Digital Systems Design with PLDs and FPGAs

Kuruvilla Varghese
Department of Electronic Systems Engineering
Indian Institute of Science-Bangalore

Lecture-43
Case Study 2

So, welcome to this lecture on the course digital system design with PLDs and FPGA around last 42 lectures I have covered the digital design the PLDs, FPGAs and VHDL to a great extent what is the required for synthesis okay. Today we are going to discuss a reasonably complex case study okay, what we can discuss in a class room, because we have limitation on the slide what we diagram.

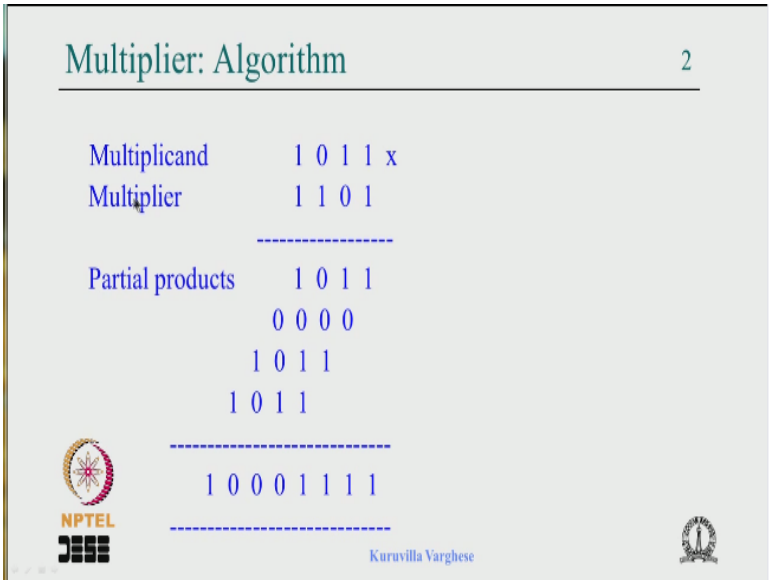
We can show on the slide all that but this case study captures in a net shell everything that requires for a complex project it is not that it is not useful if you thoroughly understand the process which will bring together almost all the aspects, we have discuss. Now some will be evident when we put this when we use a tool which will be the next lecture.

I will show as I said I will kind of decouple the tool complete it towards the end part so, that the can the because tools change, and I do not want that to come into any of kind of lectures which will deal with concept okay. So, it will be completed FPGA part, VHDL part all that it will be completed. When we discuss the case study. Today I will show you the design how to go about designing it.

And how to write the VHDL code also only thing is that, maybe when we implement that using the tool some of the FPGA are related how much of the FPGA is use, what is how the FPGAs slices are wired up or something like that, which we can see insight the device when we use the tool. So, today we are going to discuss a case study which is nothing but a very simple 8 bit multiplier okay.

But I will go to every bit of detail with regard to case study that we apply what all we have learn what are possible can be crammed into this case study it is done. So, let us look at this particular design example okay. So, let us move on to multiplier.

(Refer Slide Time: 03:07)



So, we are trying to implement an unsigned integer multiplier, this can changed into a sign multiplier by you know 1 easy way of doing that is by doing the booth encoding, you know radics for booth encoding you can do. And I mean you can build on this particular case study say you can do a radics for booth encoder , which will support the sign multiplication you can modify this to do the division, restoring division, non restoring division and so on.

You know lot of things can be done as an extension of this case study okay but let us start with the algorithm, and naive way of implementing it. Then whether, we can optimise it and so on okay. So, the and for the sake of clarity I am showing a 4 bit multiplication.-But the real design we have going to do an 8 bit multiplication okay. So, we have a multiplicand which is 4 bit, multiplier 4 bit okay.

And you know how the multiplication is done on a paper. So, what we do is that, we take the lease significant bit of the multiplier if it is 1, we form a partial product, that is in the 2 rise to 0 kind of place okay. Then the 2 rise to 1 position the multiplier bit it look it is 0. So, we write the

partial product corresponding partial product that is nothing but the multiplicand kind of anded mass by this bit okay. So, that is all 00.

And that has to be shifted to the left, because it is 2 rise to 1 position, then 2 rise to 2 position partial product is written like that, 2 rise to 3 is written like that okay. Then we add all the partial product together, to get product okay. So, that is a algorithm, and like this is a pencil paper method. And that is a basic algorithm, there is no difference it is a basic algorithm is same.

So, in this case if you try to implement this as it is written in the paper. Then we will need a register to store a multiplicand, the register to store a multiplier, then we need a say 8 bit in this case, a result register, and we need to add all these and you know that these 2 partial products can be added using an adder. Then that result and this can be added with another adder, and ultimately one more adder. So, it requires 3 adders to do this parallely okay.

So, and in the case of 8 bit multiplier algorithm we need in a paper, pencil method, we need 7 adders okay. And that is a very costly operation to have a 7 adder. So, you can imagine about 16 bit multiplication. That will require 15 adders each that will be 15 bit okay, or even kind of extended. Because it need to need more than, you know.

Because here you see there it is shifted, so instead of that, there are 5 bit and then you need a 5 bit adder and so on okay. So, we do not want to waste adders. So, we will try to use a single adder okay. So, the idea is that we use this as an accumulator. So, make at the beginning everything 0 okay. So, then what you do is that first partial product is added here, the second partial product is added shift and so on okay.

Now there is a better method of doing it okay. Instead of shifting the partial product to the left, we can shift the accumulator to the right okay. So, the idea is that make the accumulator 0 at the beginning. Then take the first partial product add it at the msp part here. And after adding we just shift it okay by 1 bit okay. Because then this part is shifted okay like that. Now the second partial product is added at msp, because we write shift accumulator.

That left is it is equal to left shifting the partial product. So, we add it here, because it is convenient, because we can add it at the same place every time okay. And then you shift by 1 and so on. So, the next partial bit, partial product you add it here, shift then this 1 you add it here shift okay. Then you get an 8 bit result okay, now another optimisation this not that very kind of important.

But, this add to little bit complexity, and which is nice okay. So, what we can do is that at the beginning this part is not used at all okay. So, first time you add a partial product here, and shift by 1 bit then this bit is occupied. The next time you add and shift 2 bits are occupied. So, what we can do is that we need not waste another register for multiplier. At the beginning we can make this part 0 and store the multiplier here okay.

Then look at the least significant bit of the result to check the current multiplier bit to check. If it is 1 add the multiplicand here. Then you shift, so that bit is gone, that bit no more required 0 comes here. So, you look at this bit, current bit if its 0 then what we do is that we can add 0 here. But, there is another way of doing it. So, we take this part just pump it back to this register okay.

So, just put it back and shift okay. So, instead of adding 0, we can just re-circulate the msp back to here and put it okay. Our idea is that everything goes on in the iteration. We do not want anything to be start like when you start iterating, you add either the multiplicand here or you re-circulate Moore significant by. So, that it done say in this case 4 times okay.

In an 8 bit we have to do that 8 times okay. So, by storing the multiplier we get an advantage that we do not need an additional storage, and also we do not need to kind of look you know kind of you know move to find the current bit you know since it is getting shifted we need to look at result 0^{th} bit of result to find, what is the current bit of the multiplier okay.

So, that is 1 major change we are going to do. We are going to use a single adder we use an accumulator, and we have to add the partial product at the most significant bits, half of bits here and shift if right okay. And we are going to add 0, whenever we want to add 0, we just state this

part and put it back. So, instead of adding 0, so that is when you add 0 the result is same. So, we take it and put it back okay.

That is second kind of variation. Now there is another variation what we have going to is that, there is no point in adding this bit here because to add we need this part to the adder. And this multiplicand to the adder, you add together. But then there is no point in result of the adder to be loaded here and then shift, it is a waste of time. Because to load we need a clock and shift we need another clock okay.

