

HARDWARE MODELING USING VERILOG

Prof. Indranil Sengupta
Computer Science and Engineering
IIT Kharagpur



INDEX

S. No	Topic	Page No.
	<i>Week 1</i>	
1	Lecture 1	1
2	Lecture 2	17
3	Lecture 3	35
4	Lecture 4	53
5	Lecture 5	68
	<i>Week 2</i>	
6	Lecture 6: Verilog Language Features (Part 1)	86
7	Lecture 7: Verilog Language Features (Part 2)	103
8	Lecture 8: Verilog Language Features (Part 3)	121
9	Lecture 9: Verilog Operators	136
10	Lecture 10:Verilog Modeling Examples	155
11	Lecture 11: Verilog Modeling Examples (Contd)	170
	<i>Week 3</i>	
12	Lecture 12: Verilog Description Styles	188
13	Lecture 13: Procedural Assignment	206
14	Lecture 14: Procedural Assignment (Contd.)	223
15	Lecture 15: Procedural Assignment (Examples)	239
	<i>Week 4</i>	
16	Lecture 17:Blocking / Non-Blocking Assignments (Part 2)	258
17	Lecture 17: Blocking / Non-Blocking Assignments (Part 2)	276
18	Lecture 18:Blocking / Non-Blocking Assignments (Part 3)	294
19	Lecture 19:Blocking / Non-Blocking Assignments (Part 4)	309
20	Lecture 20:User Defined Primitives	328
	<i>Week 5</i>	
21	Lecture 21 : Verilog Test Bench	343
22	Lecture 22 : Writing Verilog Test Benches	361
23	Lecture 23 : Modeling Finite State Machines	376
24	Lecture 24 : Modeling Finite State Machines (Contd.)	392
	<i>Week 6</i>	
25	Lecture 25 : Datapath And Controller Design (Part 1)	409
26	Lecture 26 : Datapath And Controller Design (Part 2)	424
27	Lecture 27: Datapath And Controller Design (Part 3)	440

28	Lecture 28 : Synthesizable Verilog	453
29	Lecture 29 : Some Recommended Practices	470

Week 7

30	Lecture 30: Modeling Memory	488
31	Lecture 31: Modeling Register Banks	505
32	Lecture 32: Basic Pipelining Concepts	520
33	Lecture 33: Pipeline Modeling (Part 1)	536
34	Lecture 34: Pipeline Modeling (Part 2)	550
35	Lecture 35: Switch Level Modeling (Part 1)	569
36	Lecture 36: Switch Level Moddeling (Part 2)	582

Week 8

37	Lecture 37 : Pipeline Implementation Of A Processor (Part 1)	597
38	Lecture 38 : Pipeline Implementation Of A Processor (Part 2)	612
39	Lecture 39 : Pipeline Implementation Of A Processor (Part 3)	628
40	Lecture 40 : Verilog Modeling Of The Processor (Part 1)	644
41	Lecture 41 : Verilog Modeling Of The Processor (Part 2)	660

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

Lecture – 01
Introduction

Welcome to the course on Hardware Modeling using Verilog. Now in this course over the next 8 weeks, we shall be discussing the various features of the Verilog hardware description language, and see that how as a designer you can utilize the facilities and the features that are they are as the part of the language to the best possible extent.

So, today we start with some of the basic introductory topics.

(Refer Slide Time: 00:58)

Main Objectives of the Course

Hardware Modeling Using Verilog

1. Learn about the Verilog hardware description language.
2. Understand the difference between behavioral and structural design styles.
3. Learn to write test benches and analyze simulation results.
4. Learn to model combinational and sequential circuits.
5. Distinguish between good and bad coding practices.
6. Case studies with some complex designs.

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

So, let us start by talking about the main objectives of this course. So, as you know the name of this course is hardware modeling using Verilog. Now Verilog is one of the so-called hardware description languages that you may already be knowing, it is a language using which a designer can specify the behavior, or the functionality, or the structure of some given hardware; some specified hardware circuit. Now in this as part of this course well we shall of course, be learning about the Verilog hardware description language, its various features, the syntaxes, and so on.

Specifically, we shall be looking at, two different distinct way of modeling the functionality of a circuit this so-called behavioral and the structural design styles. So, we

shall be explaining the differences. And from the point of view of verifying whether the design is correctly working or not, its correct or not, we have to write something called test benches, or test harness.

