

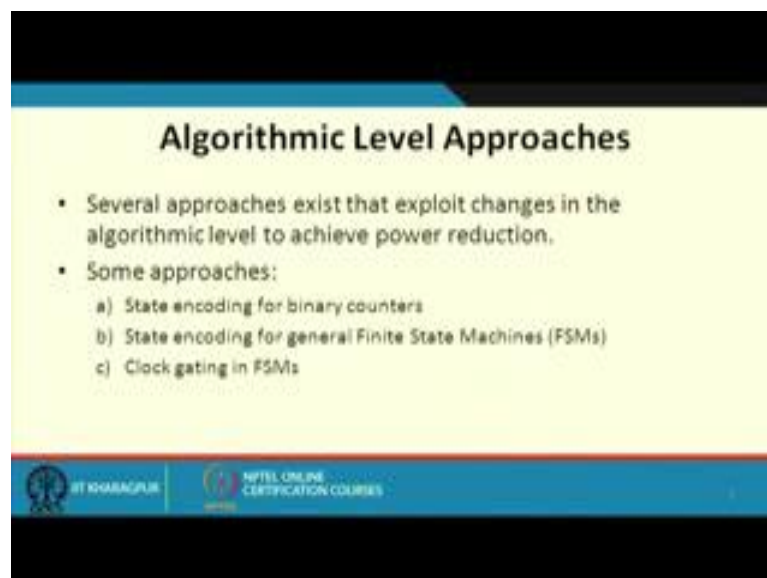
VLSI Physical Design
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 63
Algorithmic Level Techniques for Low Power Design

So, in this lecture, we shall be looking at some of the algorithmic techniques that you can use to reduce the power dissipation in a design. So, when you say algorithmic technique it means that we have possibly not yet carried out the synthesis. We are looking at the design from either a functional point of view or from a behavioral point of view. We are yet to carry out the design. If and at that level we can use a number of intuitive techniques, which if you follow is expected to give us a final circuit later on, that will be good in terms of power dissipation.

Many of these methods that we will be discussing they actually come out of design experience. So, experience designers know that if I change my input specification in this way, my final circuit will be better in terms of power consumption and some other characteristics also. So, we look at some of these algorithmic level techniques here.

(Refer Slide Time: 01:31)



Algorithmic Level Approaches

- Several approaches exist that exploit changes in the algorithmic level to achieve power reduction.
- Some approaches:
 - a) State encoding for binary counters
 - b) State encoding for general Finite State Machines (FSMs)
 - c) Clock gating in FSMs

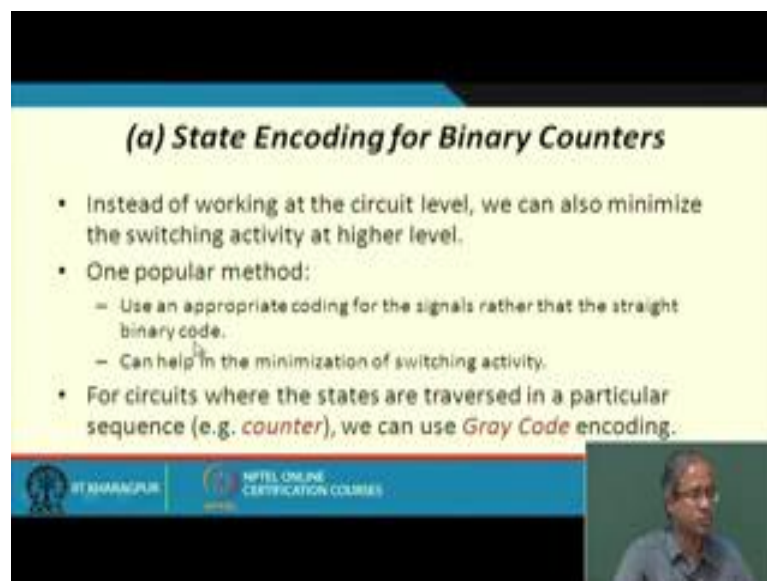
IIIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, as I had said that these techniques try to exploit some changes at the algorithmic level. So, we have not yet made the design. Again these methods may not be very general, they may not be applicable to all designs, but for many of the designs which

appear quite commonly, you can use them. Like for example, we will be illustrating these approaches with respect to design of a binary counter how we can carry out state encoding in an efficient way for low power then how we can carry out state encoding for finite state machines and clock gating in FSMs. You see clock gating in FSMs is something very interesting.

So, for whatever clock gating methods we talked about and looked at, we said that well I have a circuit of functional block. I tried to find out whether or not that block is being used. If it is not being used I switch off the clock. I use a gate for that, but now I am saying that well I have not designed my circuit I do not know what my circuits are I have only my finite state machines specification, may be a state transition diagram. Just by looking at the state transition diagram, I can predict that well these are the places where I want to shut off my clock. So, that unnecessary state transitions are avoided this is the idea.

