

Hardware Security
Prof. Debdeep Mukhopadhyay
Department of Computer Science and Engineering
Indian Institution of Technology, Kharagpur

Lecture – 03
Algorithm to Hardware

Welcome to this third lecture on the topic of Hardware Security. So, today we will be trying to understand how to map Algorithms to Hardware.

(Refer Slide Time: 00:26)



So, this is essentially, this talk will be covering few aspects on hardware designs. So, we will start with understanding about what is mean by data path and control path. We will be trying to understand or identify data path and control path elements to implement algorithms. We will be trying to see you know like data path and control path design in the context of an example.

So, this example is based on a gcd processor or a greatest common device computation processor. And finally, we will be trying to see how we can do design exploration, because designing is one thing and exploration of the design is another aspects.

So, how do you design is one aspect, but how do you also explore various opportunities and various implications of your design, how do you analyze the scalability of your design. So, we will be considering FPGAs as I mentioned that K input LUTs. So, in this

case, I will be taking K equal to 4 and we will try to model the design, and will try we to understand how the performance scales in our design.

(Refer Slide Time: 01:24)

Mapping an Algorithm to Hardware

- Conversion of an algorithm to an efficient hardware is a challenging task: Performance is a key-decider.
- For an efficient design, one needs to:
 - Understand the components of a hardware design
 - Understand the architecture of the design

Data-path elements are the computational units of the design

Control-path elements sequence the data flow through the data-path elements.

The diagram shows a central 'Data Path' block connected to 'Input/Output' on the left and 'Memory' below it. A 'Control Path' block is connected to the 'Data Path' and 'Memory' via 'Control Signals'.

Logos for Swamyam and other institutions are visible at the bottom of the slide.

So, to start with when we map an algorithm to hardware, I mean you understand that performance is the main reason why we do hardware design. So, definitely you know like that is a primary goal of a design. So, for an efficient design, like if you really want to understand how to efficiently implement an algorithm in the form of an architecture or an hardware, So, you need to understand the components of the hardware ok. So, you need to understand what are the basic blocks which are hardware is made of ok.

So, there are two important aspects on a in a hardware; one is the data path, the other one is a control path ok. So, data path are basically those things, which constitutes a computational a units of your design. And the control path essentially sequences the data path elements. So, it basically kind of switches, kind of loops in around in that along the data path elements; so it basically does sequencing of your data path computational blocks ok.

So, in this diagram for example, you can see that there is a date there is, there are three important blocks. For example, there is a data path, there is a memory and there is a control path. So, the idea is that when you take inputs for example, the inputs are processed by both data path as well as the control path, but data path essentially comprises of the main computational unit ok. So, they essentially (Refer Time: 02:51)

your processing ok. Some of this data may also be residing in the memory. So, it may happen the data path elements needs to fetch some data from the memory, and also write back into the memory. And the control path is essentially kind of communicates with both. So, basically it kind of talks with both data path as well as the memory, and tries to communicate and tries you sequence your operations ok.

(Refer Slide Time: 03:15)

Case Study: Binary gcd processor

- **Input:** Integers u and v
- **Output:** Greatest Common Divisor of u and v , $z = \text{gcd}(u, v)$
- while ($u \neq v$) do
 - If (u and v are even)
 - $z = 2\text{gcd}(u/2, v/2)$
 - else if (u is even and v is odd)
 - $z = \text{gcd}(u/2, v)$
 - else if (u is odd and v is even)
 - $z = \text{gcd}(u, v/2)$
- else
 - if ($u \geq v$)
 - $z = \text{gcd}((u-v)/2, v)$
 - else
 - $z = \text{gcd}(u, (v-u)/2)$

We need to realize a co-processor on FPGA to compute gcd of two given numbers

References: Hardware Security: Design, Threats, and Safeguards, CRC Press.

swayam

So, let us take an example. So, this is an example of a binary gcd processor, a very common example from straight from the textbooks. So, you see that there are two integers u and v . And what we are interested is to; we are interested is to compute the greatest common divisor of u and v . So, in this case, the greatest common divisor is stored in the variable z ok.

So, just to parse through the binary, so you also called a binary Euclidean algorithm. So, you can see that the algorithm is to sort of a modification of the original long division Euclidean algorithm for computing, the greatest common divisor. So, the algorithm is broken up into certain parts or certain possible branches ok.

So, the first thing is that you check that whether u and v are same or not, because if you know that u and v are same, then you have a trivial result. If u and v are not same, then you do or compute in a recursive fashion ok. So, what do you first check is that whether both u and v are even; if both u and v are even, then you can say that ok, this is the if the if the greatest common divisor of u and v is z . Then that is same as if I take 2 or fact

about 2, and compute gcd of u by 2 and v by 2 ok; so that comprises your first branch here ok.

So, this is this branch which I am talking about, else it may happen that one of them is even say u is even, in that case I can divide u by 2 and compute the greatest common divisor of the of u by 2 with v ok, or it may happen that u is odd and v is even in that case, I will just divide v by 2 ok.

So, I know that since one of them is odd, and the other one is even, 2 is not a common factor or it may happen that both u and v are odd, in that case I will just check whether u is greater than equal to v. So, I will if u is greater than equal to v, then I will subtract out v from u. And I know that if there are two odd numbers, and if I subtract right, then I get ready I get an even number. So, I can divide this by 2, because I know that v is not odd v is not even. So, therefore 2 does not divide v.

So therefore, I can write it is greatest common divisor of u minus v by 2 with v, and if it is other way round instead of doing u minus v, I do v minus u ok. So, now what we want is we want to realize a co-processor on FPGA to compute the greatest common divisor of two given numbers, but I want the use this algorithm. I want to essentially kind of map this algorithm into an hardware block.

(Refer Slide Time: 05:39)

Identification of the States of the Algorithm

- **Input:** Integers u and v
- **Output:** Greatest Common Divisor of u and v, $z = \text{gcd}(u, v)$

register u and v

$XR = u, YR = v, \text{count} = 0$ State 0

while ($XR \neq YR$) do

If ($\{XR[0] \text{ and } YR[0]\}$) State 1

$XR = \text{RIGHT_SHIFT}(XR)$

$YR = \text{RIGHT_SHIFT}(YR)$

$\text{Count} = \text{Count} + 1$

else if ($\{XR[0] \text{ and } YR[0]\}$) State 2

$YR = \text{RIGHT_SHIFT}(YR)$

else if ($\{XR[0] \text{ and } YR[0]\}$) State 3

$XR = \text{RIGHT_SHIFT}(XR)$

else State 4

if ($XR \geq YR$) State 4

$XR = \text{RIGHT_SHIFT}(XR - YR)$

else



$YR = \text{RIGHT_SHIFT}(YR - XR)$

while ($\text{count} > 0$) State 5

$XR = \text{LEFT_SHIFT}(XR)$

$\text{count} = \text{count} - 1$

References: Hardware Security: Design, Threats, and Safeguards, CRC Press.



So, in order to do that, first we will try to identify the states of the algorithm, so that is the first step ok. So, this is the sequential design. We can understand this is a sequential design kind of does the computation in not like a combinational design like in one short, but it does not over several iterations ok. So, the first thing what we try to do is, we try to write the code in a format which is an HDL language, which is something like a very long language for example of VHDL language, but is not exactly HDL des depiction of the algorithm.