So, we shall also see how to write such test benches and evaluate the results of simulations, we shall be learning about modeling both combinational and sequential circuits. And during the course of this we shall be learning also what are the good practices and what are the so-called avoidable practices that a designer should be aware off. And of course, we shall be looking at some of the case studies. Specifically, at the end we shall be looking at the design of a complete processor and we see how using Verilog, we can design the data path and also the control path of the processor, ok.

(Refer Slide Time: 03:06)

VLSI Design Process

- Design complexity increasing rapidly
 - Increased size and complexity
 - Fabrication technology improving
 - CAD tools are essential
 - Conflicting requirements like area, speed, and energy consumption
- The present trend
 - Standardize the design flow
 - Emphasis on low-power design, and increased performance

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

So, we start by talking a few things about the VLSI design process. Because you see whenever you are talking about a hardware description language, you are directly or indirectly talking about some piece of hardware which you are trying to design. Now, the first and the most natural kind of hardware building block that comes to our mind is a chip, it is an IC. So, today we are in the area of very large-scale integration or VLSI, so we talk about a VLSI chip as our basic hardware building block.

So, when you talk about the VLSI design process there are a few things that we need to keep in mind. First thing is that over the years the complexity of VLSI circuits and consequently the design, they have increased dramatically. So, means when I say it is

increase dramatically, means in fact there has been an exponential increase over the years. I shall show you a slide just depicting the kind of increase that has taken place. But the point to notice that earlier few decades back we used to design some chips, which consisted or contained few 100 or 1000 of gates or transistors, but now today we are talking about circuits or chips consisting of billions of transistors. So, you can just see the difference the dramatic advancements, and improvements that have been taken place over the years, ok.

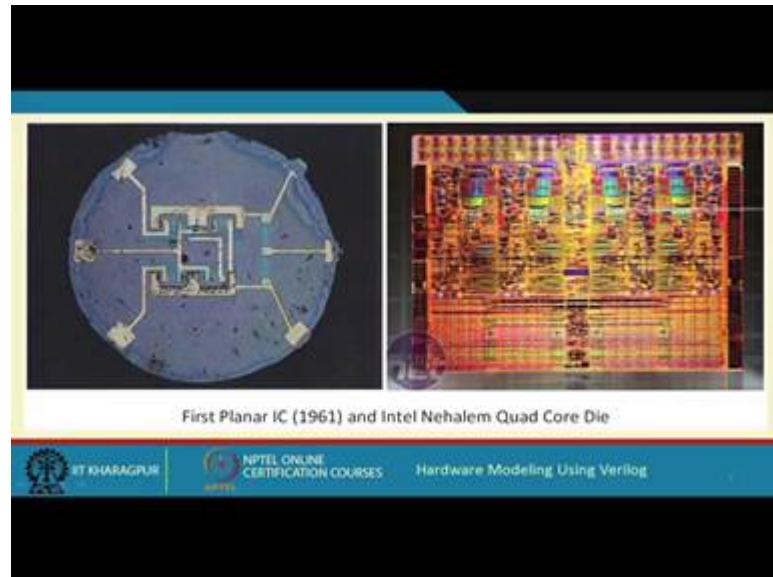
So, because of this increased size and complexity, this has been made possible of course because of improvement in the fabrication technology. The VLSI fabrication technology have improved dramatically. And as a consequence because of the complexity of the circuits manual design is simply ruled out. So, earlier when the circuit was smaller you could have designed your circuits on a piece of paper layout, on a piece of graph paper and so on and so forth, but now when we are talking about millions and billions of transistors you have to make use of a computer system and use some so-called computer aided design tools or the CAD tools.

And in the process there is some conflicting requirements that often come, in front of the designer. Like for example, the designer may want to reduce the area, the designer may want to increase the speed, the designer may also want to reduce the energy consumption, because as you know most the circuits today are working on battery. So, it is quite natural to try and conserve the power or the energy in the battery for a longer period of time.

So, it is very important to come up with some design ideas or principles that will consume less energy, right. But often these requirements are conflicting. When you try to reduce area may be your delivery increases, may be if you want to reduce the power, your area will increase and so on. So, these requirements are not independently controllable. If you try to optimize one you may make the other one worse, right.

So, the present trend is to standardize something called design flow means that steps that you need to follow to create a VLSI circuit or a chip. And as I said the present emphasis is number 1 on low power design and number 2 on increased performance.