(Refer Slide Time: 03:11)



(a) State Encoding for Binary Counters

- Instead of working at the circuit level, we can also minimize the switching activity at higher level.
- One popular method:
 - Use an appropriate coding for the signals rather than the straight binary code.
 - Can help in the minimization of switching activity.
- For circuits where the states are traversed in a particular sequence (e.g. *counter*), we can use *Gray Code* encoding.

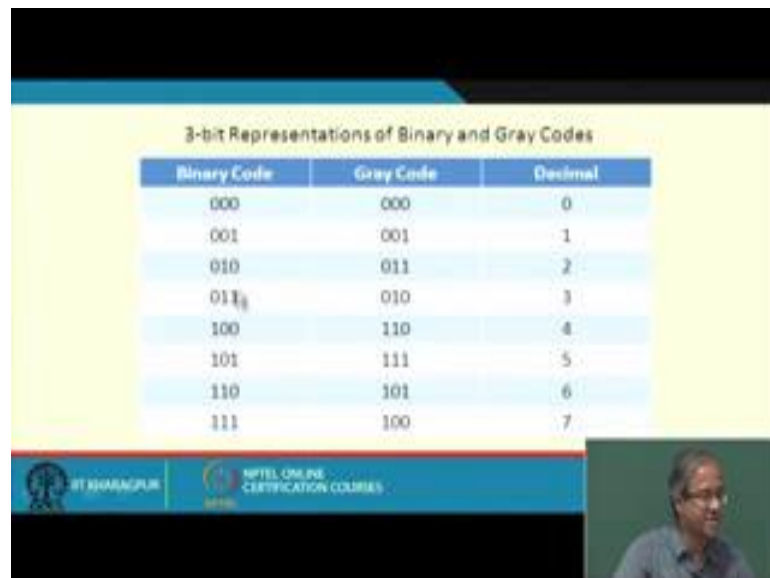
NPTEL ONLINE CERTIFICATION COURSES

So, let us start with state encoding for binary counters. So, here as I had said we are not working at the circuit level. So, we are looking at the design specification at a much higher level. And by using a suitable state encoding, we are also able to minimize the switching activity even at this higher level right. So, what is the approach we are looking at? You see normally when we design a counter, whatever we have learnt in our course earlier whatever is available in the text book.

Firstly, they talk about binary counters a counter. That counts in a binary sequence 1 2 3 4 where the count values are generated in binary, so there are many such design methodologies which have been thought to you possibly in some courses they are available in test books. Now these design methodologies, they try to optimize the circuit in terms of either the number of gates or the delay and so on. But often they do not look into the power dissipation issue. That what kind of design can give you less number of signal transitions once you design the counter. Well I have said we have not yet designed the counter. We are still looking at the conceptual level or at the specification level. So, what we are saying is that, that instead of using a straight binary code we use an appropriate coding for the signals, if we can do that this can help in the minimization of switching activity.

You see normally the counters that we use in many applications, they count numbers in a particular sequence 0 1 2 3 4 like that. For these kinds of applications, we can use an encoding like something called gray code encoding. This can give you or give us some very good properties with respect to reducing signal transitions.

(Refer Slide Time: 05:39)



Binary Code	Gray Code	Decimal
000	000	0
001	001	1
010	011	2
011	010	3
100	110	4
101	111	5
110	101	6
111	100	7

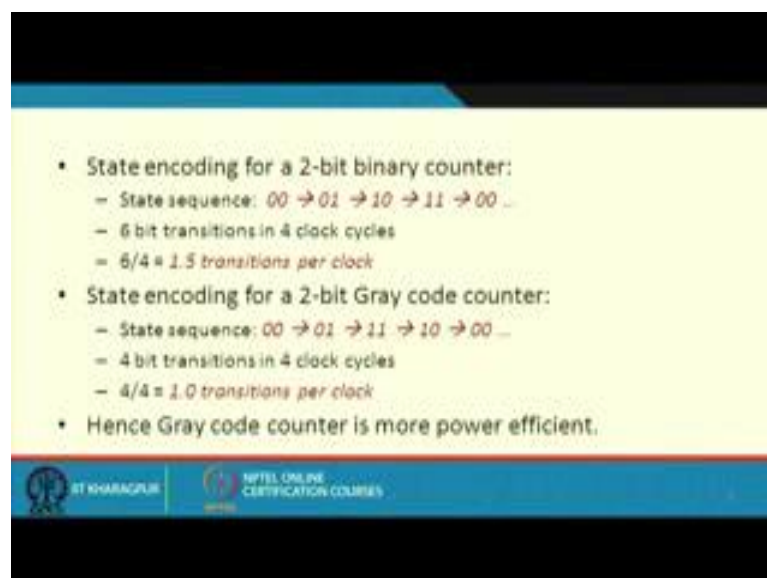
Now, let us understand what is gray code. In this table we are showing as an illustration 3-bit representation of an of the count values in a counter, binary counter. See the decimal numbers in a 3-bit counter will range from 0 to 7. So, the counter will count

from 0 1 2 4 up to 7 then again back to 0. So, if it is binary counter, the count values will come like this binary 0 1 2 3 4 like this.

