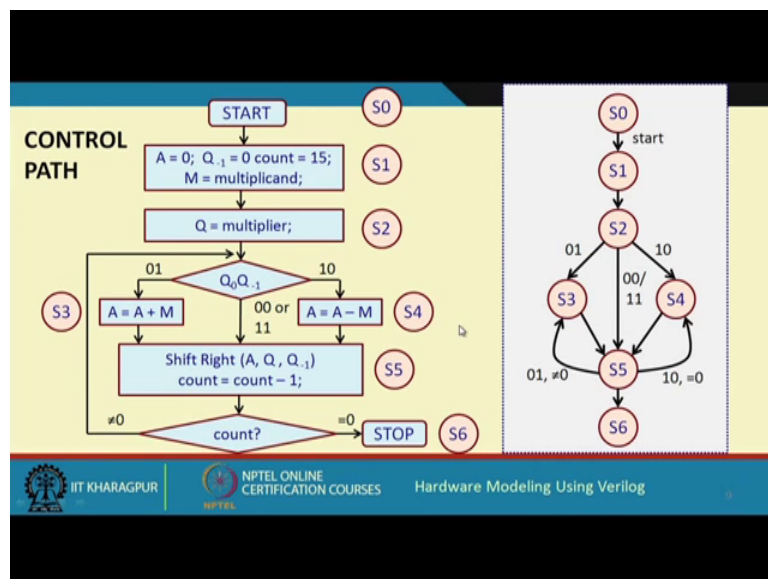
And this ALU takes A and M as the 2 inputs, and there is a control signal add or subtract. So, if it is 0 it will add if it is 1 it will subtract the output let us call it z.

(Refer Slide Time: 15:09)



So, it is very simple. So, data path again there are so many control signals as we illuminate it, and from the data path Q 0 and Q M 1 are coming this Q 0 this 0 and Q 1. Because these 2 bits we have to check to decide that whether to add or subtract or only shift right. And the rest signals are the same. Now let us come to the control path. So, again the same flow chart i am showing and here i am identifying the states.

(Refer Slide Time: 15:34)



Now, this loading of the data i have divided in 2 states, multiplicand M multiplier Q you see count equal to 15 you can do together because count here although i have not shown

here, there will be another register called count which will be loaded with 15 and to be decremented right, but that can be done in parallel. Because from data in your loading either M or Q, you cannot do these 2 things in together in parallel. That is why loading M and loading Q are in 2 different states. Then depending on this Q₀ Q₁ if it is 0 1 i do addition i call it s 3 1 0 subtraction i call it s 4 or otherwise shifting i call it as 5. At the end when you are done count equal to 0 i call it s 6

So, the state transition diagram will look like this if you see compare side by side. So, from s 0 whenever we give the start signal it goes to s 1 then 2 s 2. And from s 2 depending on the result of the comparison i can either go to s 3 or s 5 or s 4 depending on these 2 bits. Now once i mean s 3 well i can either go back you see from here i go to s 5 5 sorry, from s 3 i go to s 5 there is only one path. S 3 to s 5 and also s 4 s 5 s 3 to s 5 s 4 s 5. And from s 5 there is a comparison i either go to s 6. Or i go back in this comparison and can go to s 3 or to s 4. So, from s 5 i can go to s 3 or to s 4. So, the condition for going to s 3 is that Q₀ Q₁ minus 1 is 0 1 and this count is not 0. And we go to s 4 if it is 1 0 and the count is not 0 sorry this will also not 0.

And if it is 0 then it will come to s 6 right fine.

(Refer Slide Time: 17:55)

```

module BOOTH (ldA, ldQ, ldM, clrA, clrQ, clrff, sftA, sftQ,
              addsub, decr, ldcnt, data_in, clk, qm1, eqz);
  input ldA, ldQ, ldM, clrA, clrQ, clrff, sftA, sftQ, addsub, clk;
  input [15:0] data_in;
  output qm1, eqz;
  wire [15:0] A, M, Q, Z;
  wire [4:0] count;

  assign eqz = ~count;

  shiftreg AR (A, Z, A[15], clk, ldA, clrA, sftA);
  shiftreg QR (Q, data_in, A[0], clk, ldQ, clrQ, sftQ);
  dff QM1 (Q[0], qm1, clk, clrff);
  PIPO MR (data_in, M, clk, ldM);
  ALU AS (Z, A, M, addsub);
  counter CN (count, decr, ldcnt, clk);
endmodule

```

So now we can straight away write down the data path just from this diagram which is shown the same thing, just see here these are the parameters. The exact the signals that we have identified their load A load Q load M clear a clear Q clear flip flop these are all

input signals. Data in which is coming from outside to load M and Q this is an input signal 16 bits. And Q M 1 and equal to z 0 these are output signal which will go in to the controller. And A M Q z a temporary signals of type wire. And count i need i need to initialize the count to 15 right 15 and go down to 0. So, 15 i can store in 4 bits 5 bits. So, i use a counter of appropriate size. So, 15 i could have used a 3 bit 3 to 0 a 4 bit counter also. So, there is no problem.