So, what we try to do is; we try to kind of think of a possible you know like we try to first of all think about how do we store u and v ok. So, the first thing is that we think of a register transfer logic on a RTL description. So, here we know that there are two variables u and v. So, in order to stored u and v, I allocate two registers say XR and YR which are used to store the values of u and v in my computation.

So, here the first thing is the assignment where I essentially assign u to XR and I assign v to YR ok. And then I initialize a count to 0, why I will come to explain very soon, but this is the first step ok. Now, what I do is in the algorithm what I see that I check whether XR is not equal to YR. So, this is straight away from previous algorithm, because it was u is not equal to v in the previous algorithm.

Here, I have to check that the first thing which has to check is that whether both u and v are even ok. And the way I can you know I check that very easily, if we think of a hardware or you know like an implementation of the algorithm is you what we just need to check is the LSB, you just need to check the Least Significant Bit; if the LSB is 0, then you know that u is divisible by 2 likewise for v.

So, what I just check is the last bit of XR and YR. And if we turns out that both of them are 0, then I know that we have a even number ok; we have both u and v as even. So, then what I do is I have to even if we go back to by algorithm what we have to do is we have to divide this by 2 right, we have to divide u by 2 and v by 2.

So, how will you do this in hardware, the usual way of doing this in a hardware is doing a right shift. So, here we do a right shift of XR and we do a right shift of YR, both are equivalent to saying that I divide them by 2. And then I if we remember that in my previous algorithm right I have to factor out 2, but rather than doing that now, I just keep that count of the number of times I am doing this operation ok.

So, what I do instead I a kind of increment the counter and count was initially 0 and stabling incrementing count like count plus, plus ok. Count is equal to count plus 1. So, this is one part or state of the algorithm. The others step, which I will or other possibility which can happen is that again one of them is even, and the other one is odd.

So, it can happen that either XR is even or YR odd is out or it can happen that XR is odd and YR is even. So, so therefore I kind of again you know like do a similar thing here, I just check the LSB of both XR and YR. If XR 0 is 1 and YR 0 is 0, I do a step I mean which is like I just I know that if this happens, then y is even; and if y is even, I will just like right shift y ok, I will not disturb XR.

Likewise, if it happens in other way right, I then I will disturb XR and not affect YR. The other possibility is that both of them that are odd that is a else part here. So, here again I do with comparison between XR and YR; if XR is more then I just right shift XR minus YR ok. So, basically I do $u \text{ minus } v \text{ by } 2$ that is a analogy here, likewise right I do YR minus XR.

And then when finally, all of this is done that means, when I exit this while loop I come to this point and remember I have a value of count to indicate like how many times I have taken out too. So, how I how I can do that is I mean what have to do is have to multiply two count times right, I have to do 2 to the power of count into XR or 2 to the power of count XR. So, 2 to the power of count means have two kind of length shift XR each time ok. So, therefore, I left shift XR count times and that is equivalent to do you saying that I am doing 2 to the power of count ok.

So, I do 2 count, then in 4 count and so on in powers of 2. So, if you see this algorithm, therefore, we can say that this algorithm has got six states, I have numbered them from 0 to 5. So, this is my state 0, where are initialize. This is my state 1 where I check that both of them this one possibility right, all of these states are basic an indicating some unique configuration of your algorithm.

So, here you see that state 0 stand typically for the initialization part, whereas state 1 stands for the case when both of them are even state 2 and state 3 stands for the cases when one of them is even and other one is odd. And state 4 stands for the case when both of them are odd ok. And state five stands for the case when I am I am almost going to

give you the output ok, that is a final output state ok. And hereafter where is very soon we will get the result which is been stored in a register XR ok.

So, this essentially is a very you know like a very simple example, but we will see you know like that you can actually apply this technique to implement even complex algorithms ok. So, if it is important that we understand how the state assignment are done, and how we how we translate subsequently to a hardware design ok.

(Refer Slide Time: 11:28)

The slide is titled "Identification of the Data Path Elements" and lists the following components:

- Subtractor
- Complementer
- Right Shifter
- Left Shifter
- Counter
- Multiplexer:
 - Required in large numbers for the switching necessary for the computations done in the datapath.
 - Selection lines in the multiplexer are configured by the control circuitry which is essentially a state machine.

References: Hardware Security: Design, Threats, and Safeguards, CRC Press.

The slide also features logos for IIT Bombay, Swayam, and the Ministry of Education, Government of India, along with a navigation bar at the bottom.

So, now let us see that how do we identify the date path element. So, basically as I said data path elements are nothing but computational units ok. So, let us try to see; what are the computational units which you have already used in your design ok. So, if you see for example, the elements or the components which you have used are typically shifters, you have used right shifters, you have used left shifters, you have used added ok. So, basically you have done subtraction which can be released by addition. So, definitely we need a subtractor here. And there we also need comparisons ok.

And comparators you can also implement with I mean a by a subtractor ok. So therefore, you virtually right, if you count the number of or the data of path components, then you have a subtractor, you have a complementer, you have a right shift and left shifter, and counter because you also have counted ok.

The other element which is not very explicit here is a multiplexer, because you see that there are several branches in my algorithm. So, these branches are essentially inferred as multiplexers in hardware, because multiplexer essentially gives you select logic either you go in this path or you go in other path depending upon the value of the control logic ok. So, multiplexers are required in large numbers for the switching which is necessary for competitions done in the data path, and selection lines in the multiplexers are configured by the control circuitry.

So, therefore, I mean each multiplexer we will have a select line. And this select line right will get values. So, these values will be given by the controller will be given by the control parts. So therefore, what we have to do is that we have to sequence the data path elements, we have to place the data path element, we have to place the data path element, we have to place the multiplexers, the multiplexers will be receiving inputs ok. So, the data path inputs will come from the data path elements, but the control input will come from the controller. The controller will basically configure the multiplexers and will kind of switch, so that your data path sequence gets configured ok.

(Refer Slide Time: 13:30)

The slide is titled "Identification of the State Machine of the Control Path". It contains the following bullet points:

- Control Path is a sequential design.
- It can be represented by a state machine.
- In this example, there are 6 states.
- The controller receives inputs from the partial computations of the datapath.
- Based on the current state and input, it performs state transitions.
- It also produces control signals which configures the datapath elements or switches the multiplexers to sequence the dataflow

References: Hardware Security: Design, Threats, and Safeguards, CRC Press.

The slide also features a video feed of a presenter in the bottom right corner and logos for "swayam" and "THE ONLINE EDUCATION" in the bottom left corner.

So therefore, right I mean that is what we have to discuss about the data path component. We also have the control path ok. So, the control path as I said is a sequential design. So therefore, the data path is lastly is a combination design, but the control path is a sequential design.

So, in this case, you can see that it is a 6 state, state machine, because we had six states in my algorithm. And the controller in this case receives inputs from the partial computation of the data path elements. And based upon the current state, and the input it perform state transition, it goes from like in between all the six states and produces control signals. Now, these control signals are what configures data path elements or the switches ok. So, that basically gets configured the data flow gets sequenced and that essentially is done explicitly by the controller. The controller generates these control signals which configures the multiplexers ok.

(Refer Slide Time: 14:27)

Data Path Architecture

References: Hardware Security: Design, Threats, and Safeguards, CRC Press.