(Refer Slide Time: 07:25)



So, here in this diagram I am showing 2 circuits side by side. So, on the left you have the first IC planar means it is laid down in a single plain this is called a planar IC. So, you can very easily see the connections the metals and the devices which are prepared, this was a very simple circuit. And on the right side you think of one of the; you see one of the modern processor chips the Intel Quad Core Nehalem series processor. So, here as I said you have something of the order of billion transistors packed in a single chip.

So, when you look at the layout in a very compacted way you see a colorful picture like this, where of course the different colors indicate the different layers of the circuit, right,

(Refer Slide Time: 08:23)



Ok, this is a very interesting plot, Moore's Law is a very important law in semiconductor design you can say. The persons who are into semiconductor design, they all know about Moore's Law. There was a person called Gordon Moore who as early as in the 1960s predicted some behavior about the growth in the semiconductor industry. So, what he had said at that time was that the number of transistors that you can put inside the chip would be increasing exponentially with time, with the number of years that pass.

So, there have been some refinements to this basic you can say law or postulate. So, what it is accepted more or less today is something like this it says that every 18 months or so the number of transistors in a chip single chip will get doubled. Now, there were people who had doubted this principle or law since quite a long time in the past. This said that well the devices are becoming smaller and smaller the transistors you are making smaller, so there will be a time, a time will come where you will not be able to make the devices any further smaller.

So, there will be a limit and beyond that Moore's Law will cease to exist. But actually what has happened till today is that, because of semiconductor fabrication advances we are able to fabricate bigger chips, in the process we have been able to sustain Moore's Law, we have been put more circuits in the chip. So, if you look at this graph, so on the x axis we are plotting year started from 1970 so here it shows up to 2015, and on the y axis you see the number of transistors which is in a log scale see 1000 here up to here it is

1 billion here it is 10 billion. So, you can see there is a straight line kind of a behavior which indicates exponential growth, and here the blue dots refer to the processors which are manufactured by the processor major Intel Corporation, the red dots are the processors manufactured by some other companies, ok.

So, Moore's Law as you can see, this has continued to hold and this trend is still a straight line behavior which indicates exponential growth over the years.

(Refer Slide Time: 11:30)



Well, the technologies that have made this possible are well CMOS. You may be knowing that CMOS is the most dominant technology today, with which we are manufacturing our VLSI chips, and the CMOS transistors are becoming smaller and smaller and smaller over the years. There is something called feature size which we talk about, that is roughly that is equal to this smallest feature or the transistor that you can fabricate.

Well, the state of the art CMOS technology today you can go down up to 22 nanometer, this is traditional CMOS fabrication. But there has been some very innovative kind of CMOS designs also, there is something called FinFET, where the gate drain and the source they are staked vertically instead of horizontally as in the traditional case, and in this way you can pack transistors in a smaller area.

So, today in the FinFET state of the technology you can go down to 14 nanometer. And many of the modern chips that are coming in the market, they are actually manufactured using this FinFET technology, ok. And the picture which is shown in the right, this is of course futuristic, this we do not have today, tomorrow quantum computer may come so you may be having a new technology the quantum technology.

(Refer Slide Time: 13:19)

VLSI Design Flow

- Standardized design procedure
 - Starting from the design idea down to the actual implementation.
- Encompasses many steps:
 - Specification
 - Synthesis
 - Simulation
 - Layout
 - Testability analysis
 - and many more

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

So, looking at the VLSI design flow again, so what is a VLSI design flow? VLSI design flow is nothing but a standardized set of design procedure. Means, here we specify the step by step procedure to be followed starting from our given specification, what you want to do, down to the actual hardware circuit, that you want to built or manufacture. This is the overall design procedure. And this typically encompasses many steps, just a few of them are shown here. Starting from the specification, you go through some steps called synthesis, simulation, layout generation, testability analysis, and there are many more steps in between, ok.

So, these steps have to be followed before we can actually get a design fabricated, or manufactured. Now, because of the complexity of design as I said that we need the help of computers. So, it is just beyond the capability of human being to carry out the design in a manual way the more.