Now, if we look at this count value successive count values, see now we are somewhat power aware we are thinking in the back of our head, that well let us see how many transitions are taking place. Let us see from the count 0 to 1, there is one transition. From 1 to 2 there are 2 transitions in the last 2 bits, from 2 to 3 there is one transition. From 3 to 4 it is pretty bad all 3 bits are changing, 4 to 5 1 transition, 5 to 6 2 transitions, 6 to 7 1, and again from 7 to 0 all 3 transitions. So, here as you can see minimum we have one transition. Maximum we can have all the bits changing. So, in the alternate grey code encoding, we are encoding the count values in such a way that across successive counts only one bit is changing at a time. Right that is the property of grey code. So, you see the grey code of the decimal numbers are defined like this 0 0 0 0 1 0 1 1 0 1 0 like this.

You see, exactly one bit is changing from one count to the next, 1 0 0 back to 0 0 0 right. This is the advantage of grey code. So, now, intuitively if we start at the very beginning by designing a counter for grey code, that is expected to give us a circuit that will be more efficient. Naturally number of signal transitions will be less in terms of power dissipation right.

(Refer Slide Time: 07:58)



The slide contains the following text:

- State encoding for a 2-bit binary counter:
 - State sequence: 00 → 01 → 10 → 11 → 00 ...
 - 6 bit transitions in 4 clock cycles
 - $6/4 = 1.5$ transitions per clock
- State encoding for a 2-bit Gray code counter:
 - State sequence: 00 → 01 → 11 → 10 → 00 ...
 - 4 bit transitions in 4 clock cycles
 - $4/4 = 1.0$ transitions per clock
- Hence Gray code counter is more power efficient.

At the bottom of the slide, there are logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.

So, just a few observations for a 2-bit binary counter, the count values will be from 0 0 0 1 1 0 1 1 and back to 0 0. So, in 4 clock cycles, there are 6 transitions like from 0 0 0 1

there are one transition. 1 to 2 there are 2 transitions, this to this there is 1 and 1 1 to 0 0 another 2. So, 1 2 1 2 total 6. So, in 4 clock cycles you have an average of one in 1.5 transitions per clock, but if you have a 2-bit gray code counter, the count values will go like this, 0 0 0 1 1 1 and 1 0 back to 0 0. So, for every transition one bit is changing. So, 4 transitions in 4 clocks which means one transitions per clock. So, in terms of the average you can see that grey code counter is 50 percent more efficient in terms of number of transitions per clock. So, when you design a circuit expectedly grey code counter will be more efficient in terms of power.

(Refer Slide Time: 09:18)

2-bit binary counter design with original encoding.

Present state	Next state		
a	b	A	B
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

$A = a'b + ab'$
 $B = a'b' + ab$

The slide also features a logic circuit diagram with two D flip-flops, two AND gates, and two OR gates. The inputs are labeled 'a' and 'b', and the outputs are 'A' and 'B'. The clock input is 'CK' and the clear input is 'CLR'. A small video inset in the bottom right corner shows a man speaking.

So, I am just shown the design of this counters. This is a simple 2-bit counter with normal binary encoding. So, the count values are going like that 0 0 0 1 1 0 1 1. The next states will be this. So, here I am not done minimization you can do minimization also. Because here this a bar b or a b bar is alright, but b bar this is a bar b bar or a b bar if you take b bar common it will be a bar or a or it will be only b bar. So, this can be minimized, but this has been kept non means non minimized because of balancing the delays from inputs to outputs. Sometimes where low power design we do that. But here the number of transitions are as I said will be about 6, 6 transitions in 4 clock side like this.

(Refer Slide Time: 10:21)

2-bit binary counter design with Gray code encoding.

Present state	Next state		
a	b	A	B
0	0	0	1
0	1	1	1
1	0	0	0
1	1	1	0

$A = a\bar{b} + ab$
 $B = a\bar{b}' + ab$




Now, for grey code the state table will change. For gray code the changes are 0 0 0 1 1 0 and 1 1. So, my functions will be like this $a\bar{b}$ or $a\bar{b}$. So, if you minimize it will be only $b\bar{a}$ or $b\bar{a}$ or $a\bar{b}$. If we minimize this will be only $a\bar{b}$, but if we want to keep it like this or balancing the delays, you can keep it also like this.