- The data-path stores the value of XR and YR in two registers.
- Registers are **loadable**, ie. XR is updated when say *load_XR* is high.
- Values of *u* and *v* are initially loaded through the **input multiplexer**, using the control signal *load_uv*.
- Least bits of XR and YR are passed to the controller to indicate whether present values of XR and YR are even or not.
- Next iteration values of XR and YR are fed back after needed computations, and this is controlled by the signal *update_XR* and *update_YR*.

So, I believe that this will be more clear with this diagram. So, here you see that this is the overall architecture. So, we have got the data path element. So, this is your correspondingly your data path component. And here is your controller logic or control path ok. So, basically like this splitting of an algorithm into two components, the data path and the control path separately is a very important or central part of VLSI design at least when digital design is concerned ok.

So, in this case, you will see that I mean we can try to understand how this hardware works. So, the first thing which you would notice that the data path stores the values of XR and YR in two registers, that means, when you get the value of *u* and *v*, you load them in XR and YR ok. So therefore, XR and YR are two registers, and they are actually loadable registers which means there is an explicit load signal which is indicated as by

this signal called load underscore XR. When load underscore XR is high then XR gets loaded. Likewise when load underscore YR is high then YR gets loaded ok.

So, the values of u and v are initially you see that there is a big MUX which are placed here, which I call as the MUX A. So, it is not actually one single MUX, but it is kind of comprised of several MUX ok. It is kind of a symbolic logic. So, what it stands for here is that it receives some inputs, it receives for example, a load underscore u v. So, load underscore u v, when it is 1, then initialization happens, that means, u and v are being loaded into XR and YR. And of course, you have to also make load underscore XR high, and also load underscore YR high, to load the values of u and v into the registers XR and YR ok.

So, therefore, once the values of u are and v are initially loaded, then we start our processing. So, again you know like the state machine sees the values of XR and YR. So, these XR and YR or rather XR 0 and YR 0, which means the LSB of XR and the LSB of YR are being transferred to the controller, the controller sees these (Refer Time: 16:34), because that is very important for our control path design. So, we probably tries to understand whether both of them are even, both of them are odd, one of them is even and the other one is odd ok.

So, based upon this least bits of XR and YR are now passed to the controller to indicate whether present values of XR and YR are even or not, and then the next iteration begins ok. So, the next iteration values of XR and YR are essentially you know like based on certain computations. As we have seen in the algorithms we have to do some computation. So, for example you have to do a subtraction, you have to do a shifter and so on ok. So therefore, depending upon the after whatever is the immediate computation you do them, and then you have to update your XR and YR ok. So, you have to either update both XR, YR or maybe you have to update one of them ok. So, this logic is essentially you know like told by this signal which is called as update underscore XR. So, update underscore XR or update underscore YR basically tells whether you have to update XR or you have to update YR or you have to both or any one of them ok.

(Refer Slide Time: 17:35)

Data Path Architecture

- Computations on XR and YR are:
 - Division by 2, which is done by two **Right Shifters**.
 - Subtraction, Equality Check ($XR \neq YR$), Comparison ($XR \geq YR$), all of which is done by a **Subtractor**.
 - In case, when $XR < YR$, subtraction $YR - XR$ is to be performed, which is obtained by a **Complementer**.
- Next iteration values of XR and YR are loaded, either directly or after subtraction, which is controlled by the signal `load_XR_after_sub` and `load_YR_after_sub`
- Circuit also has an **updown counter**, which increments when both XR and YR are even.
- Finally, when $XR = YR$, the result is obtained by computing $2^{\text{count}}(XR)$, which is done by a **Left Shifter**, until count becomes 0.

Data path comprising of computational and archiving elements

Left

So therefore, what are the computations which we are doing on XR and YR are subsequently listed. So, the first thing which we have seen is the division by 2 ok. And as I said the division by 2 is done by two right shifters. So therefore, you will see that in the data MUX that there are two right shifters which are placed here. So, this is one right shifter and this is the other right shifter. So, these are the two right shifters which have been placed ok. The subtraction the other thing that you have to do is subtraction.

So, you have to do equality check trying to understand whether XR and YR become same given by the while loop condition ok. And the other thing which you have to do is to understand whether XR is greater than or equal to YR or XR is lesser than YR ok. So, this you can do both you can do by a subtractor, because the first one is an explicit subtraction, the second one you can do by a complementer which is in the 2S complement. So therefore, that also you can realize by a subtractor.

So, what we do here is therefore we place a subtractor here followed by a complementer. The subtraction is the usual subtraction between XR and YR. And depending upon whether the result is you know like negative or positive, you do a subsequent inversion of the result. So therefore, if you do not want XR minus YR, you compliment it to get YR minus XR ok. So therefore, the circuit also has got an up down counter as I said because therefore, before that before I go into that. So, after you have done these

operations, and you want to kind of load your data into XR and YR, you see that there are two ways in which you can probably load.

So, if you go back to the algorithm, you will see that the two possible ways or where you can load is you either update XR directly that means, you just right shift and update XR or it may happen that do a subtraction and then update XR ok. So, this logic is encapsulated by this signal. So, this signal is told as or written as load XR after sub or load YR after sub. That means, if load XR after sub is high, then rather than taking the result from this point which is the right shift actually what you do is you can take the result after subtraction ok. So, you basically take the result after subtraction and then y shift. Otherwise you take the result directly from XR and right shift, then you update XR. So, these are the two possible things which can happen, similarly, for YR.

So, once this process is done you have to basically there is an upcounter which is essentially as I said you did in incrementation of a counter. And finally, right when you are at the final state that is S 5 or state 5, you also can do a left shift you have to also do a left shift ok. So, there is a shifter which can either there is a counter which can either count up or countdown ok, because when you are when you have when you have done.


So remember that when we are doing this when we are in this part of the loop that means, when we were in this part of the loop we were counting up and when we are in this part of the algorithm, then we are counting down ok. So, we are either counting up or we are either counting down. So, this means that the counter that we have should be a configurable counter, it should be an up down. The counter can count up as well as countdown ok; so that is essentially the elaboration on the data path of the algorithm.

(Refer Slide Time: 20:59)

State Machine of the Controller

Present State	Next State				Output Signals													
	0	100	110	101	111	load uv	update X_R	update Y_R	load X_R	load Y_R	load X_R after sub	load Y_R after sub	Update counter	Inc /Dec	left shift	count zero		
S_0	S_5	S_1	S_2	S_3	S_4	1	0	0	1	1	0	0	0	0	1	1	0	
S_1	S_5	S_1	S_2	S_3	S_4	0	1	1	1	1	0	0	0	1	1	0	0	
S_2	S_5	S_1	S_2	S_3	S_4	0	0	1	0	1	0	0	0	0	0	0	0	
S_3	S_5	S_1	S_2	S_3	S_4	0	1	0	1	0	0	0	0	0	0	0	0	
S_4	S_5	S_1	S_2	S_3	S_4	0	1	0	1	0	1	0	0	0	0	0	0	
$(X_R \geq Y_R)$	S_5	S_1	S_2	S_3	S_4	0	0	1	0	1	0	1	0	1	0	0	0	
$(X_R < Y_R)$	S_5	S_1	S_2	S_3	S_4	0	0	1	0	1	0	1	0	1	0	0	0	
S_5	S_5	S_5	S_5	S_5	S_5	0	0	0	0	0	0	0	0	0	1	0	1	0

There are 6 States of the Controller.
 Controller receives 4 inputs from the data-path: $\{(X_R=Y_R), X_R[0], Y_R[0], X_R \geq Y_R\}$
 Example:
 Present State: S_0
 $load_uv=1, load_XR=load_YR=1.$
 Input=(0xxx) $\Rightarrow X_R=Y_R \Rightarrow$ Next State is $S_5.$
 Input=(100x) $\Rightarrow X_R=Y_R,$ both X_R and Y_R are even \Rightarrow Next State is $S_1.$



Now, we can come to the state machine of the algorithm. So, in the state machine as I said there are six states starting from S_0 to S_5 . And what we try to see over here is how the **controller** how the state transitions take place. So, I will try to explain few of them for example, I mean I believe that we can understand the remaining from those from these examples.