So, it is a 2 clock per iteration. So, what we do is that we need this the highest bits to the adder, multiplicand to the adder. Then we add together, but when we load we load it shifted okay by 1 bit, because if you add two 4 bits you get a 5 bit. And that fifth bit comes here fourth here, third here, second here and the first bit here and everything get shifted okay. So, I hope you get the algorithm now .

We use an iterative multiplier okay, that means we 1 by 1 it is waiting we accumulate the partial product and we store the multiplier in the lease significant bits, and we add the multiplier multiplicated along with this you know this partial product at this place and shift and we do not load and shift. We will you know we will load shifted okay. So, that we do everything in a single clock cycle okay.

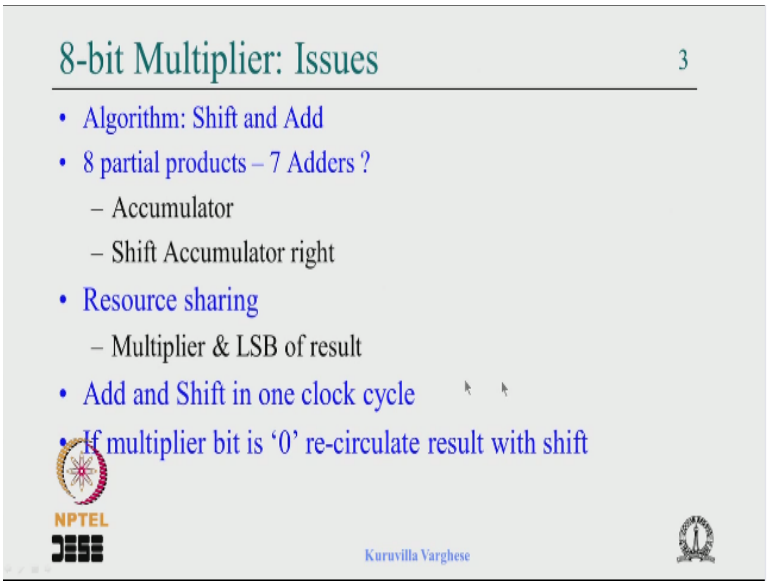
So, that is the algorithm, the modified algorithm. So, we need now a multiplicand register in the case of 8 bit, 8 bit register. We need an 8 bit adder okay which gives a 9 bit result then we need a 16 bit register okay. So, that much is required, and now to re-circulate we need a mugs okay. Because we either have to choose the adder output or current the most significant bits back.

So, we need a multiplexer and there are 2 other things which is required, which is not maybe obvious okay I told you that this process of loop has to be done in the case of 4 bit, 4 times okay. And in an 8 bit multiplier you have to do this process of you know adding or re-circulating 8 times. So, we need to keep track of the iteration, and so, we need a counter which count from 0 to 7 okay.

So, that we get to know that when to stop okay. So, a counter is required in addition to this data path elements okay. Data path elements are the multiplicand register, result register which contains a multiplier register, you know multiplier and adder, now we need a counter to keep track of the iteration much more important. So, we talked about the data path, but we need a controller to control the data path okay maybe there is a start signal from the external world.

And that will start the whole operation. And when it is done it will indicate that it is done. That can be sampled by a processor or it can be an entrapped to the processor or entrap to the rest of the circuit whatever it is. So, we need a controller okay. So, I hope you are clear the basic of multiplication, the algorithm, the modification to algorithm the elements what are the resources in terms of sequential elements. And the combinational circuit what you required all that is clear que. So, let us move on and so.

(Refer Slide Time: 15:30)



This is what we have discuss we our algorithm is shift in the add. So, basic algorithm is shifting the partial product to the left. But we shift the accumulator to the right. We need 8 partial products, but and not 7 adders, we are going to use an accumulator and single adder with shifting the accumulator right. And in the result the lease significant byte we are going to store the multiplier, and we are not going to load the adder output into the accumulator in 1 clock cycle and shift in another clock cycle.

We are going to do that in a single clock cycle okay just 1 clock cycle. Now if the multiplier bit is 0 we will as I said we are not going to add 0. But we re-circulate the result with a shift oaky. So, **so** that is algorithm.

(Refer Slide Time: 16:32)

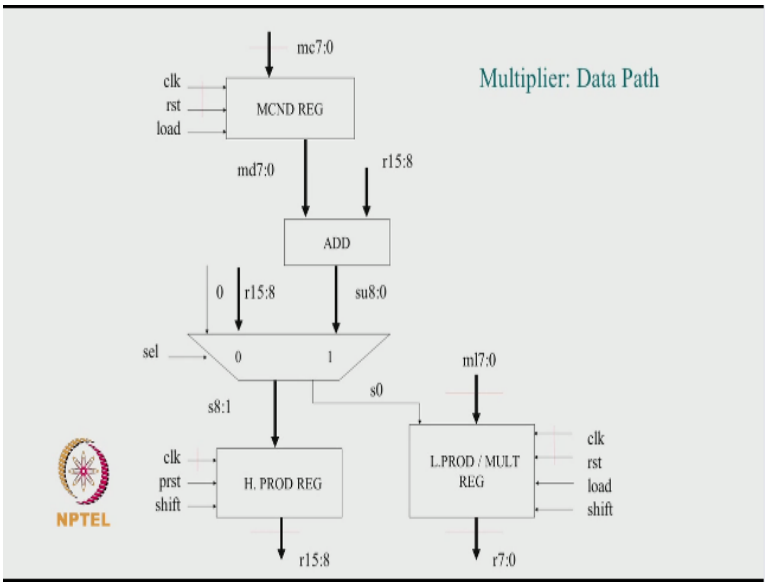
Now, the resources required is that, we need a multiplicand register 8 bit because we are going to implement the 8 bit multiplier. We need a result register which is 16 bit, lease significant by it of that is multiplier. Then we need an 9 bit adder, because we add two 8 bits and produce the 9 bit. We need a 9 bit 2 to 1 multiplexer to select the adder output or most significant byte of the result to re-circulate.

We need a bit counter which counts from 0 to 7. So, we require 3 bit counter to keep track of the iteration, we need a controller which control the whole data path operation okay. So, it is quite detail I am writing it down. So, maybe when you design it is better to write resources required. So, that there in absolutely no lack of clarity when you get to the data path and the controller design okay.

So, let us look at the data path at the beginning in a little abstract way the top level okay. The top level is simple block with multiplier, a like multiplicand, multiplier and result. And some control

start to that, but I am going to show you the next level, level 1 partition of the data path okay not the controller data path.

(Refer Slide Time: 18:05)



So, this is the whole thing which is represented from a top to bottom data flow. So, look at this part. This is the multiplicand register, it gets an input mc 7 down to 0 multiplicand is mc, that is only to hold the value like at the beginning of the iteration we load the value here. Then the output of this register is used for multiplication. So, the output is called md which is 7 down to 0.

Now you look at control signal a power on reset called rst come to it. Because at the power on this register is reset, and once it reset it is enough. Because every time a multiplication happens we loading a new value it not be every time at the power on we need to reset. So, that it does not get into metastability at the power on. Then we need a clock, and we have seen how controller can control with a clock not touched okay.

We have seen that the controller control signal will come directly to the data path enable some muges, and all that okay. Now the control signal which is required is called load. Because when the controller say load the input value it will be load this mc **77** 7 down to 0 to md 7 down to 0 okay. So, only one control signal load, and we will take this and expand in the subsequent slide okay. Now let us look at the result register which is a 16 bit register which is r 15 down to 0.

The higher product register is a accumulator, so it has a clock. Now it has a reset, now mind you this not the power on reset this I am calling a p reset or peripheral reset whatever. And that comes from the controller okay. So, that is a signal which is output of the controller, because every new value you load to the multiplicand, and multiplier this has to be clear for this has to be make 0. So, the controller will clear at the beginning of the multiplication this particular register okay.

Now we know that this has to be loaded with the adder output or the re-circulating, and that need a shift signal it can be called load also. Because we are just loading the value shifted. But since same control signal is use for shifting these, this register we just use the word shift okay. So, the same control signal when it is active it gets loaded the shifted value gets loaded here whatever is you know the value s0 along with this is shifted and loaded okay.