So, using an assign statement i use a reduction and operation this is actually an nand to just check for it actually, this is for checking for 0 the count is 0 or not reduction or. So, you take the or of all the bits and then you do or not. So, if the bits are all 0s then it will become 1. Now you instantiate this models. Shift register A shift register Q or deep flip flop then there will be a parallel in parallel resistor for M, and ALU which will do addition or subtraction, and a counter which will take the value of count and decrement it right.

(Refer Slide Time: 20:08)

```

module shiftreg (data_out,data_in,
  s_in, clk, ld, clr, sft);
  input s_in, clk, ld, clr, sft;
  input [15:0] data_in;
  output reg [15:0] data_out;

  always @(posedge clk)
  begin
    if (clr) data_out <= 0;
    else if (ld)
      data_out <= data_in;
    else if (sft)
      data_out <= {s_in,data_out[15:1]};
    end
endmodule

module PIP0 (data_out,data_in, clk, load);
  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 dff (d, q, clk, clr);
  input d, clk, clr;
  output reg q;

  always @(posedge clk)
    if (clr) q <= 0;
    else q <= d;
endmodule

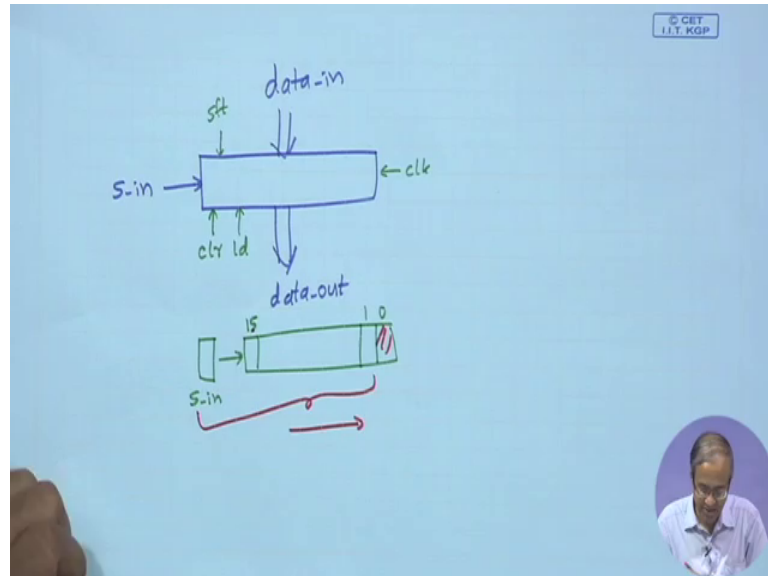
```

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

So, these modules you can actually verify the parameters i am showing the different modules that you have defined. This is shift register module. So, here the parameters are the arguments data out is the output data in. So, it is like this, it is a shift register the output parallel data out is you call as data out. The input you call as data in. Let us see the other signals serial in clock load clear shift. So, there is a serial in signal for shift register. Then in addition there are some control signals, like there is a clear signal you

can clear this register, then you can load this register with data in load signal. Then there is also another signal shift. So, when you want to shift it.

(Refer Slide Time: 21:08)



And of course, there will be a clock. These are all the signals i means at the boundary of a of this shift register. So, this is a shift register we have designed here. Data in is a 16 bit number data out also 16 bit and this is our description. So, whenever clock comes this is synchronous clear and load we check if clear is one if clear is one data out becomes 0. So, initialize it to 0 else if load is active then whatever there is data in that goes to data it is loaded. Else if shift control is active then this s in and data out 15 up to 1 this entire thing gets shifted left by one position in to data out Right.

So, this is shift right. So, this is how we are implementing shift right. Data out 15 to you see, data out data out is a 16 bit quantity right. This is your bit 0 this is bit 1 and this is bit 15. And there is something which is coming from outside this is your s in, now you want to shift it right. So, what we expect is that, this bit should go out and whatever is here this should be shifted right one position and go here. So, s in and bit number 15 to 1 whole together should be assigned here. That is exactly what you wrote a s in and data out 15 to 1 is assigned to data out. This creates the shifting. Similar the pipo register is very simple parallel in parallel out.