So, in this case the controller receives four input from the data path ok. So, if you see right, here the controller receives some inputs. So, these inputs are X_R not equal to Y_R , X_R is greater than or equal to Y_R likewise right there are four inputs with the controller receives. It receives X_R equal to Y_R , $X_R[0]$, $Y_R[0]$ which are the LSBs of X_R and Y_R . And the result of the subtraction, that means, X_R is greater than or equal to Y_R or not. So, all of them can be indicated by 0, 1 values ok. So, therefore these essentially determine; the state transitions as well as also has got influence on the output of the controller ok.

So, for example, if your present state is S_0 , that means, that you are in the initialization state, then your load underscore u v is 1. Load underscore X_R and load underscore Y_R are both 1, because you are loading X_R and Y_R with the content of u and v ok. And suppose you receive an input denoted as 0, and followed three don't cares, because the moment you see this being 0, you know that this is 0; that means, X_R is not equal to Y_R is 0, which means X_R is equal to Y_R ok.

And what you do in the algorithm, if you remember right if you see XR if you see if you go back right to the algorithm, and check what you do when you see that. So, this loop right, this while loop continues when XR is not equal to YR, so that means, when XR becomes equal to YR, you exit the while loop and you come to state 5 ok. So, therefore, exactly this is what is been depicted here in the state machine, that means, whenever you see that this happens you know that the next state is S 5 ok, and that is essentially tabulated in the table.


Likewise if you receive an input which is 1 0 0 cross which means like XR is not equal to YR, and you see that both XR 0 and YR 0 are 0 that means, both of them are even. Then you see that the state is S 1, because remember that the state S 1 was what which was which was processing that particular scenario right. I mean XR this is the state right that is when both of them are even, then this is the state which is handling this ok. So, therefore then you need to jump to state 1 ok. So therefore, that is exactly what is done here the next state is S 1.

(Refer Slide Time: 23:38)

State Machine of the Controller

Present State	Next State					Output Signals										
	0	100	110	101	111	load uv	update XR	update YR	load XR	load YR	load XR after sub	load YR after sub	Update counter	Inc/Dec	left shift	count zero
S ₀	S ₅	S ₁	S ₂	S ₃	S ₄	1	0	0	1	1	0	0	0	0	0	0
S ₁	S ₅	S ₁	S ₂	S ₃	S ₄	0	1	1	1	1	0	0	1	1	0	0
S ₂	S ₅	S ₁	S ₂	S ₃	S ₄	0	0	1	0	1	0	0	0	0	0	0
S ₃	S ₅	S ₁	S ₂	S ₃	S ₄	0	1	0	1	0	0	0	0	0	0	0
S ₄	S ₅	S ₁	S ₂	S ₃	S ₄	0	1	0	1	0	1	0	0	0	0	0
(X _R ≥ Y _R) S ₄	S ₅	S ₁	S ₂	S ₃	S ₄	0	0	1	0	1	0	1	0	0	0	0
(X _R < Y _R) S ₄	S ₅	S ₁	S ₂	S ₃	S ₄	0	0	1	0	1	0	1	0	0	0	0
S ₅	S ₅	S ₅	S ₅	S ₅	S ₅	0	0	0	0	0	0	0	1	0	1	0

Present State: S1
 load_uv=0, load_XR=load_YR=1, update_XR=update_YR=1, Update Counter=1, Inc/Dec=1.
 Input=(0xxx)=>XR=YR=>Next State is S5.
 Input=(100x)=>XR=YR, both XR and YR are even=>Next State is S1.
 Input=(110x)=>XR≠YR, XR is odd, YR is even=>Next State is S2.



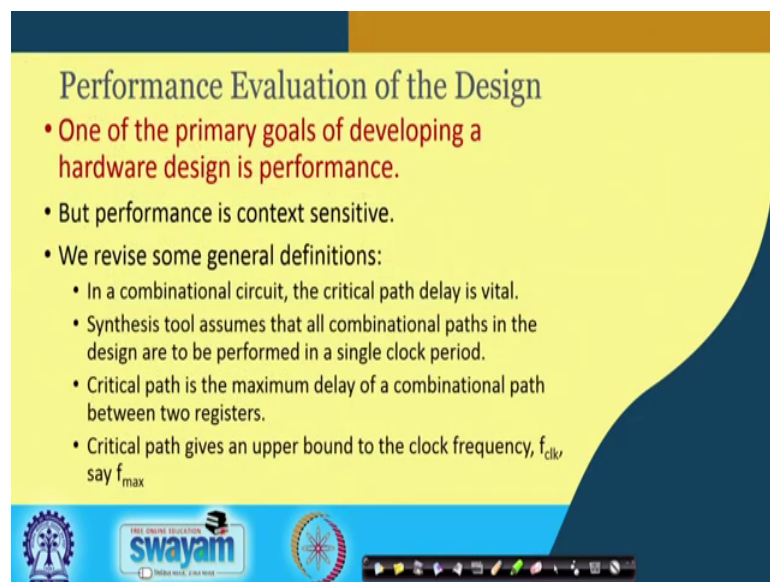
So, likewise when you see the present state is S 1 ok, if the state is S 1 at this point, you see that what you essentially do right if you go back to the algorithm, when you see that both of them are even, then I mean for example, right if you go back to the algorithm, and check exactly what is done here, you do a right shift of XR and a right shift of YR. And you update XR and YR and you also up count a counter.

So now, if you go back to the state table then you see what you do exactly is this you make load underscore XR and load underscore YR both 1. You update underscore XR and update underscore YR also as 1 ok. And update counter is also 1 that means you also have to increment your counter. And whether you are incrementing or decrementing is denoted by this signal ok.

So, likewise you again receive inputs like 0 cross or there is 0 don't care, don't care, don't care. So, in this case again you have to make the next state as S 5, if you get 1 0 0 cross, then you have to make the next state as S 1. But if you get and input like 1 1 0, then you know XR is odd, and therefore, your next state has to be S 2 ok. So, that is what you have essentially done throughout the **state** transition table, and you essentially can populate the entire table in this fashion ok.

Now once we have done that right you basically have your hardware. So, your hardware you should implement this in very log, and you should have the data path and the control path. And essentially it should be like as I said in the previous class, you have to simulate that and see that computation.

(Refer Slide Time: 25:18)



Performance Evaluation of the Design

- One of the primary goals of developing a hardware design is performance.
- But performance is context sensitive.
- We revise some general definitions:
 - In a combinational circuit, the critical path delay is vital.
 - Synthesis tool assumes that all combinational paths in the design are to be performed in a single clock period.
 - Critical path is the maximum delay of a combinational path between two registers.
 - Critical path gives an upper bound to the clock frequency, f_{clk} , say f_{max} .