(Refer Slide Time: 14:40)

The slide content is as follows:

- Need to use Computer Aided Design (CAD) tools.
 - Based on Hardware Description Language (HDL).
 - HDLs provide formats for representing the outputs of various design steps.
 - A CAD tool transforms its HDL input into a HDL output that contains more detailed information about the hardware.
 - Behavioral level to register transfer level
 - Register transfer level to gate level
 - Gate level to transistor level
 - Transistor level to the layout level

At the bottom of the slide, there is a footer bar with the following elements:

- IIT Kharagpur logo
- NPTEL ONLINE CERTIFICATION COURSES logo
- Hardware Modeling Using Verilog

So, you have to rely on computer aided design tools. And this computer aided design tools today are all based on some hardware description language. See just like the high level languages like C, C++ or Java we have a set of language, languages with which we can specify our hardware, and after specifying that we give it to our CAD tools and the CAD tools will be doing the rest for us, ok.

So, these tools are based on hardware description language as I said. These description languages they provide ways to represent designs. Not only the initial design, so as the cad tools translate or transform the designs they will represent this specification at the different steps of transformation as well, ok this we will see later. So, here is exactly what I was trying to mean. So, the CAD tool will transform some input which is specified in the hardware description language and generate an output which will also be a hardware description language, but the output will contain more detailed information about the hardware than the input.

Like some of the typical steps in the CAD tool transformation as follows. From the behavior you can translate or transform your design into a register level design, register transfer level which means from the behavior, you convert it into a form where you have the registers, counters, adders, multipliers, multiplexers, a design at that level, ok. Now, once you have done that may be the next step will be to convert each of those functional blocks into gate levels, then the gate level you convert to the transistor levels, then each

transistor you convert to the final layout level. So, once you have done this your design is ready for fabrication.

So, once you have carried out sufficient analysis and simulation to find out, that your design is meeting your requirements in terms of power consumption and delay, you can send it for fabrication.

(Refer Slide Time: 17:16)

Two Competing HDLs

1. Verilog
2. VHDL

Designs are created typically using HDLs, which get transformed from one level of abstraction to the next as the design flow progresses.

There are other HDLs like SystemC, SystemVerilog, and many more ...

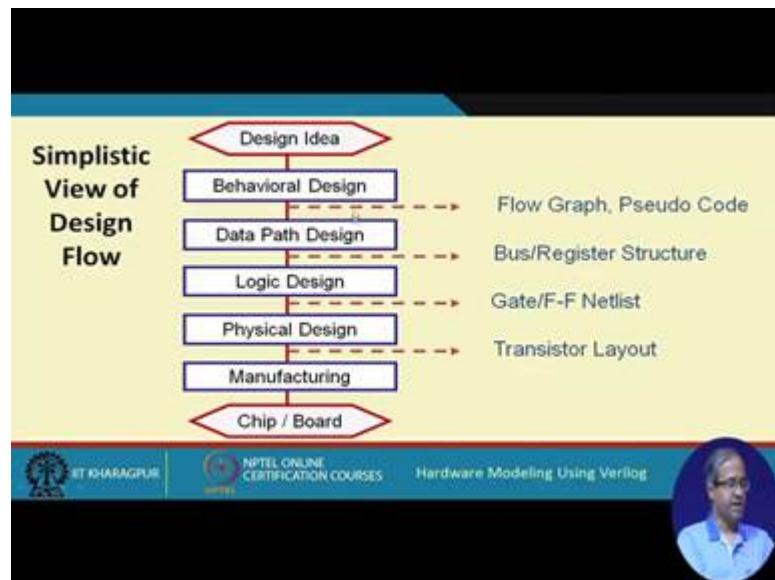
NPTEL ONLINE CERTIFICATION COURSES | IIT KHARAGPUR | Hardware Modeling Using Verilog

So, there are two computing HDLs today most popular: so one is Verilog other is VHDL. So, in this course as I have told you we shall be looking at the language Verilog. Now earlier as I have said that the designs are created using HDLs, so Verilog or VHDLs are very typical examples of these HDLs.

So, you can specify a design in either Verilog, or in VHDL and as the CAD tools transformed these designs from one level to the next, so the transform design is also expressed in similar kind of hardware description languages.

Now, these are not the only ones there are other hardware description languages as well, some of the popular languages are like SystemC, SystemVerilog and so on, but here in this course we shall be concentrating only on Verilog.