So, in 1 clock cycle everything happens in 1 clock cycle. But this if you look at the least significant byte of this register, we you know that at the beginning the multiplier is loaded here okay. So, it need a load signal which will load this ml 7 down to 0 to r 7 down to 0 okay. Now we are going to look at the r(0) to add accumulator most significant by with the multiplicand or to re-circulate this part back here okay.

Now this a 8 bit, 9 bit adder which add the most significant byte of this accumulator which is r 15 down to 8 with the multiplicand. And you get a 9 bit, and this mux depending on this bit 1 of 0 will select. If it is 1 it will select the adder output you see that it is a 9 bit output and the adder output is called su 8 down to 0, the multiplexer is called s8 down to 0. Now you see 8 to 1 is loaded here which is already shifted okay.

Because is not that we are loading s7 here and shifting with s8, we are loading s8 to r 15, s1 to r8 and s0 to r7 and whatever was there in r7 will be loaded back to r6, r6 will be loaded back to r5 . and r0 will just go out okay. So, when whatever we do either we whether it is adder output or re-circulator output shifted okay in the clock cycle. And if r(0) is 0 then you see this r 15, 8 which is shifted version is getting loaded here you know it is shifted with the 0 at the most significant bit**.**

Because every operation we need a shifting, and that is what this is representing and the s0 goes down here. So, it does not matter as for as the iteration is concerned this way or this way. So, initially we load the algorithm is like that when the start signal come, controller will give this load signal and the multiplicand and multiply get loaded. And now the controller will look at the r(0), r(0) is goes to the controller.

So, if r(0) is 1, the controller will give this select line as 1. Then this gets added it gets load and if it is 0 this get re-circulated and controller will do that 8 time okay. So, we need a counter, the output of which will go to the controller okay. And the controller will look at the r(0) and you know select this line. Similarly the load signal is given by the controller both all the 3 loads signal, all the 2, and all the 2 shift signal is given by the controller.

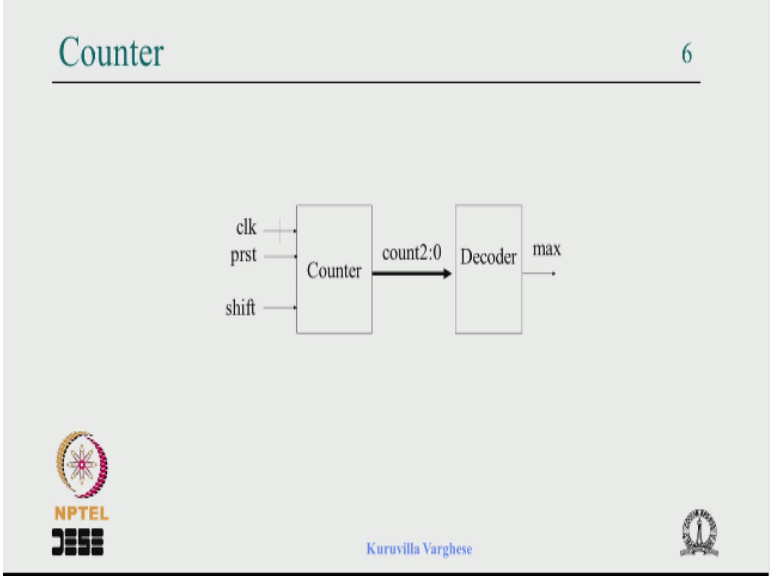
Select signal is given by the controller and this prst is given by the controller okay. But this reset is a power on reset at the beginning, now you see that I have put some red line dotted line on some signal what I do is that I am indicating that these are the external signal okay. So, these are the course, clock reset mc, ml which coming from outside and r 15 and r 8 are the results okay. So, why I am putting a dotted line is that when you write the VHDL code we can declare al the red line.

The cross line signal as port and rest of them as internal signal so, looking at the picture I will say entity, multiplier I say code, clock, reset is in standard logic. Then select is a like select is a signal. So, r 15 down to 0 is a out standard logic vector 15 down to 0 and so on okay. So, and mc is standard logic vector 7 down to 0. So, it is very easy and rest all signal I can declare a signals okay internal signal, that is why I put this dotted line.

So, what we have going to do is that we have going to complete the data path, it is not complete yet, we need a counter. So, the counter you know that has to keep track of the iteration. So, the counter can use shift as an enable okay. So, when the shift is 1, counter will increment. Otherwise counter will not increment okay. And this reset the p reset when accumulator is made 0, the counter also need to be reset. Because every iteration new data value, the counter need to be reset okay.

So, we give the counter reset not the power on reset, if you do the power on reset it may not be right. So, we use the p reset for the counter, so let us see the counter block diagram.

(Refer Slide Time: 26:13)

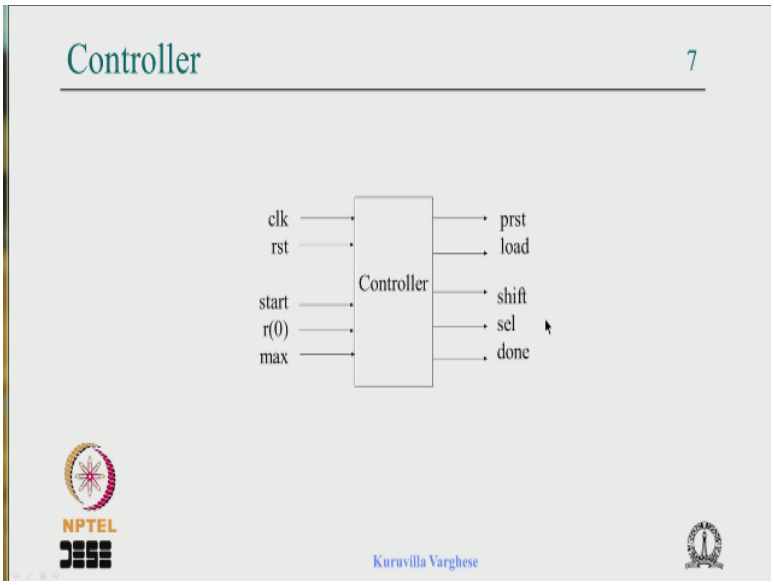


So, the counter as I am showing it as a single block, we will expand it later okay, we will see the level 2 diagram of the counter we gets clock which is an external signal, p reset is which is coming from controller, shift which is a enable, whenever the shift is 1 it will count increment okay. So, when shift is 1 it will increment. So, that is given by the controller and it is a 3 bit counter.

But now controller should know when it has reach 7. So, it is a decoder which is equal to 7 oaky. It is an AND with 111 okay like Q2 is 1, q1 is 1, Q0 is 1, this signal is high and the controller knows and this goes as input to the controller okay. And controller knows then it has to stop the iteration okay. So, what comes out of the controller is the select line, the load line the p rest line and the shift line okay.

And r(0) goes as an input to the controller. Then depending on that it will generate the select line and also this max from decoder value of the counter goes as an input to the controller okay. Now we can draw the controller block diagram.