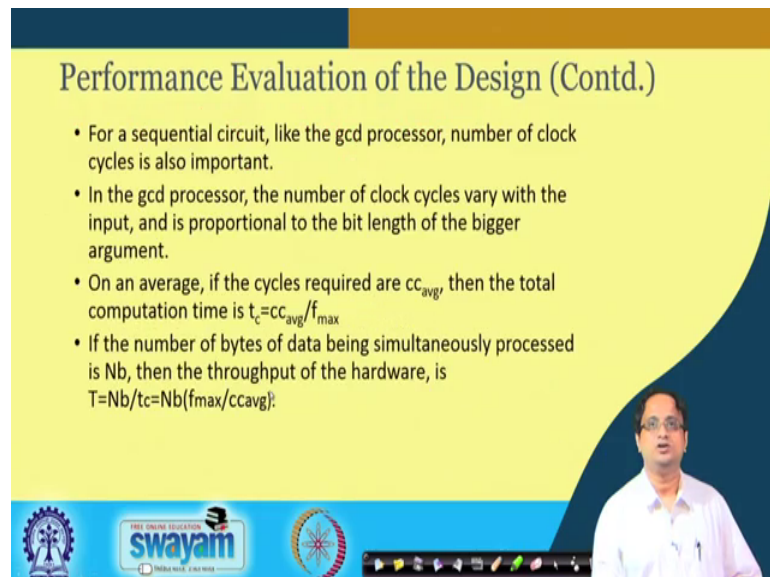
The slide features a yellow background with a blue and orange header. At the bottom, there are logos for 'swayam' (The Online Education) and 'INDIAN INSTITUTE OF TECHNOLOGY' along with a navigation bar.

So, now we essentially come to the next part or you know the exploratory part. We are here to do a performance evaluation of the design, because as I say that is one of the primary goals of developing a hardware design is performance ok. So, the performance is context sensitive like depending upon several you know like your application, you may

either want you know like to make it low power or you want to make it fast or maybe the objectives are different.

But broadly right in a combinational circuit, the most important aspect what you want to optimize is the critical path delay ok. So, the critical path delay right essentially is often predicted by synthesis tool which tells you right that from register to register what is the maximum path, and that kind of gives a bound on the maximum frequency that you can operate ok. So, for example, your maximum frequency right at which clock frequency, f_{clk} can operate you say f_{max} ok.

(Refer Slide Time: 26:10)



The slide is titled "Performance Evaluation of the Design (Contd.)" and features a yellow background with a dark blue curved shape on the right side. A presenter is visible in the bottom right corner. The slide contains the following text:

- For a sequential circuit, like the gcd processor, number of clock cycles is also important.
- In the gcd processor, the number of clock cycles vary with the input, and is proportional to the bit length of the bigger argument.
- On an average, if the cycles required are cc_{avg} , then the total computation time is $t_c = cc_{avg} / f_{max}$
- If the number of bytes of data being simultaneously processed is N_b , then the throughput of the hardware, is $T = N_b / t_c = N_b (f_{max} / cc_{avg})$.

At the bottom of the slide, there are logos for "THE INDIAN EDUCATION SWAYAM" and "SWAYAM" along with a navigation bar.

So, likewise right for the sequential if you consider sequential circuits, then along with the critical path, the other important thing is the number of clock cycles which you need to complete. Now, if you observe the gcd processor, we just now discussed right it is not a constant time implementation that means right depending upon the inputs u and v , the number of clock cycles which would be required can vary. For example, in this case it will be proportional to the bit length of the bigger argument ok.

So, therefore, on an average right, if you denote the number of clock cycles as cc_{avg} , then the total computation time will be cc_{avg} / f_{max} ok. So, if the number of bytes of data being simultaneously processed is N_b , then another very important parameter was what is called as throughput ok. In this case the throughput will be denoted as N_b / t_c because t_c is the time which is required to do a computation and

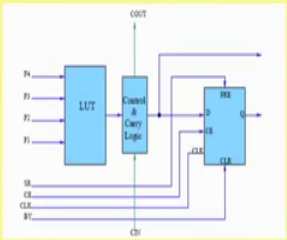
therefore, you will get something like $N \cdot b$ into f_{max} by cc average. So, this can denote the throughput of your hardware. So, when you are making a new design, these are the parameters which you have to quote about your design and show how competitive it is ok.

(Refer Slide Time: 27:15)

Resource Consumption

The other important aspect is resource consumed.

In context to FPGAs, the resources largely comprise of slices, which are made of LUTs, and flip-flops.

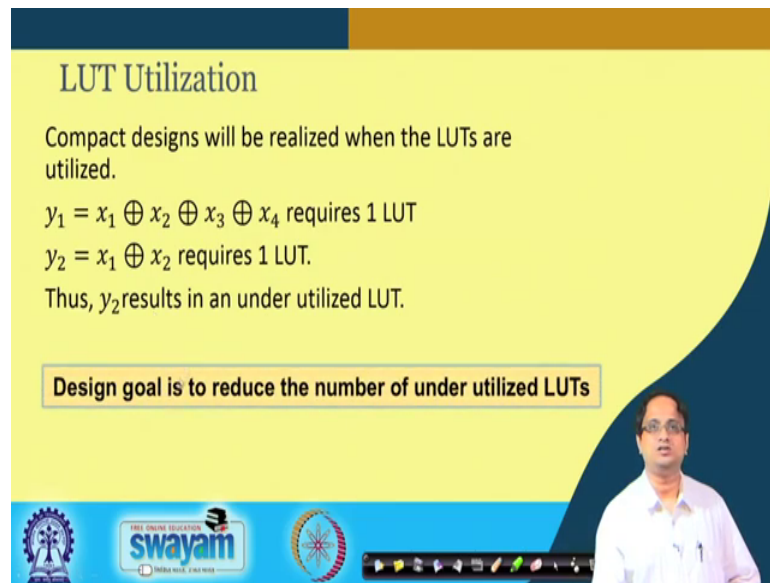


- LUT Structure of Xilinx Virtex-4
 - Four Input , One Output.
 - Can contain 16x1 SRAM.
 - Can implement any four input truth table.

The slide features a yellow background with a blue header and footer. A small inset image of a man in a white shirt is visible in the bottom right corner of the slide area. The footer contains logos for Swamyam and other educational institutions.

So, now when you want to make a design right, the other important thing is the resource consumed. So, you also want to make your design more compact. So, therefore, let us take a look back at our structure of the LUTs. So, this is the structure of our LUT for Xilinx Virtex 4, where the LUT has four inputs. So, it is basically realizing any four bit Boolean function ok.

(Refer Slide Time: 27:36)



LUT Utilization

Compact designs will be realized when the LUTs are utilized.

$y_1 = x_1 \oplus x_2 \oplus x_3 \oplus x_4$ requires 1 LUT

$y_2 = x_1 \oplus x_2$ requires 1 LUT.

Thus, y_2 results in an under utilized LUT.

Design goal is to reduce the number of under utilized LUTs

The slide features a yellow background with a dark blue curved shape on the right. At the bottom, there is a blue banner with logos for 'swayam' and 'THE ONLINE EDUCATION' along with a presenter's video feed.