So, it has a data out data in clock and load. So, here whenever clock is coming if load is active data in goes to date out. And then we have the flip flop we already seen a flip flop description earlier.


(Refer Slide Time: 23:26)

```
module ALU (out, in1, in2, addsub);
input [15:0] in1, in2;
input addsub;
output reg [15:0] out;


always @(*)
begin
if (addsub == 0) out = in1 - in2;
else out = in1 + in2;
end
endmodule

module counter (data_out, decr, ldcnt, clk)
input decr, clk;
output [4:0] data_out;

always @(posedge clk)
begin
if (ldcnt) data_out <= 5'b10000;
else if (decr) data_out <= data_out - 1;
end
endmodule
```




IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES

Hardware Modeling Using Verilog




So, it has a clear input if clear is active then Q becomes 0 output else Q becomes equal to the input d. Then alu output d to input and there is a control signal, if add sub is equal to 0 you do subtraction if it is 1 you do addition, and there is a counter. So, if there is a load count signal. So, initially if you activate load count. So, it will be initialized to data out is initialized to 1 0 0 0 0, else if decrement is active data out is decremented by 1. So, you load it with 16 1 0 0 0 0 mean 16 you decrement it one by one and you check whether the result is 0 or not.

(Refer Slide Time: 24:18)

```
module controller (ldA, clrA, sftA, ldQ, clrQ, sftQ, ldM, clrff, addsub, start,
                  decr, ldcnt, done, clk, q0, qm1);
input clk, q0, qm1, start;
output reg ldA, clrA, sftA, ldQ, clrQ, sftQ, ldM, clrff, addsub, decr, ldcnt, done;
reg [2:0] state;
parameter S0=3'b000, S1=3'b001, S2=3'b010, S3=3'b011, S4=3'b100, S5=3'b101, S6=3'b110;
always @(posedge clk)
begin
    case (state)
    S0:    if (start) state <= S1;
    S1:    state <= S2;
    S2:    #2 if ((q0,qm1)==2'b01) state <= S3;
           else if ((q0,qm1)==1'b10) state <= S4;
           else state <= S5;
    S3:    state <= S5;
    S4:    state <= S5;
    S5:    #2 if (((q0,qm1)==2'b01) && !eqz) state <= S3;
           else if (((q0,qm1)==2'b10) && !eqz) state <= S4;
           else if (eqz) state <= S6;
    S6:    state <= S6;
    default: state <= S0;
    endcase
end
```

THE CONTROL PATH



Whenever it is 0 you stop right. So, you initialize it with 16. Now the controller again if you follow that state transition diagram we have just coded it in the same way. So, I would recommend that you look at this controller design carefully we saw is the state transition diagram that we have discussed earlier. And see whether these specifications are matching ok.

So, here again the parameters or the arguments are the same as whatever the controller is generating or taking as inputs Q 0 Q M 1. There are 7 states s 0 to s 6. So, the state transitions are actually specified as per that diagram. From s 0 if start is active go to s 1 from s 1 go to s 2 now in s 2 you check whether these 2 bits Q 0 and Q 1 are 0 1 if it is 0 1 go to s 3 if it is 1 0 go to x 4 or otherwise go to s 5 means 0 0 or 1 1. So, which is s 3 you straight away go to s 5 s 4 also goes to s 5. So, in s 5 also you check if bits are 0 0 and not equal to 0 yet then go to s 3.