(Refer Slide Time: 27:41)

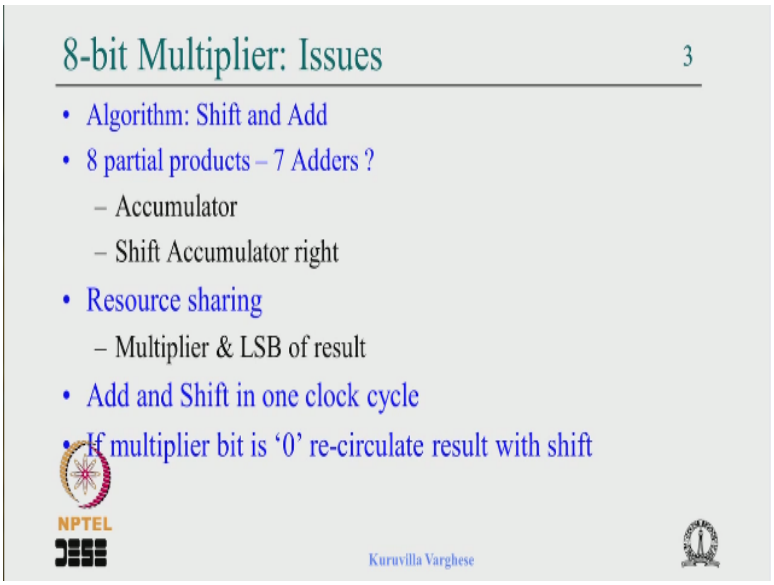


So, let us put the controller which anyway get the clock and reset. Reset is required because that has to come to the first initial state and clock is a 1 being the controller being synchronous with the clock. And r(0) is lease significant bit of result which is also the current bit of the multiplier okay. That is why it is taken into the controller. The max signal is decoded value of the counter. That will tell the controller to stop all business.

And, the start is an external signal which is coming to the controller, which say start the multiplication now okay. So, the external world is giving the multiplicand and the multiplier, and will give a start signal and then the controller will start loading, multiplying and all that. And all these are the control signal p reset which goes to the counter and the accumulator load which is going to multiplicand and multiplier.

Shift is going to the multiplier and the accumulator, select is to select the mugs to choose the adder output or the re-circulating one. And the done is to indicate that the multiplication is complete okay. So, that is what is the data path and the controller okay. So, looking.

(Refer Slide Time: 29:12)



So, we started with algorithm okay and we modify the algorithm, optimise the algorithm.

(Refer Slide Time: 29:14)

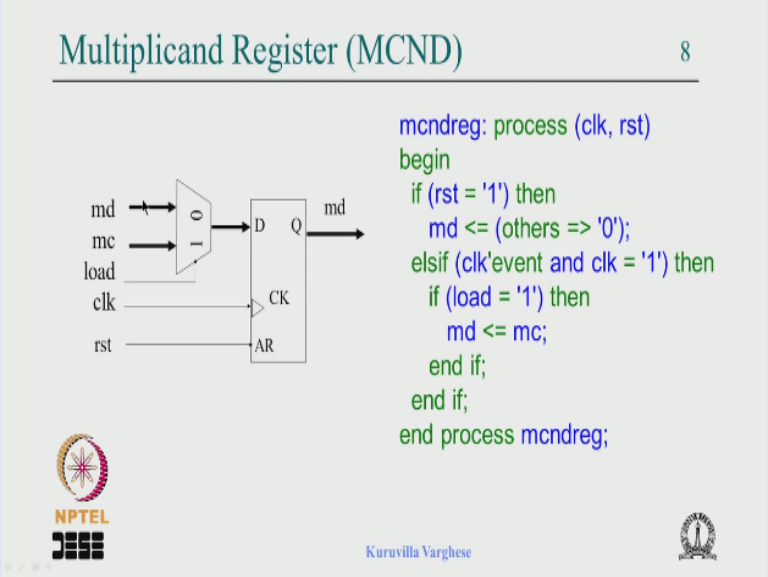
We found what are the resources required, then we started with the data path, registers and adders and multiplexers and as I said these block need further expansion okay. We will do that and a counter is required to keep track of the iteration, and a controller which control the data path and the counter**.** Counter we can think of it as a part of the data path. Now what we are going to do is that we have going to expand it.

We have going to take this register expand it this 1 we will expand, this 1 we will expand and this one, this counter we will expand, and then what we do is that we will write the state diagram

for this controller okay. Because we are clear in algorithm. Now we have told the algorithm clearly. I am not written it down, but then with that description we will go to this state machine okay.

So, let us take the multiplicand register , that is this register what is required is that, when the load signal comes upon the clock this gets loaded, and then it is remembered, it is re-circulated. So, that is what is shown here.

(Refer Slide Time: 30:33)

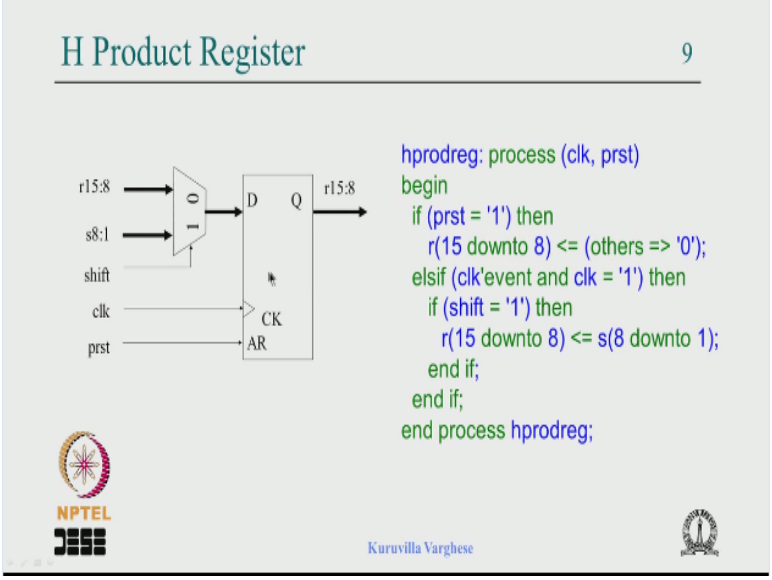


We have an 8 bit register which get the power on reset, and the clock upon the reset this is made 0, when the load signal from controller comes, the input, multiplicand, mc goes to md. And if it is 0 this multiplicand is re-circulated okay. I am not showing a path like that, but that you imagine there is a connection like that okay. Once again upon the reset multiplicand is made 0, the output is made 0 upon the clock if load is 1 the input gets loaded.

Otherwise it gets remembered okay. So, how do we write the VHDL code, so we write a process clock comma reset both are like reset is asynchronous. So, we put it in the sensitivity less begin, if reset is 1, then this output md is others 0. Because it is 8 bit, else if clock event clock is equal to 1, if load is 1, then md get mc end if, end if means else it is re-circulated okay. So, that is how the code for this is written very simple you know.

If you do this approach everything is simple, and when you synthesise you will get what you have written okay, there no need to complicate. So, let us pickup this particular register now high product register it is almost same. You see here what we have going to do pre reset is 1, this is made 0, if upon the clock if shift is 1 what is done is that s8:1 gets loaded here, that is all, otherwise it will re-circulate okay, so, very simple.

(Refer Slide Time: 32:30)



So, we have the higher product register okay or the accumulated part and the p reset is the reset which will reset r 15 down to 8 to 0. Upon the clock if shift is 1, that multiplexer output s8:1 get loaded into r 15:8, if not that is re-circulated it is remembered okay. So, that only when this shift signal is come, this get kind of you know loaded okay. And that is the VHDL code for it.

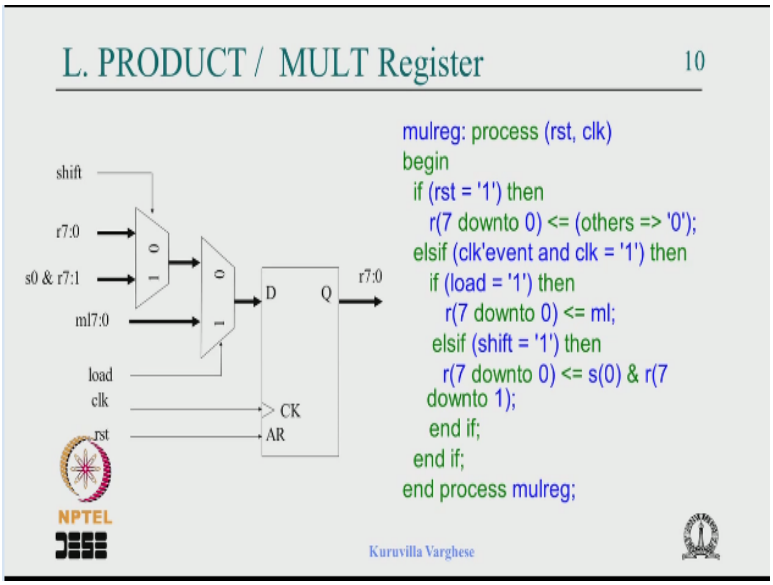
If p reset is 1, then r 15 down to 8 is others 0, else if clock event clock is equal to 1, and if shift is 1, then r 15 down to 8 gets s8:1 down to 1 okay. So, r 15down to 8 gets s8 down to 1 end if. That means if it is shift is not 1, shift is 0, this is re-circulated okay. So, see how simple is the coding it is very straight forward, if you draw the diagram after some practice maybe you do not need to do this you can kind of imagine in your head.