So, now if you take these two functions like these are both Boolean functions y_1 and y_2 , one of them is XOR of x_1, x_2, x_3 , the other one is an XOR of x_1 and x_2 . So, you see that in terms of gate equivalence, one actually takes only one XOR, whereas the other one takes 3 XORs ok. So, I would expect this one is more consuming than this one. But if you consider FPGA designs which is slightly different you see both of them will take one LUT ok. So, I will say therefore, say that the first design which does more computation is actually still needing 1 LUT, that means, it is utilizing the LUTs better ok.

So therefore, y_2 is actually underutilized LUT. The LUT is not maximal utilize because you see that once you have realized y_2 in that you cannot realize any more function in that the remaining part is kind of a waste. So therefore, what we want right is we want to make the number of unutilized or underutilized LUTs as minimal as possible ok.

(Refer Slide Time: 28:34)

LUT Under Utilization

Minimum number of LUTs required for a q-bit combinational circuit (for 4 input LUT)

$$\#LUT(q) = \begin{cases} 0, & \text{if } q = 1 \\ 1, & \text{if } 1 < q \leq 4 \\ \lceil q/3 \rceil, & \text{if } q > 4 \text{ and } q \bmod 3 = 2 \\ \lfloor q/3 \rfloor, & \text{if } q > 4 \text{ and } q \bmod 3 \neq 2 \end{cases}$$

Delay of a q-bit combinational circuit.

$$DELAY(q) = \lceil \log_4(q) \rceil D_{LUT}$$

LUT_k denotes that k inputs out of 4 are used by the design block realized by the LUT.

$$\%UnderUtilizedLUTs = \frac{LUT_2 + LUT_3}{LUT_1 + LUT_2 + LUT_3 + LUT_4} * 100$$

The slide also features logos for IIT Bombay and Swamyam, and a small video inset of a presenter in the bottom right corner.

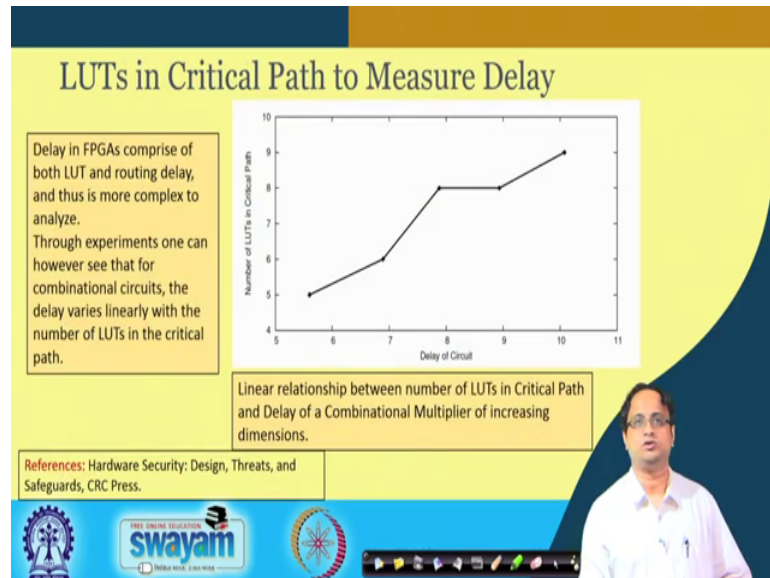
So, how do we do that? So, here we without you know like going into the derivations you see that we can model them pretty accurately ok. So, what we can do here is for example, the number of LUTs for a q bit combinational circuit that means, a combinational circuit which has got q inputs for a four input LUT can be modeled by these equations ok. So, you see that I have covered different cases, where q is 1, then of course, I do not need any LUT ok. If q is less than 1 and 4, then I need one LUT, but if it is more, then depending upon two cases, I can pretty much say that is like the (Refer Time: 29:09) or float of q by 3 ok.

So, the other important parameter is the delay. So, again you know like delay is a hard thing to compute in FPGAs, but we just want to compute the number of LUTs which lies in the max path. Then you can pretty much accurately model that by log q base 4. For a k input LUT, it will be log q base k ok. And then I multiply it with the delay of the LUT to get an understanding about what is the delay for say log q base 4 LUTs. So, therefore, this gives me the delay for a q bit combinational circuit ok.

So, now the question is like if I want to know; what is the percentage of underutilized LUT, then you observe that I u, there can be I denote LUT k, to denote the fact that the k input k inputs out of 4 are used. For example, in my previous case, y 1 was LUT 4 because I have got 4 inputs to the function ok; y 2 was a two input, so it was LUT 2. So, you can say in that case right that LUT if I consider LUT 2, LUT 3 and LUT 4, then LUT

LUT 4 is not wasting any look up table, but the remaining thing that means, LUT 2 and LUT 3 are what are contributing to your underutilization of look up tables ok. So, therefore, the way in which you can calculate the percentage of underutilized LUTs is by adding LUT 2 with LUT 3, and dividing by LUT 2 plus LUT 3 plus LUT 4. So, you will get a sort of an estimate about the under utilization of the look up tables ok.

(Refer Slide Time: 30:45)



As I said that for FPGAs because in FPGAs you do not only have LUTs you also have routing delays, it is very complex to analyze delay ok. But from experiments you can see that if you take combinational circuits, then the delay will be actually vary linearly with the number of look up tables in the critical path ok. So, here is an experiment that we did for some combinational multipliers.

And for this combinational design, we will start increasing the dimension of the multiplier and we start plotting the delay with the number of look up tables in the critical path. Then we see more or less linear relationship ok. So therefore, we can pretty much accurate for accurately predict the delay or at least understand the trend of the delay by using the number of look up tables or by like giving an estimate of the number of look up tables which lie in the critical path of my design ok.

(Refer Slide Time: 31:37)

Modeling the Components of the gcd processor

Most important component in the data-path of the gcd processor is subtractor, which can be realized by an adder.

On FPGA platforms, carry chain based adder are specially optimized and are fast.

For Xilinx Virtex IV FPGAs, s is almost 17.

The carry chains use m-MUXCY for m-bit adder. They are much faster compared to the LUTs which are used for other parts of the circuit. So, in order to compare we scale the delay, and state:

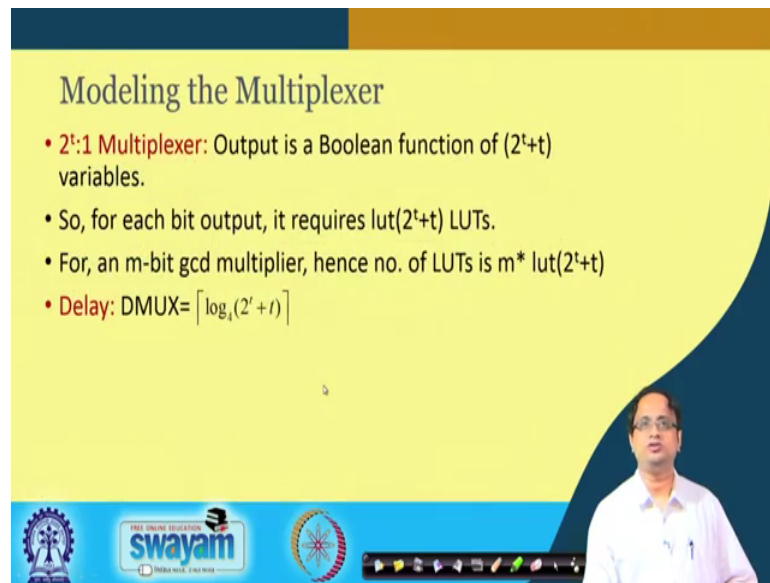
$$D_{add} \text{ (or } D_{sub}) = \lceil m/s \rceil$$

References: Hardware Security: Design, Threats, and Safeguards, CRC Press.