So, the idea is if you want to design a grey coat counter, this is expected to result in much less number of transitions, because across counts exactly one of the inputs will be changing. So, the other inputs will not be triggering any transitions. This is the advantage.

(Refer Slide Time: 11:08)

Comparison for 3-bit counter.
75% more toggles in binary counter.

Binary		Gray-code	
State	No. of toggles	State	No. of toggles
000	0	000	0
001	1	001	1
010	2	011	1
011	1	010	1
100	3	110	1
101	1	111	1
110	2	101	1
111	1	100	1
000	3	000	1






So, just for a 3 bit counter I am making a comparison for binary across states. So, how many transitions are taking place that I have just noted down 0 to 1 1 1 to 2. Similarly, 0 7 2 0 3, so on the average, if you take the average transitions per clock comes to 1.75, but for grey code counter for every transition, there is one toggle. So, average transitions per toggle is only one right. This is one interesting observation 75 percent more toggle here.


(Refer Slide Time: 11:50)

Toggles for N-bit counter design:

- Number of toggles for binary counter $T_{bin} = 2(2^n - 1)$
- Number of toggles for Gray counter $T_{gray} = 2^n$
- Thus $T_{gray} / T_{bin} = 2^n / (2(2^n - 1)) \rightarrow 0.5$

Bits	T(binary)	T(gray)	T(gray)/T(binary)
1	2	2	1.0
2	6	4	0.6667
3	14	8	0.5714
4	30	16	0.5333
5	62	32	0.5161
6	126	64	0.5079
∞	-	-	0.5000

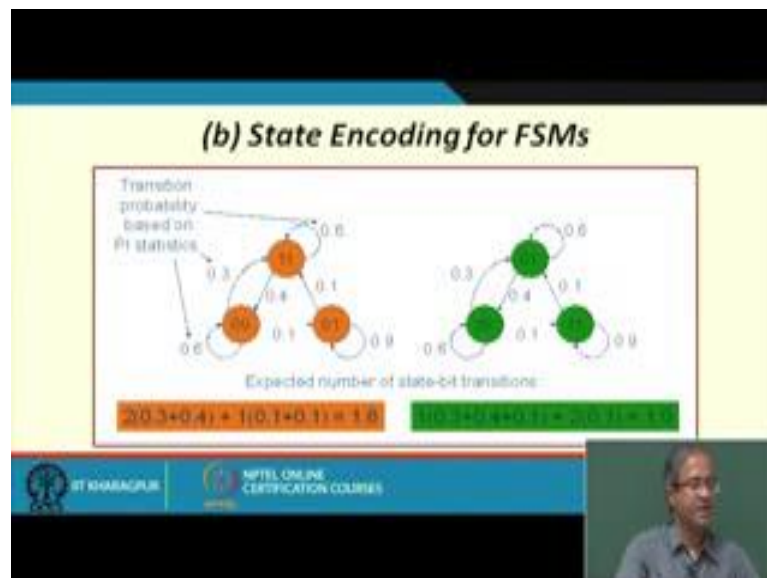


Now, in general terms, if you look at an n bit counter design, for a binary counter if you make a simple calculation, the number of toggles can be calculated as 2 into 2 to the

power n minus 1. So, for a simple case you see already you have seen, for 3 bit counter it is how much 3 4 6 7 10 14 14. So, it is 2^2 into 2 to the power 3 minus 1, 7 into 2 14. So, this simple this expression gives the number of toggle.

But for a grey code counter every transition there is only one, because there are 2 to the power n total transition the total toggles will also be 2 to the power n . So, if you take the ratio of T_{grey} divided by T_{binary} this approximates to 0.5 as n becomes large. So, it is 50 percent less transitions. So, this table shows this. So, as the number of bits increases this ratio approaches 0.5. So, for a 6 bit counter it is 0.5079. As it tends to infinity this will be exactly 0.5. So, this is one of the very commonly used designed techniques for a counter to reduce power by using grey code encoding.

(Refer Slide Time: 13:21)



Now, let us come to finite state machines. You see whenever we design a controller for any kind of circuit, with the first thing we do is that we draw a state transition diagram this is what you want. And from this state transition diagram we try to synthesize or design my circuit. So, let us take an example.

Suppose I have a state transition like this. There are 3 states and these edges, this is an arrow here this edges are marked with some signal probabilities 0.3. This indicates the probability of transitions. I am assuming that when I have designed my FSM. I have already done some simulation analysis based on some real life kind of data and found out how many or what is the chance of each of the transitions means it turns are occurring.

So, I have some probability values I am assuming these are available with me right. So, which are most commonly taken transitions? So, these numbers are shown here 0.6, 0.1. Say from this state there is 60 percent probability of remaining in this state and 40 percent of probability of coming to here. From this state 60 percent here 30 percent here and 10 percent, this arrow is not shown. There is an arrow here. There is an arrow here.