Like you know that what is that, this is that register and you can write the code. But at the beginning I suggest that you go through this little bit TDS process, so that everything is clear

even in even if you are an expert drawing such an blog diagram will help a lot there will be less errors, it will much more optimise than orbit rally writing the code okay.

So, let us speak up this particular register which is little most complex register we have. So, it has 2 control signal. So, upon the reset r 7 down to 0 is 0. Now the load as priority, if load is 1 then r 7 down to 0 gets ml 7 down to 0, if not else if, if shift is 1 s0 get into msp, r 7 get into 6 bit, r 6 get into fifth bit and so on okay. So, it gets right shift okay, so that is what is we are going to see detail blog diagram.

(Refer Slide Time: 34:59)

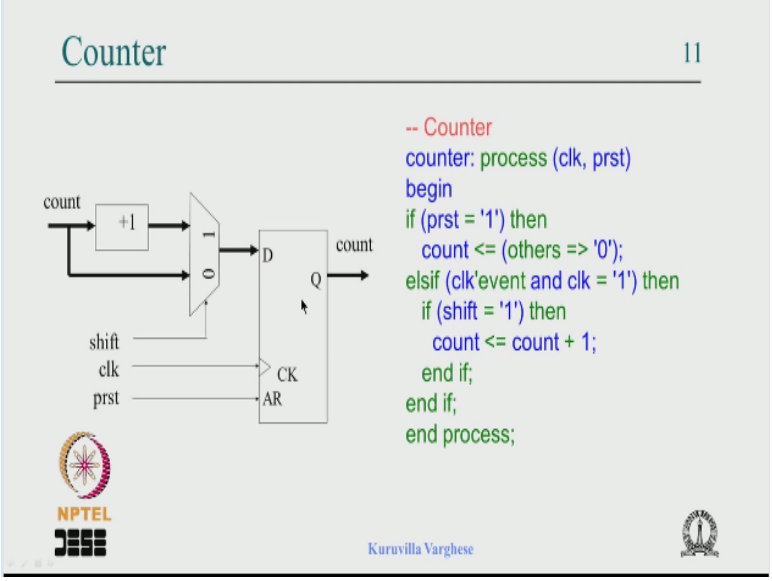


So, that is shown here, so there are 2 control signal with load as a priority, so you see 2 muxes. So, upon the reset r 7 down to 0 is 0, if not upon the when the load is 1 which has priority, the multiplier input gets to the multiplier output okay, and if not if load is 0 then if shift 1 okay. Then you see that s0 is a most significant by bit, r 7 is an x and up to 1 and r 0 goes off okay, if not the r7 gets you know re-circulated okay.

So, that is multiplier register or the lower product term the detail blog diagram. Now let us look at VHDL code we write a process the clock and reset in the sensitivity less, if reset is 1 then this is 0 r 7 down to 1 assigned others 0. Else if clock event clock is equal to 1, load is 1, r 7 down to 0 is ml okay which is 7 down to 0 else if . Shift is 1, then r 7 down to 0 is s0 and r 7 down to 1 and end if okay.

So, that means if not then it is re-circulated okay, end if, end process. So, very simple you know the code way the code is written same as all the register only the control signal comes in the proper priority, and these diagram definitely help it brings in clarity, and if you write a code like this from the blog diagram, when you synthesise you will get this back you know exactly similar. And we need the last the counter we have to show the detail view of the counter, so that is where we have coming.

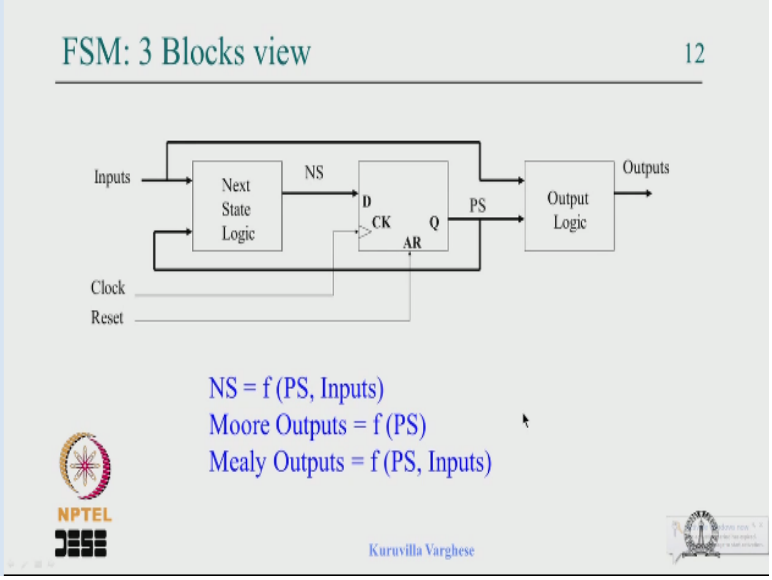
(Refer Slide Time: 37:05)



So, if the counter is that it is a 3 bit counter, 3 flip-flops, p reset because it need to be reset at the beginning of a an iteration and the clock. And you see when the shift is 0, then the count is re-circulated when the shift is 1, the count is incremented and loaded okay. So, that is what we need when the shift is 1, counter gets incremented. So, the code is if p reset, process, clock and p reset, p reset is 1 count is others 0, else if clock event clock is equal to 1.

If shift is 1, then count is count+1, end if okay, if shift is 1 then the count is count+1 otherwise it re-circulates okay. So, that is how what the detail blog diagram of the various elements of the data path. Now what is remaining to be seen is the state machine okay.

(Refer Slide Time: 38:10)

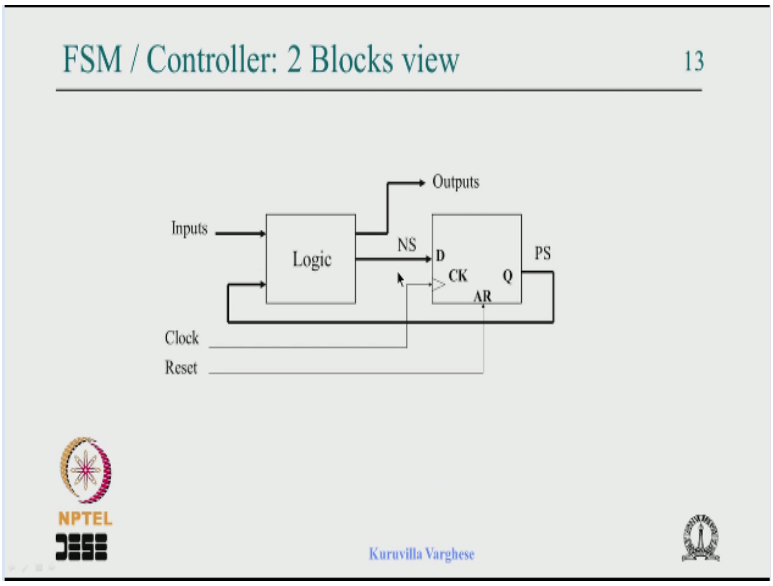


So, we know the controller or the state machine as this view the state flip-flops, next state logic which is looking at the present state and inputs and output logic. And I have shown earlier the picture of the controller which gets clock and reset, start r(0) and max are the inputs and PRST, load, shift, select, done are the outputs okay. So, that is what we have going to see now. We have going to see the state diagram okay.