So, once we have these two formalism right, we can essentially start modeling the components of the greatest common divisor. So, if you see in the greatest common divisor, the other very important component is the adder ok. Now, modeling the adder by this look up tables is slightly tricky, because here if we can observe that in most of the FPGA circuits there are dedicated adders which mean that we have got the normal look up tables, but there is also an additional chain ok. Now, this addition chain is much faster compared to the normal LUT delay ok.

So, in order to understand that what we did is we basically incorporated a scaling factor. You can incorporate a scaling factor which will essentially tell say that you now like if you are if you want to kind of estimate the delay of your adder, then for an m bit adder the delay can be estimated by m by S and seal of that. That means, like that means, right you are basically reducing or time to time incorporate the speed of your carry chain ok. So, for vertex 4, you can estimate that S will be around 17, so that means, like with this value of S , you can actually you can actually you know like compare the added delay with other look up table components. So, it kinds of brings it to the same platform.

(Refer Slide Time: 32:55)



The slide is titled "Modeling the Multiplexer" and contains the following text:

- **2^t:1 Multiplexer:** Output is a Boolean function of (2^t+t) variables.
- So, for each bit output, it requires lut(2^t+t) LUTs.
- For, an m-bit gcd multiplier, hence no. of LUTs is m* lut(2^t+t)
- **Delay:** DMUX = $\lceil \log_4(2^t + t) \rceil$

The slide also features a presenter in the bottom right corner and logos for "swayam" and "THE ONLINE EDUCATION" in the bottom left corner.

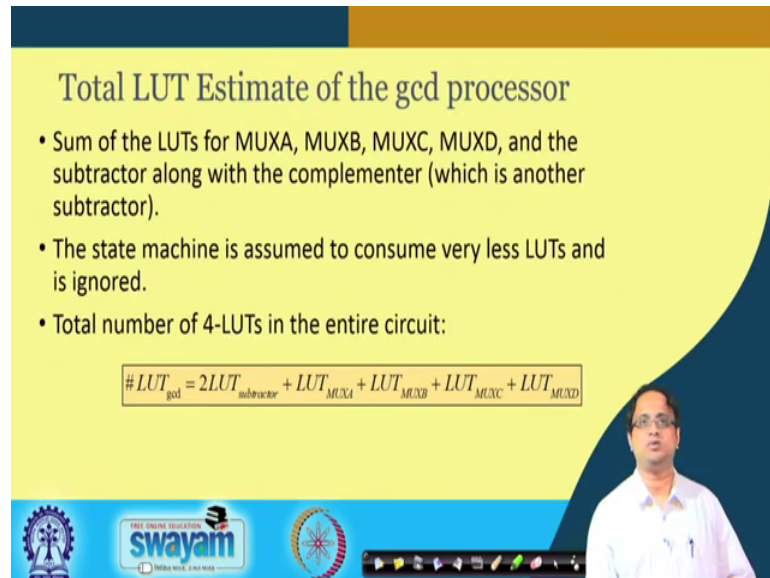
So, therefore, therefore, once you have these tools, you can observe that you can model the multiplexer. Remember that I am multiplexer is a 2 to the power of t is to 1 multiplexer which means that there are t select lines, and there are 2 to the power of t inputs. So, therefore, the multiplexer is a Boolean function of dimension 2 to the power of t plus t variables ok. That means the number of variables to the multiplier the number of input variables of the multiplexer can be modeled as or can be enumerated as 2 to the power of t plus t.

So, therefore, if you fit into the previous expressions the number of look up tables will be around you can model it by LUT 2 to the power of t plus t. So, 2 to the power of t plus t is an argument to the LUT function that we just now elaborated mentioned. So, therefore, for, a m-bit gcd multiplier, the number of look up tables is m into LUT 2 to the power of t plus t. So, here m is the m-bit. So therefore, you can visualize that each multiplexer is I mean each multiplexer can be broken up into m multiplexers, so all these multiplexers are working in parallel ok.

So, therefore, the number of look up tables will be m times the number of look up tables which are required to implement one multiplexer ok. And the number of look up tables to implement one multiplexer is LUT 2 to the power of t plus t and that I multiply it with m to get the overall number of look up tables. Likewise right you can get the delay MUX, the delay MUX is 2 to the power of t plus t, and you do a log with base 4 ok. And

therefore, you get an estimation of the delay you exactly fit into the previous expression and you get an estimate of the delay.

(Refer Slide Time: 34:26)



Total LUT Estimate of the gcd processor

- Sum of the LUTs for MUXA, MUXB, MUXC, MUXD, and the subtractor along with the complementer (which is another subtractor).
- The state machine is assumed to consume very less LUTs and is ignored.
- Total number of 4-LUTs in the entire circuit:

$$\#LUT_{gcd} = 2LUT_{subtractor} + LUT_{MUXA} + LUT_{MUXB} + LUT_{MUXC} + LUT_{MUXD}$$

The slide also features a presenter in the bottom right corner and logos for Swamyam and other institutions at the bottom.

So, now you kind of estimate the delay of the LUT of the overall design, you can see that you have got MUXA, MUXB, MUXC and MUXD in the there are four multiplexers in your architecture, and there are two subtractors ok. So therefore, the overall the total LUT count will be two times that required for a subtractor plus the summation of all the four multiplexers. Similarly you can, so in this case actually I have ignored the look up tables which are required to the state machine. For the state machine assuming that it is much lesser compared to the data path components ok.

(Refer Slide Time: 35:02)

Total Delay Estimate of the gcd processor

- Critical path of the design:
 subtractor → complementer → MUXD → MUXB → MUXA.

$$D_{gcd} = 2D_{sub} + D_{MUXD} + D_{MUXB} + D_{MUXA}$$

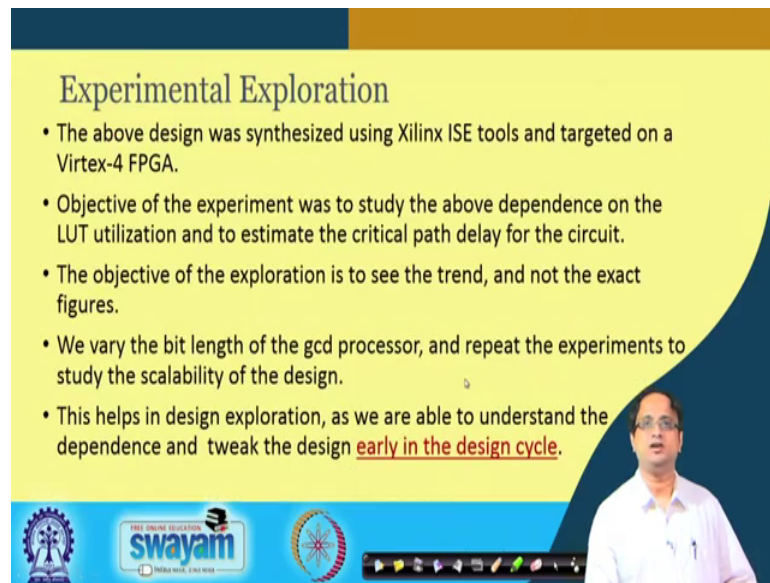
$$= 2 \lceil m/s \rceil + 1 + 1 + 1 = 3 + 2 \lceil m/s \rceil$$

Note that the delay of MUXA comes from the fact that the multiplexer is made of 2 smaller 2-input multiplexers in parallel: one input writing to XR, while the other writing to YR.