(Refer Slide Time: 18:23)



Talking about the design flow the simplistic view is as follows: starting from a design idea we have to finally come down to our chip design or a board design. So, we typically start with the behavioral design which is in the form of a pseudo code in a hardware description language or some kind of a flow graph notation. So, in the first level of translation we can convert the behavioral design into something data path design, which is the so-called register transfer level design, where the basic building blocks are buses, registers, multiplexers, adders, and so on. Then in the next step we convert it into logic design where you have gates and flip flops, then physical design where you have transistors, then we go for the last step of manufacturing, where the transistors are finally laid out and they are ready to be fabricated on silicon.

So, now we can send out designs, to the fabrication house where they can actually fabricate our chip for us, ok.

(Refer Slide Time: 19:39)

Steps in the Design Flow

- Behavioral design
 - Specify the functionality of the design in terms of its *behavior*.
 - Various ways of specifying:
 - Boolean expression or truth table.
 - Finite-state machine behavior (e.g. state transition diagram or table).
 - In the form of a high-level algorithm.
 - Needs to be synthesized into more detailed specifications for hardware realization.

So, talking about the steps in the design flow, the first was the behavioral design. As I have said here, we just specify the functionality of the design in terms of the behavior we do not say, how it is doing it, we just say what we want. Some examples, in the behavioral style, so we can express a Boolean expression (Refer Time: 20:02) function and the form of Boolean expression or in the form of a truth table. If it is a sequential circuit we can express it as a finite state machine: for example, just as a state transition diagram, or as a state transition table. Or you can even specify it at a very high level, just like a program written C or C++ or Java in a very high level of abstraction you can write a pseudo code just in the form of an algorithm.

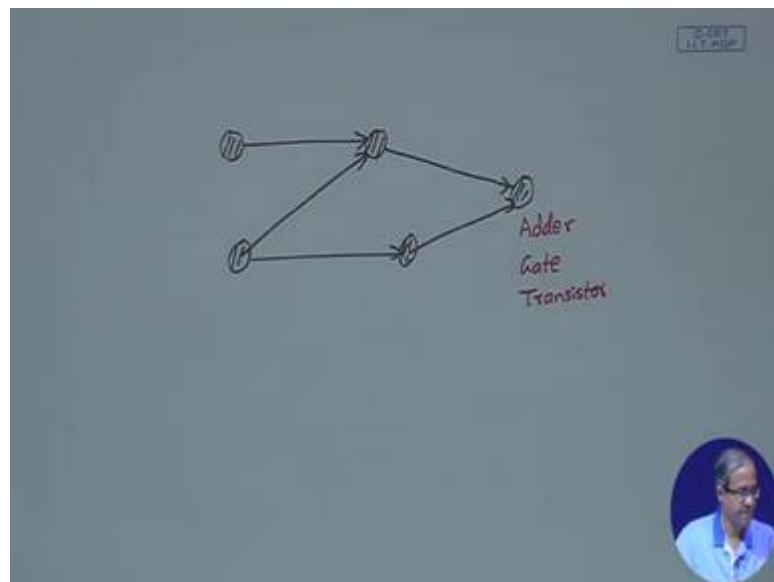
Now, during the design flow the steps, this behavioral design have to be transformed which is called synthesis, synthesized into more detailed specification as a result of which you will be finally getting your hardware.

(Refer Slide Time: 20:54)

- Data path design
 - Generate a netlist of register transfer level components, like registers, adders, multipliers, multiplexers, decoders, etc.
 - A *netlist* is a directed graph, where the vertices indicate components, and the edges indicate interconnections.
 - A netlist specification is also referred to as *structural design*.
 - Netlist may be specified at various levels, where the components may be functional modules, gates or transistors.
 - Systematically transformed from one level to the next.

So, data path design as I said here, we talk about register transfer level components like registers, adders, multipliers, multiplexers, decoders, bus etc. Now, when you call a netlist, netlist is nothing but some kind of a graph where the vertices indicate components and the edges indicate interconnects.