Because by from the state diagram we can arrive at the next state logic and the output logic we have seen that. But when we write the code, we just write the code for the state machine, and the synthesis tool will find the equation for the next state logic and the output logic okay. And we have you know that the Moore output when the output is a function of the present state, Mealy output when the output is the function of the present state and the input.

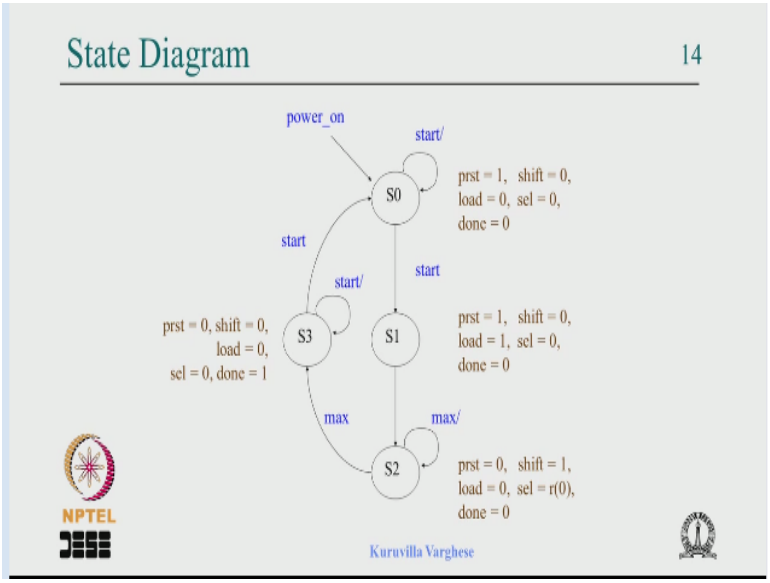
And we have said we can have 2 blog view where the output logic and in the next state gets the inputs similar inputs. So, we can combine these 2 blocks called logic and take the output from here okay.

(Refer Slide Time: 39:27)



So, that is a 2 block view, you have the logic which is comprising of the next state logic and the output logic which gives the next state and the output. So, we are going to use this view we have going to code and the output together okay, it is very convenient, because looking at the state diagram we can write the VHDL code in that case. So, we will use this block for implementing the state machine or the controller okay.

(Refer Slide Time: 39:58)



So, this is the state diagram of the algorithm you have discuss. Look at the state diagram at the power on we come to a starting state okay. Now at this point the state machine is waiting for the start signal to come. If the from external world maybe it is a processor which is giving the signal as long as the start is low remain in this state, and we have to initialise all the signal properly.

Normally the reset is 1, because the counter is reset, the accumulator reset all the control signals take the inactive value.

So, the select can be 1 or 0 it does not matter, but the shift load all that are control signal, that has to be 0, the done is 0, because we have not done yet okay. So, when the start comes from the external world, we assume that the multiplicand and multiplier input is given. So, what we do is when the start comes we keep the reset high, because we do not want the counters to start now.

Because still we are not started the iteration, so reset is 1 and the load is 1 okay. There is no issue when if the reset is removed, because the counter is controlled by the shift. Unless shift is 1, counter would not be incrementing. But it is okay to have the reset 1, the load is made 1. So, you know that at this state this s1 state the multiplicand and the multiplier will get loaded upon the clock into the multiplicand register and the result register.

Now we can start the multiply, multiplication process we come to this state and we make shift as 1. So, now you can see many things happening. So, what we what happens is that the counter will start counting, and this select line at each bit will reflect the current value of r(0). So, either this will happen or this will happen like that you know depending on like maybe this is 3 time, this is 1 time whatever depending on the bit pattern that will happen.

So, that is this place, but the state machine has to make the shift one. So, that the registers are loaded properly this particular register and this register gets loaded, the counter gets incremented and all that. So, the shift is made 1 and the select line has to be chosen properly, if it is 1 if this bit 1, then you have to choose this path 0 this path. So, the select line is nothing but the r(0) okay.

And the done bit is 0 okay. We have not done yet. And this state has to continue till the counter decoded value is 7 okay, when the 7 comes next clock cycle it can exit, because, we have done 8 iteration the counter started with 0. So, in the next clock cycle it is incremented to 1, so when **re** we reach 8, it is a 8 iteration at the end of it, you can get out of that state. So, that is as long as the max is low remain here, if the max is high come to this state.

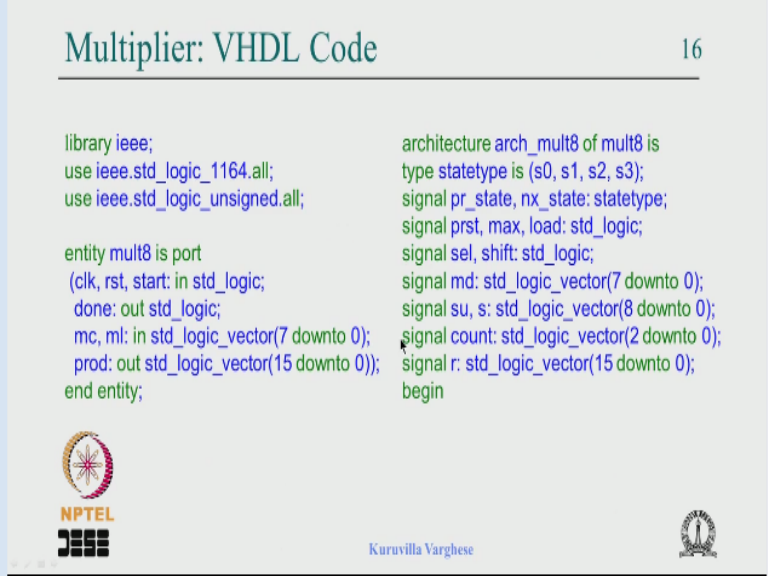
Now we should make the load 0, shift 0 the select does not matter. But we will say the result is there already, because all the iteration has finish, and the done is made 1, this can be sample by the rest of the circuit or by a processor. Now what we have going to do is that as long as the start is low remain here, the new start comes, and we assume start is a small pulse**.**

Because if start is very of long duration when it comes even if it completes still the start maybe high and it will restart again okay. So, we assume that start is a pulse okay. So, that is the state diagram at the beginning as I said reset upon the start come to a state load the values, then you start the iteration by making shift is 1, select is r(0) wait for max signal, when the max signal comes make everything inactive and make that done bit 1 okay.

So, in like as for as the design is concern, we have completed the design. Because this can be coded directly in VHDL we have done the data path design at the level 1 showing all the blocks. We have taken each block, and implemented that in a detail and we have seen the corresponding VHDL code. We had the algorithm at the beginning and we have come out the state diagram for to implement that algorithm.

So, now by writing the VHDL code we can get everything together. So, we are now going to look at the VHDL code for this particular multiplier okay. So, let us move to that.