So when you are designed, you do some state encoding in the first step. Means each of the states you encode by a binary number, let us say I give 0 0 1 1 0 1. Now if you calculate the weighted sum of this signal probabilities and the sum of the bit changes across transitions, like you see 2 between 1 1 and 0 0 2 bits change. And what is the total probability one is 0.3 other is 0.4, these 2 edges are there between the so 0.3 plus 0.4. And one is between 1 1 and 0 1 1 bit change and between 0 0 and 0 1 1 bit changing. So, the total probability here is 0.1 here and 0.1 here 0.1 plus 0.1. So, the total weighted sum is 1.6 suppose I change my signal encoding I make this 0 1 make this 0 0 make this 1 1. So, what I have actually done. The states between which most of the transitions take place, I have encoded them to be having a single bit transition 0 0 and 0 1, but here it is 2. So, now, one is 0.3 plus 0.4 plus 0.1 these are one into this and 2 is between this and this 0.1.

So, now I see is the average is becoming so much less 1.0. So, so even at the level of FSM I have not yet designed by circuit. So, I can use my state encoding in such a way such that the weighted sum of the signal transitions gets minimized. If I do this it is expected that when I finally, come to the designing of the circuit, it will result in a circuit which will be creating less number of signal transitions. These are all architectural or algorithmic level you can say or higher level design techniques.

(Refer Slide Time: 17:11)

(c) Clock Gating in FSMs

- In a Moore machine, the output(s) depend only on the state variables.
- If a state has a self-loop in the state transition diagram, then clock can be stopped whenever the self-loop is to be executed.

Clock can be stopped when (X_k, Z_k) combination occurs.

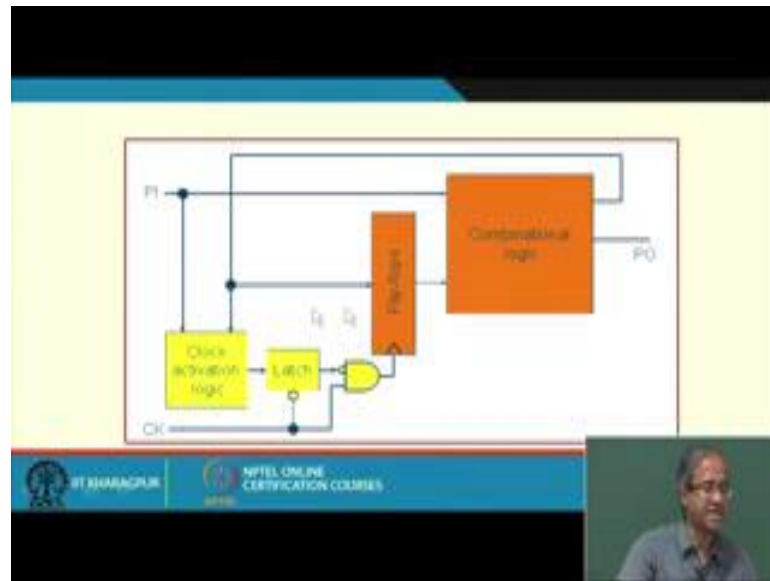
Again let us come to clock gating in the level of FSM. We'll consider a finite state machine; you know that there are 2 kinds of machines Moore machine and Mealy machine. So, in a Moore machine the output depends only on the state variables, but in contrast in the Mealy machine the output depends not only on the state, but also on the present input, but let us concentrate on the Moore machine here.

So, Moore machine the next state depends only on this state where you are. So, it is not dependent on the input. So, the output sorry, the output depends on the state, but the input transition may depend on the input. So, let us say. So, I have a small FSM where I am showing 3 states. So, from X_i to Z_k we make a transition, when the input is X_i and the output is Z_k . From X_j to X_k , I make a transition when the input is X_j output is something. And I remain in state X_k when the input is X_k and the output is Z_k .

Now what I am saying is that when the inputs are X_i and X_j , then some signal transition across states are happening, but when the input is X_k the state remains in X_k . So, the observation is that when I am in state X_k , and my input is X_k , why do not I stop my clock because nothing is changing here. I am in X_k and my input is X_k I remain there. So, I can stop the clock during these periods.

So, whenever in these FSMs there is a self-loop like this. We can stop the clock. So, at the FSM level we can take some decisions like this, when we can do clock gating to stop the clock.

(Refer Slide Time: 19:20)



So, in terms of a circuit diagram it can look like this. So, how we do it? So, this is my clock activation logic. So, my flip flop contains this state. So, this states you can feed to the inputs and also the primary input depending on their combination X_k and X_k , you can decide whether to enable or disable the clock. So, you can store something in the latch that latch will decide. So, if the latch is one then there is inversion here it will be 0 clock will be disabled if the latch is 0 the clock will be enabled.