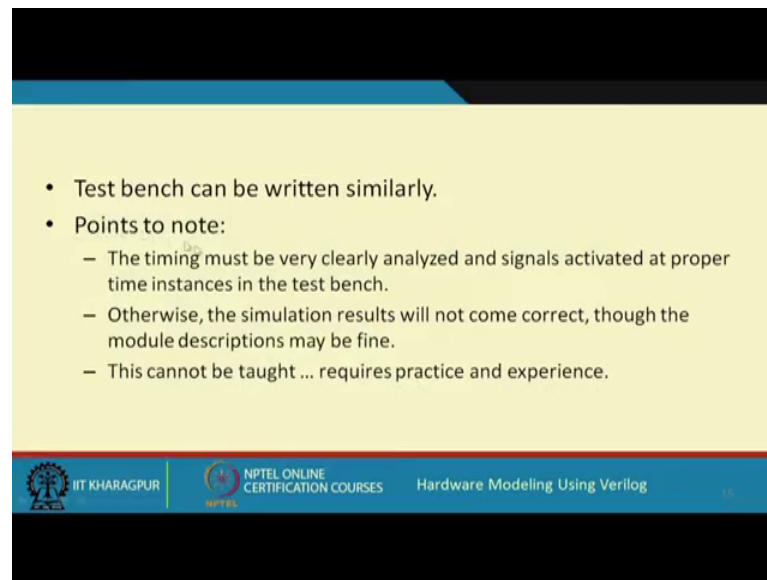
(Refer Slide Time: 25:44)

```
always @(state)
begin
  case (state)
    S0: begin clrA = 0; ldA = 0; sftA = 0; clrQ = 0; ldQ = 0; sftQ = 0;
        ldM = 0; clrff = 0; done = 0; end
    S1: begin clrA = 1; clrff = 1; ldcnt = 1; ldM = 1; end
    S2: begin clrA = 0; clrff = 0; ldcnt = 0; ldM = 0; ldQ = 1; end
    S3: begin ldA = 1; addsub = 1; ldQ = 0; sftA = 0; sftQ = 0; decr = 0; end
    S4: begin ldA = 1; addsub = 0; ldQ = 0; sftA = 0; sftQ = 0; decr = 0; end
    S5: begin sftA = 1; sftQ = 1; ldA = 0; ldQ = 0; decr = 1; end
    S6: done = 1;
    default: begin clrA = 0; sftA = 0; ldQ = 0; sftQ = 0; end
  endcase
end
```

If it is 1 0 and not equal to 0 go to s 4, but it was equal to 0 you are done you go to s 6 and once in s 6 you remain in s 6. Similarly in the other block this is the blocking assignments here you have an always with state whenever state changes you just activate this. So, here all the control signals will be generated appropriately just you can check this.

So, here you are loading M load M equal to 1 loading count the counter also you are loading, the flip flop your clearing a also you are clearing you are making a 0 you are activating all the signals together. Then in s 2 you are loading Q. Now in s 3 you are activating add sub is equal to 1 which means addition. S 4 add sub is equal to 0 which means subtraction. In this way you just follow the flow chart and see that whether we have activated the signals properly or not, ok.

(Refer Slide Time: 26:46)



• Test bench can be written similarly.

• Points to note:

- The timing must be very clearly analyzed and signals activated at proper time instances in the test bench.
- Otherwise, the simulation results will not come correct, though the module descriptions may be fine.
- This cannot be taught ... requires practice and experience.

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

So, the test bench i am not showing the test bench you can write in a very similar way, but one thing remember, here when you write such complex descriptions writing the test bench is not that easy. There are 2 things first thing is that you see the way i wrote i showed you the modules. So, i gave some delays in some places in some other places i did not give any delays. Now i told earlier delays are only for simulation, but when you do synthesis.

So, any real hardware will have some non-0 finite delay right. So, in the actual scenario every block whatever you do will be having some delay. So, it is always good to specify some delays even during simulation. So, that at least you have a fair idea about what is happening. Now when you give these delays like that when we generate the test bench or write the test bench writing the test bench is also not that easy now. Because you have to understand the timing very clearly in which clock in which time whatever is happening when do you have to apply the input, when do you apply with the load signal.

So, unless these are very accurately done your final result will be wrong. So, writing the test bench is also not a trivial task it is also quite involved. So, this is exactly what is mentioned here, that the timing must be very clearly analyzed and you need to activate the signal at proper time instances. So, if it is too early then some wrong values will be captured if it is too late, then maybe some other values have been computed already. Because if you do not take these in to account the simulation results will not come

correct, but again i tell you maybe simulation result is not coming correctly, but when you synthesize it your synthesized hardware might be correct, because your specification was done in a correct way. This is you can say this is something which you have to remember. This is if you want to do correct simulation your synthesis there is no guarantee that it will happen or the other way also can happen.

So, this actually comes with more and more practice and experience. No one can really teach you that how to write a code. So, that there will be no error there will be no timing error. So, how to write good test benches this all come out of practice and experience. This is what i have mentioned in the last point. This cannot be taught. This requires a lot of practice and experience. This will automatically come with time as you design more and more.

So, with this way you come to the end of this lecture. So, over the last 3 lectures we discussed some examples, using which you saw how a complex design can be partitioned in to data path and control path. And how we can write verilog specifications for them separately and then integrate them together.

Thank you.