(Refer Slide Time: 45:13)



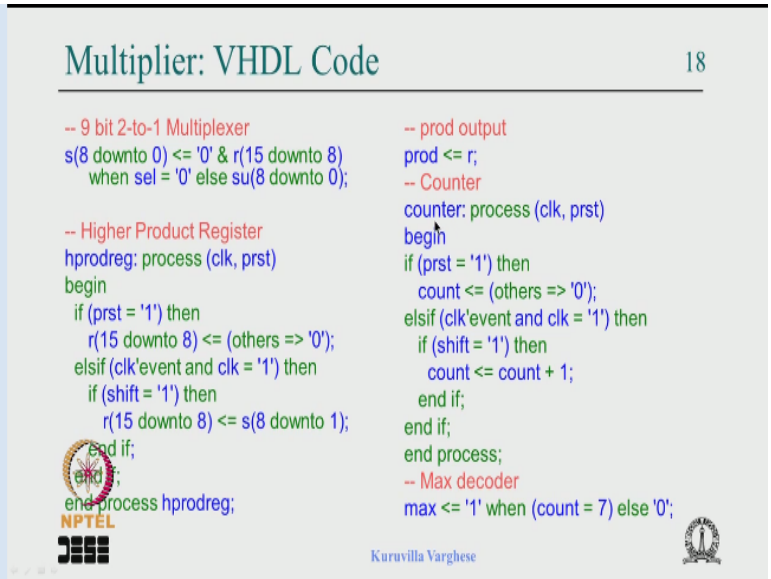
So, here this is the VHDL code, we have the library declaration, standard logic 1164 where the standard logic is define, standard logic unsigned, because somewhere we are going to use +1 for the counter, as I said wherever there is dash okay and red dotted line, we declare is as the port. So, clock, reset, start is in standard logic, done is out standard logic, mc, ml is 8 bit vector, product is 16 bit vector okay.

Now we can declare all the other signals, you know as with appropriate width which is shown here as internal signal okay. So, we need since we have a state machine, which is consisting of 4 states. We need a state type declaration. So, that is shown here, type state type is s0, s1, s2, s3. Signal present state next state is of this type, and I have mention this in the lecture where we discuss the VHDL coding of the state machine.

So, most tools allow you to kind of define an enumerated state, and declare the present state and the next state using that. And that will be treated as a standard logic vector and all the other signal like p reset, max, load select shift all that has standard logic. The output of the registers they are of appropriate width, the md is 8 bit multiplicand, su, is the output of the adder 9 bit, s is output of the multiplexer 9 bit, count is 3 bit counter, r is the real result which will be assigned to this particular product.

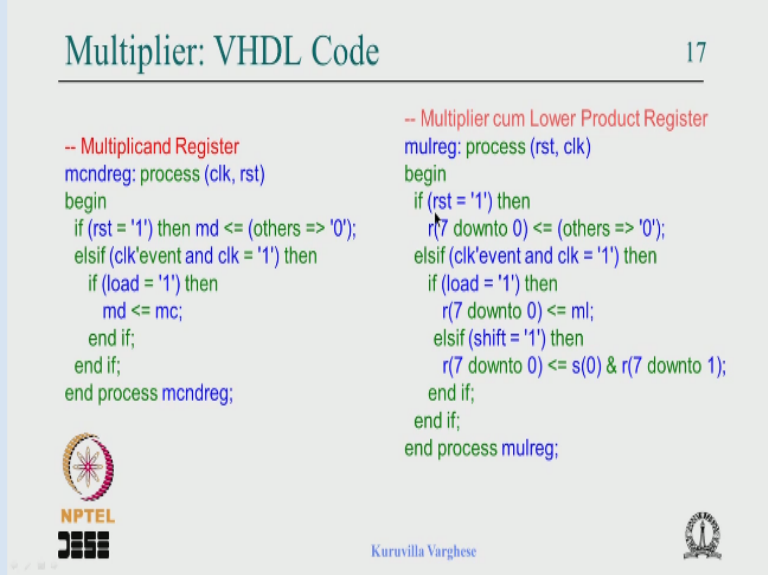
Because we are going to re-circulate r, so if you use product it can be re-circulated, because being an output it cannot come on the right hand side of assignment. That is why we use another signal call r.

(Refer Slide Time: 47:22)



Now you see at the begin what we do is somewhere we assign r to the product okay.

(Refer Slide Time: 47:28)



Now I am showing all the VHDL code the multiplicand register we have discuss this. So, I give a label, process, clock, reset then begin if reset is 1 md is others 0, else if clock event clock is equal to 1, if load is md gets mc. There is only one control signal which is load. Now this is multiplier cum lower product register which has 2 control signal load and shift. So, process, clock and reset.

If reset is 1, r 7 down to 0 is others 0, so we are talking about this okay. And if load is 1, r 7 down to 0 0get the multiplier input, if shift is 1 r 7 down to 0 get s0, and the shifted version of r 7 down

to 0, that is r 7 down to 1. So, we are talking about this load is 1 does gets in here, if shift is 1, then s0 with r 7 down to 1, which is a shifted version gets load it here. So, that is a code which is showing this.

The next is a multiplexer, because this we need a multiplexer which is nothing but if select is 1 this s8 down to 0 is su 8 down to 0, else it is 0 and r 15 down to 8. So, that is that s8 down to 0 is 0 and r15 down to 8 when select is 0, else it is su 8 down to 0. So, that is exactly 1 concurrent statement. So, it is very simple and higher product register, so we are talking about this.

So, if shift is 1 this gets loaded, that is all. So, that is if p reset and clock in the sensitivity less if p reset is 1, r 15 down to 8 is 0, else if clock event clock is equal to 1, if shift is 1, r 15 down to 8 is s8 sown to 1 okay. Now **,** so that is completion of the higher product register as I said the r is internal signal which is assigned to the external product, and we have the counter which we discuss before, process, clock, p reset.

If p reset is 1, then count is others 0, else if clock event clock is equal to 1, if shift is 1, count is count +1, end if. That means otherwise it re-circulate. We need a decoder, so which is a concurrent statement we do not write it internally and confuse. If you write it in the wrong place it will get a flip-flop, like if you write this statement within this else if clock event clock is equal to 1, there will be a flip-flop coming there.

And it will add 1 bit latency and it can create timing issue unless you address it. So, we write is as a concurrent statement max is 1, when count is 7 else 0 okay. So, it just indicates when the counter reaches 7 okay.

(Refer Slide Time: 50:26)

Multiplier: VHDL Code

19

```
-- Adder
su <= ('0' & md) + ('0' & r(15 downto 8));

-- FSM, Next state Logic, Output Logic
connsl: process (pr_state, start, r(0), max)
begin
case pr_state is
  when s0 =>
  prst <= '1'; load <= '0'; shift <= '0';
  sel <= '0'; done <= '0';
  if (start = '1') then nx_state <= s1;
  else nx_state <= s0;
  end if;
when s1 =>
  prst <= '1'; load <= '1'; shift <= '0';
  sel <= '0'; done <= '0';
  nx_state <= s2;
when s2 =>
  prst <= '0'; load <= '0'; shift <= '1';
  sel <= r(0); done <= '0';
  if (max = '1') then nx_state <= s3;
  else nx_state <= s2;
  end if;
```

NPTEL

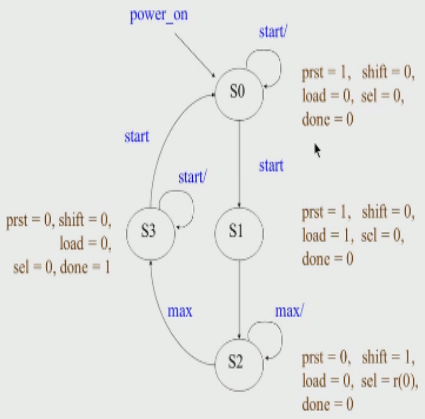
Kuruvilla Varghese

Now this is the adder, particular adder we use a +operator. So, this adder we have using a +operator to have optimise implementation in FPGA, we have discuss if you +, it implement using the look up table and the carry change. So, 0 because a 9 bit adder 0 upended with a md +0, upended r 15 down to 8. So, that we get 9 bit result okay. Now what have going to so, is that we have to implement the state diagram.