So, this is a high level decision you are taking. So, by analyzing a finite state machine the well when I design my finite state machine, I will be adding a circuit like this. So, I will be checking for this state if this state is there then I will stop the clock. So, this is a high level design technique which if I use I will be stopping some unnecessary transitions in cases where the state does not change in FSM.

(Refer Slide Time: 20:34)

RTL Level Design (Signal Gating)

```
module decoder (a, sel);
input [1:0] a;
output [3:0] sel;
reg [3:0] sel;
always @(a) begin
case (a)
2'b00: sel = 4'b0001;
2'b01: sel = 4'b0010;
2'b10: sel = 4'b0100;
2'b11: sel = 4'b1000;
endcase
end
endmodule
```

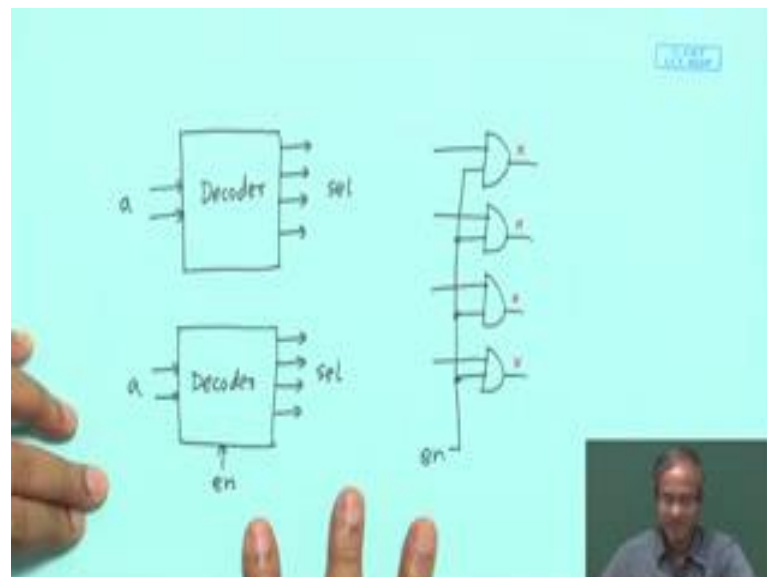
Decoder with enable

```
module decoder (en, a, sel);
input en;
input [1:0] a;
output [3:0] sel;
reg [3:0] sel;
always @(en, a) begin
case (en, a)
1'b0, 0: sel = 4'b0000;
1'b0, 1: sel = 4'b0001;
1'b1, 0: sel = 4'b0010;
1'b1, 1: sel = 4'b0100;
1'b1, 1: sel = 4'b1000;
default: sel = 4'b0000;
endcase
end
endmodule
```

IT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Now, here you look at an even higher level of a design. Well we have not seen any example of a high level design description language, but I am just giving you the giving you the idea. Here I am showing very log descriptions of a simple decoder - A 2 to 4 decoder. So, I am just showing you what this decoder actually does.

(Refer Slide Time: 21:02)



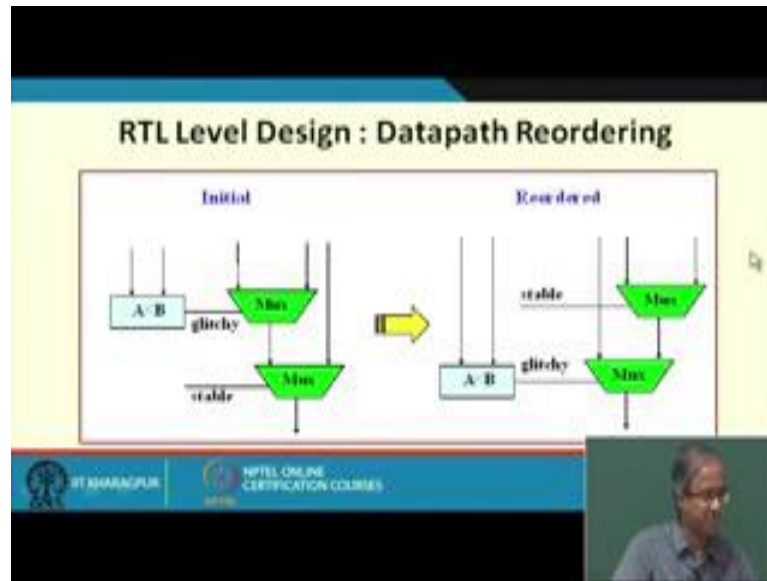
So, 2 alternate designs are shown. This is a decoder circuit. So, I am assuming that that 2 inputs are called an AND there are 4 outputs I call them sel. Depending on the value of a so one of the outputs will be selected as one, the rest will be zeroes. This is one design

and I also an alternate design; this is also the same decoder. Here also I have my input, here also I have my select outputs, but in addition I have another input called enable. So, if enable is activated, only then the outputs will change. So, if I disable it then the outputs will not change. So, which means you see the outputs are finally, taken out from some gates right. So, there will be some gates. So, this enable this input can be feeding the second input of this and gate let us say.