Likewise you can also calculate the estimated total delay. So, in this case, this is your data path. So, again you can see that there are four multiplexers MUXA, MUXB, MUXC and MUXD which we essentially already **enumerated** you know like accounted for while counting the number of look up tables. If you see the delay right, this is your critical path. So, we start from this register and we come back to this register ok, so that means, it starts with a subtractor, goes to the complementer, goes to this MUXD, comes back to MUXB and finally, goes to MUXA.

So, you can estimate the delay in this way, you can estimate the delay in this fashion. So, for example, it says like 2 m by S plus 1 plus 1 plus 1. So, in this case, it is 3 plus 2 m by S, and therefore that gives you the overall delay ok. So, why you essentially bringing one delay for the MUX is because you can visualize although it is a big MUX or it looks like a big MUX. You can visualize this MUX to be made of a two multiplexers, one multiplexer for updating XR, and the other multiplexer for updating YR. And both of them have got delay of one look up tables ok, so or one look up table. So therefore, therefore we write here one for each of these multiplexers.

(Refer Slide Time: 36:12)



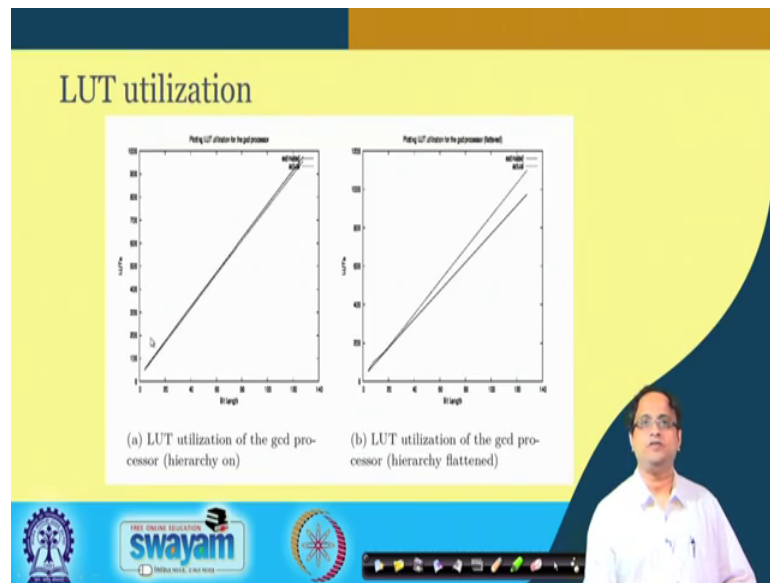
Experimental Exploration

- The above design was synthesized using Xilinx ISE tools and targeted on a Virtex-4 FPGA.
- Objective of the experiment was to study the above dependence on the LUT utilization and to estimate the critical path delay for the circuit.
- The objective of the exploration is to see the trend, and not the exact figures.
- We vary the bit length of the gcd processor, and repeat the experiments to study the scalability of the design.
- This helps in design exploration, as we are able to understand the dependence and tweak the design **early in the design cycle**.

The slide features a yellow background with a dark blue header and footer. A small video feed of a presenter is visible in the bottom right corner. The footer includes logos for 'swayam' and 'THE ONLINE EDUCATION' along with a navigation bar.

So, with this estimates right you can actually try to experimentally validates. So, the idea is that you can try to take a vertex four device and take a Xilinx ISE tool or any other FPGA tool. And try to repeat this experiment for different versions of gcd. So, you can do gcd implementations for different bit lengths, and try to see how your design skills ok. So, you will get some estimate from the cat tool and you can use these modeling estimates also and try to see how well they correlate. The reason why these models are useful if you have these models; then without doing the design you can actually got the sort of like the performance of your design, and therefore, you can make decisions already in the design cycle. Note that, it is very important to take decisions early in the design cycle because it can save lot of time and manpower ok.

(Refer Slide Time: 37:00)



So, therefore right here is an example of how your look up table skills. So, you see that we have got a very much to your like a close following. Although you know like the LUT estimations that we had were actually minimal, that means these are the minimal number of LUTs you need to implement ok. So, therefore, right it matters pretty well. If you see the delay, this is the delay estimation although it does not match exactly, but you still see that the trend is predictable ok.

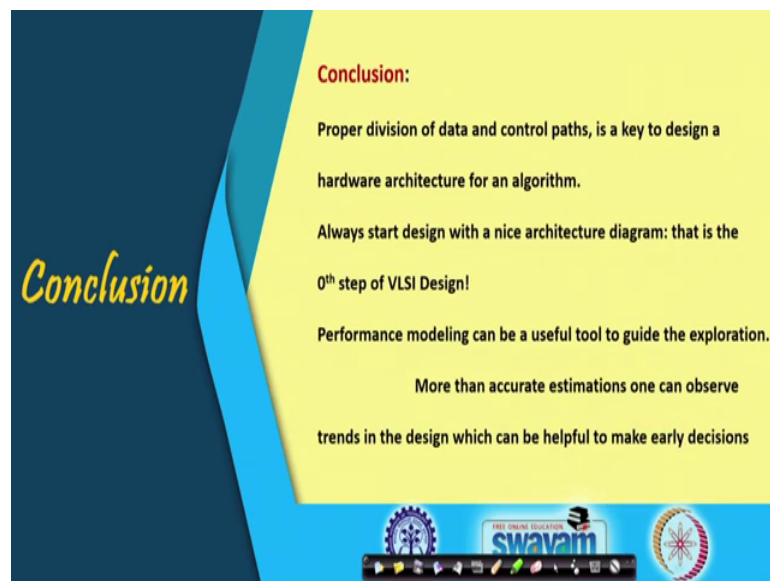
So, I have given two designs two plots here, one of them is for when you have flattened the hierarchy, and the other one when you have not flattened the hierarchy. So, these are so the flattening means that if you take a design right you basically you have got different modules in your design, but when you flatten it, you flatten them to one level that means you have got one single level. And the other is like when you have kept the hierarchy on in you synthesizer; that means, that you essentially are maintaining the hierarchy of your design ok. You will get slightly different estimates for them.

(Refer Slide Time: 37:56)



So, these are the references that I have used for the discussion.

(Refer Slide Time: 38:02)



And you know like so what we basically discussed is that it is very important that we do a proper division of data path and control path and that is a key for hardware design or hardware architecture design for an algorithm. Always we have to start with a nice architecture diagram that is a zeroth step of VLSI design. And performance modeling can be a useful tool to guide the exploration more than accurate estimation. One can observe

trends in the design which can be helpful in making all the decisions in your design cycle ok.

With this, thanks for your patience. And hope we will again continue on subsequent topics in the next class.