(Refer Slide Time: 50:58)

State Diagram

21



NPTEL

Kuruvilla Varghese

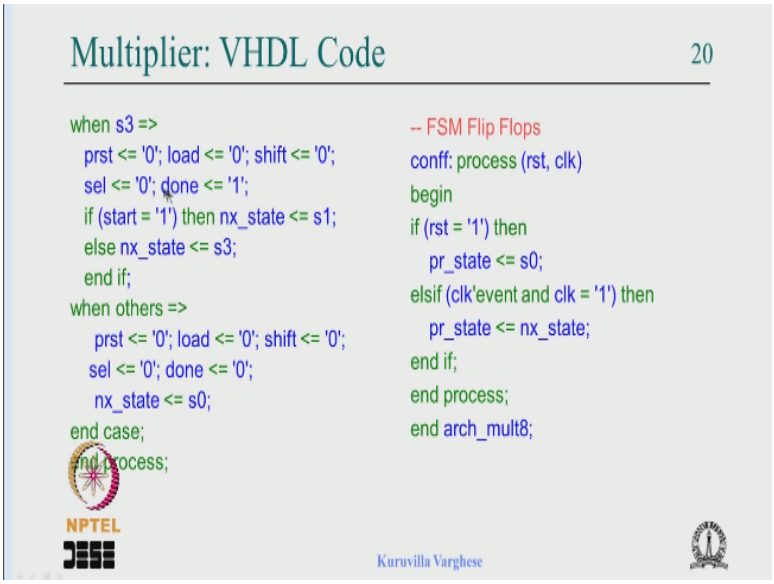
And we have going to combine the next state logic, and output logic together okay. So, idea is that we keep this in front and write the VHDL code for the next state logic and the output logic. So, it is very simple straight forward, I hope you remember this. We will go to logic part of it.

So, a process, the present state is input the process, all the inputs also like you have start r(0) max.

These are the input to the state machine. So, that has to be in the sensitivity less then we say case present is when s0 like in the state s0. We because all the outputs are Moore outputs. So, we just write the output, p reset is 1rest all are 0. Then we write the transition if start is 1, then the next state is s1, else you remain there okay so, that is it. If Start is 1 then go here, else you remain there.

And this is the value which is written here. So, when it come to when s1 like reset is 1 load is 1. And there is no condition you know unconditionally it is transiting to the next state. So, the next state is s2, in the s2 you see the shift is 1, select is ro. But as long as max is 1 go to the next state s3 else remind there. You know so, that is what is shown here, if the max is 1 go here else remind there okay.

(Refer Slide Time: 52:40)

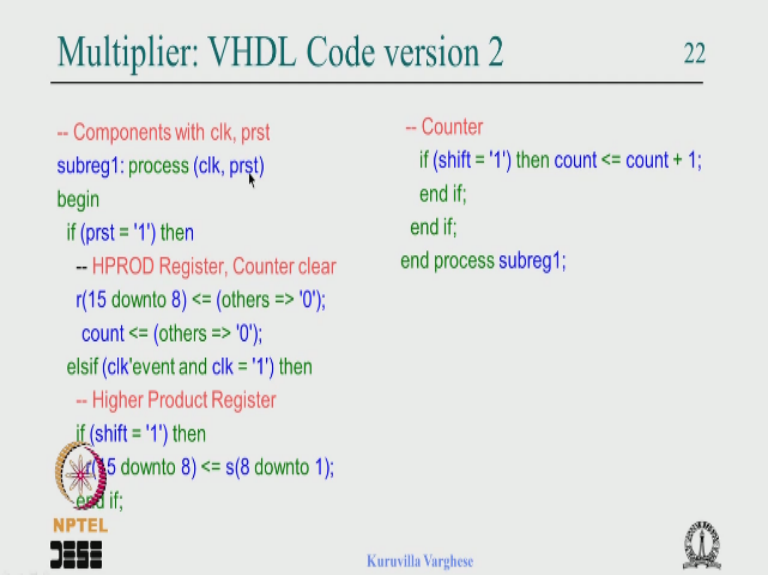


Now, the last state so, when s3 the done is 1, rest all is 0, and if start is 1 then go to next the state s1 else next state is s3. Now I made a slight modification the diagram shows that it goes to s0 but problem with s0 is that may be okay it is like it kind of repeated. So, you do not need to go to this state you can come here, because the reset is made 1 here. So, we can say 1 clock cycle so, it just if the start is 1.

You can come straight away here okay, now we need to specify the flip-flop registers this particular register which is nothing but upon the reset. The present state is made the first state otherwise upon the clock, the present state get the next state okay. So that is what is written here so, the flip-flop process clock reset clock begin if reset is 1, present state is s0, starting state, else if clock event clock is equal to 1.

Present state get next state end if end process end the architecture so, complete design is over only thing is that maybe you can like, we have shown say take this 2 registers you see that the it is working with the same clock, working with the same reset. If you want to combine them into a single process you can do that okay.

(Refer Slide Time: 54:22)

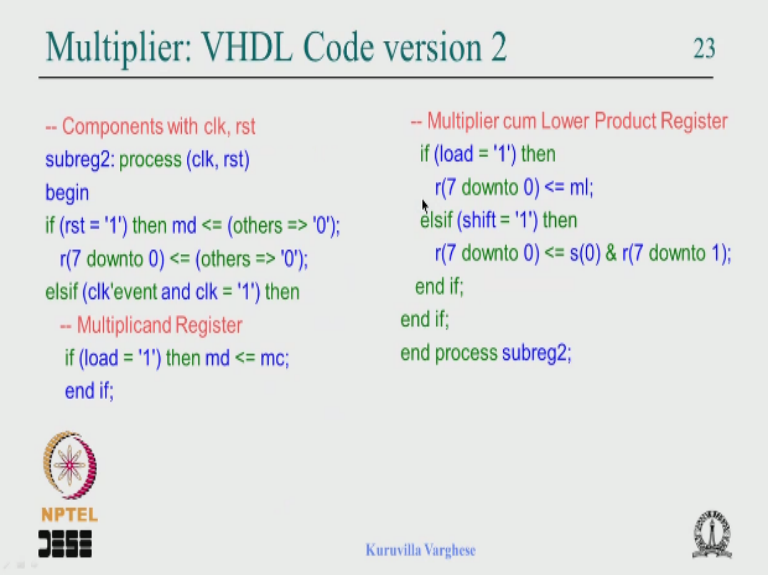


So, I have shown that say like, we have a single process with clock and p reset. If p reset is 1 or 15 down to 8 is 0, count is 0, else if clock event clock is equal to 1, if shift is 1 r 15 down to 8 is s8 down to 1. And you can maybe include the count also here, count is count +1, end if or you can write a separate if shift is 1 for counter. But this can be combine similarly you look at this particular code like the multiplicand register.

And the multiplier register can be combine, but when you combine we cannot see here it uses only, a single load signal here it is load and shift there. So, we cannot combine in the same if

statement there has to be separate if statement if you combine that so, that is is shown here upon the reset r15 down to 8 is 0, count is 0.

(Refer Slide Time: 55:23)



Now this is the earlier one, so this is multiplicand and multiplier upon the reset multiplicand is 0, result is r7 down to 0 then we write the multiplicand register with 1, if and multiplier come low product register with another if okay. So, it is little bit concise way of coding it, but you should not mix up this process with this process. Because the reset is different so, you should not mix it so, that is how the VHDL code is written.

So, I will in the next lecture I will show this in a board with tool and all that. But what we have done today is looking at multiplier case study. We have looked at the algorithm, we have optimise the algorithm we have found the resources. We have from the algorithm we have put down the data path at the top level block diagram partition. Then 1 by 1 we have taken each piece design in detail.

You know up to a register and multiplexer we have written the VHDL code from the controller algorithm you have develop the state diagram. Then from the state diagram and these second level blocks, we have written the VHDL code. You know for the data path and for the controller, now implementation is over you know I am very like if you write with this much detail nothing can go wrong.

You know very minute mistakes will be there in one short the design will work. There will be only a minor errors if you do the design like this so, I suggest that you go through this process, do not ship anything as for as possible write things down in detail. And try to write the VHDL code, so in the next lecture which is any time kind of concluding I will show you the tool with some simple combinational exercise.

Sequential exercise and this case study on a with the tool, on a particular board a FPGA board so, that you get complete picture of course. So, I suggest you try some variation of this design as I said divider or the radic for booth encoder multiplier and all that which will enhance your learning. So, you can try that exercises so, please revise I wish you all the best thank you.