So, the other input will be coming from the logic inside the decoder. So, if I am not enabling the output then, what will happen? There will be no signal transition occurring in the output of these gates right. So, only when I require to enable the decoder only then this will be enabled and the transitions will take place. The basic idea is like this. So, even at the highest level when you specify the design using very log, in the first description I am defining the module with inputs an and cell, no enable. This is an enable less decoder. In the second design I have also used an enable input. So, here I am saying whenever a changes you do this, here I am saying means whenever enable and a both changes then do this. So, this enable is also an input which activates the operation right.

So, this example actually shows that even when I am working at a much higher level, I am writing a very log or vhdl code. There itself I can be a little bit aware of where I am heading to. I am trying to create a design that is expected to consume less power. So, let me add this enable input to my decoder. So, when not required I shall not be enabling it that will; obviously, consume less power this is the idea.

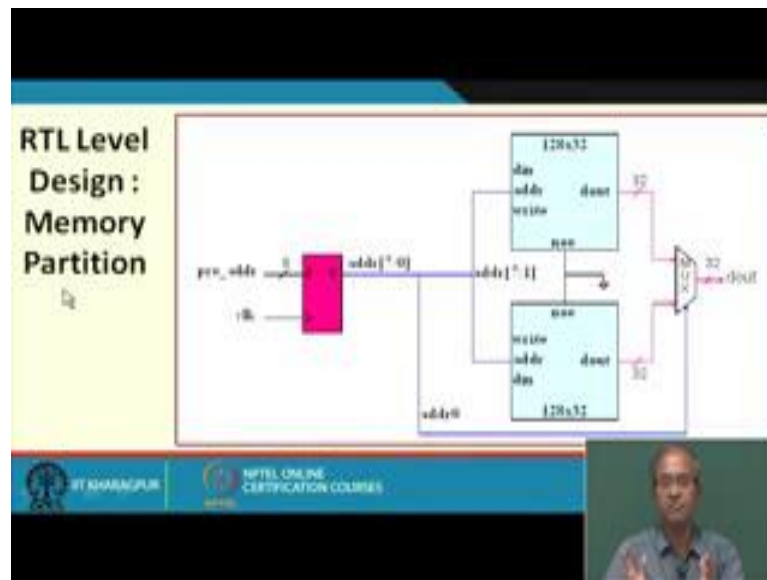
(Refer Slide Time: 23:58)



And this technique here we are saying that we can also do some re ordering at the register transfer level. This is not exactly algorithmic level; this is this is register transfer level. What it says is that well I can have several signals, some of them can be stable signal means I know that they will not change or there will be no glitch, but some of the signals I know because of unequal delays in the circuits may be comparator, there can be glitches here. Suppose I have a circuit like this where the output of a comparator is selecting some multiplexers and also some other stable signal is coming. So, some inputs are finally, selected here.

So, in a modified version of the circuit what I do, I change the order of these glitchy or stable signal, because what will happen? If the glitchy signal is used here, because of these glitches, sometimes this will be selected sometimes this will be selected. Which means this glitch will be propagating here? So, not only this line is affected this line will also get affected, but here I am using this stable signal first to control this multiplexer. So, at least up to this part there is no effect of glitch, glitch will only affect this part, but here glitch can affect not only this multiplexer also the next multiplexer. So, you can reorder it wherever it is possible.

(Refer Slide Time: 25:43)



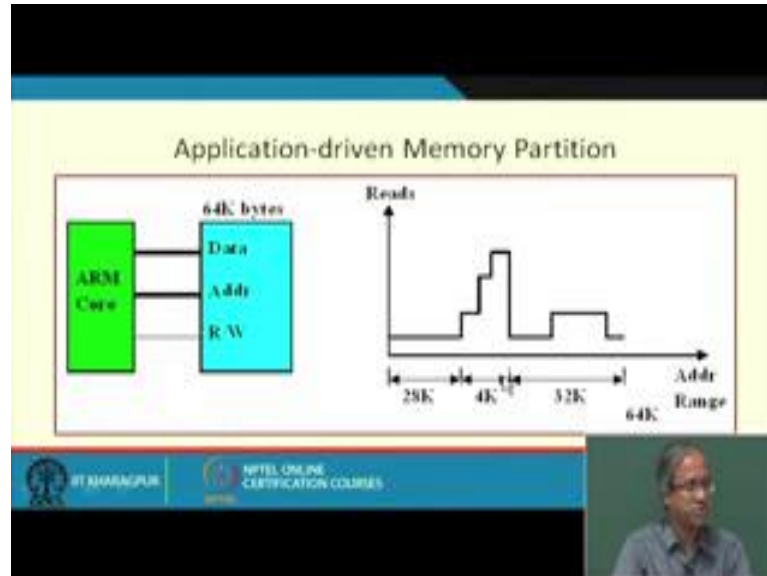
The last example that I take, here is with respect to designing memory. How to partition memory see you know whenever you design the memory system for a processor normally, you do not design it as a single block or a chip, means mean even within a chip there are multiple blocks which are designed, they are accessed parallelly. So, one of them can be selected others can be disabled there can be memory interleaving. So, many different ways are there this is what is normally done. So, if you look at this simple diagram, this is the example of a memory system, where there are let us say 32-bit data. So, these and there is 8-bit address bus.

So, this is a memory system with 256 words and each word containing 32 bits of data. So, the way we are designing it is that we are dividing that into 2 memory modules, each of them of size 128 by 32 and 128 by 32. So, what I do. So, so this among this 8 addresses, the high order 7 addresses we are using to select one word from these 2. And the least significant bit of the address I am using to select one of the data that are coming from this memory. See if I use the high order bits of selecting, and low order bits if I select one of these. So, actually I am using memory interleaving, address 0 will be here address 1 will be here, 2 will be here 3 will be here 4 will be here and 5 will be here and so on.

So, whenever the high order addresses are 0 all 0. So, address 0 the data will be coming out here, address 1 the data will be coming out here. So, by using the list address bit I

can select either this or this. This is the idea. Now if you break it up into 2 parts. So, you basically save in terms of the energy also.

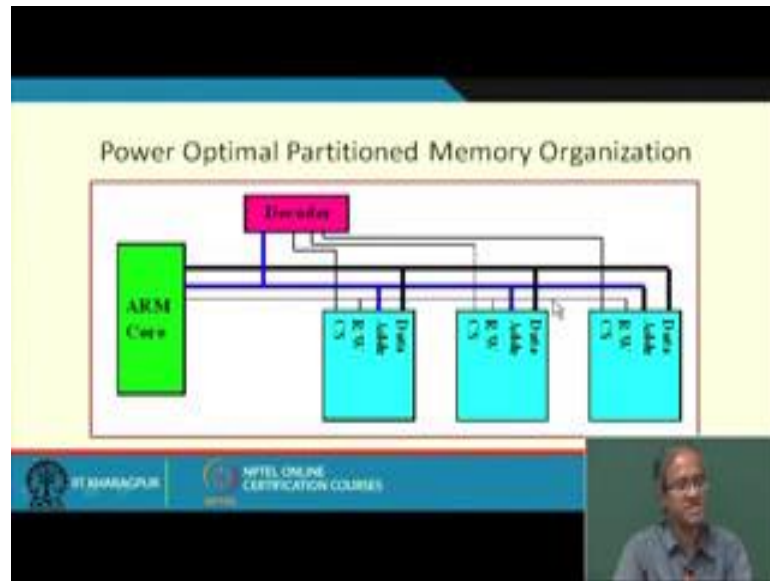
(Refer Slide Time: 28:12)



Now, let us look at another kind of a thing which is interesting, this I am saying application driven memory partition. Suppose I am designing an embedded system. So, I have a small processor let us say an arm processor, which is interfaced with 64 kilo bytes of memory. So, I do some analysis of the program that is supposed to run that is my application, and I see that out of this 64 k address space. So, I am plotting the number of reads. I see that the first 28 k there is very less number of read operations, next 4 k is very heavy in terms of read, but the last 32 k is somewhere in between.

So, I can divide my memory block into this 3 part - first, 28 k in one module; this 4 k in the second module, and this 30 32 k in the third module so my idea is that I will try to make this third module as fast as possible. The first module I can use as a power down mode, I can use a lower power supply to make it slow, but because it is accessed very rarely. So, it is fine with us and the third module can be somewhere in between because it is accessed. Relatively in frequently as compared to this 4 k.

(Refer Slide Time: 29:40)



So, you can have a circuit with 3 memory modules. So, each of them tuned means accordingly one of them can be very fast. One of them can be very slow. One of them can be in between. And the decoder can be selecting one of these 3 modules. So, so in this way you can actually partition your total memory space into several parts depending on your application and the various power profiles of this partitions can be varied.

So, there are so many techniques you can see that people use to control or reduce the power dissipation in all possible ways at the level of transistors at the level of gates at the level of architectures at the level of algorithms. So, people live no stone unturned today to reduce power dissipation in the circuits.

So, with this we come to the end of this lecture. Thank you. This is actually the last lecture of the series where we talked technical things.

Thank you.