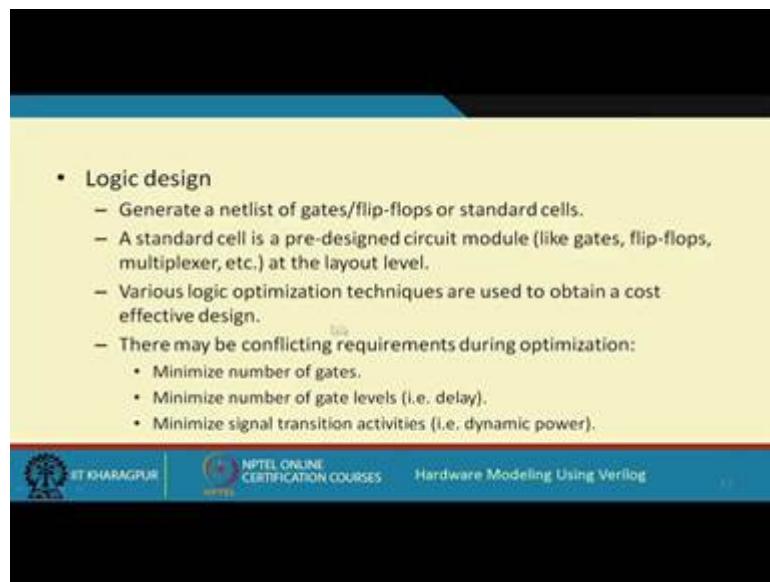
(Refer Slide Time: 21:20)



Like you think of a graph where there are some vertices and there are some edges. So, when you say netlist I have a number of such blocks, and I also specify how these blocks are interconnected.

Now, this blocks can be at various different levels. Now, this block can be very high level, high level like, it can be adder, it can be a gate, it can be a transistor for example. So, I can specify a netlist at various different levels of abstraction, right. So, whenever we specify a netlist this kind of a design specification is also referred to as a structural design. Now, this term we shall be using repeatedly when we will be discussing some details about Verilog in the next classes. So, netlist as I said, it can be specified at various levels, functional level, gate level, transistor level and so on. And during the synthesis process there will be systematically transformed from one level to the next, right.

(Refer Slide Time: 22:46)



- Logic design
 - Generate a netlist of gates/flip-flops or standard cells.
 - A standard cell is a pre-designed circuit module (like gates, flip-flops, multiplexer, etc.) at the layout level.
 - Various logic optimization techniques are used to obtain a cost effective design.
 - There may be conflicting requirements during optimization:
 - Minimize number of gates.
 - Minimize number of gate levels (i.e. delay).
 - Minimize signal transition activities (i.e. dynamic power).

Now, when you come to the logic design level, now here we have here again a netlist but now your blocks are gates and flip flops, or something called standard cells. Standard cell like, what is standard cell we should discuss it later. Well a standard cell basically is a pre designed circuit module like, it can be gates, and flip flops, small circuit like a multiplexer, whose layout is already given to you. So, you have this standard cell already present in a library, so in your design if you want you can pick up one of those standard cell you can put them in your layout directly, ok. In that way you create your layout and this standard cell library contains the most commonly used small functional modules in a highly optimized layout form, fine.

And in this type of logic design, various logic optimization techniques are used to minimize your design, to create a so-called cost effective design as far as possible,

ok. Now, as I said earlier that during this process there can be some conflicting requirements, like minimizing number of gates, minimizing delays, that means, number of gate levels, minimizing powers, which means number of signal transitions that are taking place at the outputs of the gates. Now, these requirements are often conflicting, if you want to minimize one of these, possibly the others can increase, right, so this is something that you have to keep in mind.

(Refer Slide Time: 24:27)

- Physical design and Manufacturing
 - Generate the final layout that can be sent for fabrication.
 - The layout contains a large number of regular geometric shapes corresponding to the different fabrication layers.
 - Alternatively, the final target may be Field Programmable Gate Array (FPGA), where technology mapping from the gate level netlist is used.
 - Can be programmed in-field.
 - Much greater flexibility, but less speed.

And lastly physical design and manufacturing: here, the final layout that is to be sent for fabrication is generated. Now, at this level the layout will contain a large number of regular geometric shapes, because ultimately when you are going for fabrication, you are actually fabricating some patterns on the surface of silicon: metal layer, poly silicon layer, diffusion layer, and all of them have regular polygonal shapes, that polygons typically rectangles. So, at this level your specification will consist of a very large number of such regular polygonal shapes.

Or suppose you do not want to go for a chip to be fabricated as an alternative, you can also go for something called a field programmable gate array, or FPGA, where from the design you can directly program the device which can be done in field in your laboratory, and as a result you can have much greater flexibility. But as compared to a chip you are fabricating, here the speed will be less, ok. This is the trade off you have to realize.

(Refer Slide Time: 25:52)

Other Steps in the Design Flow

- Simulation for verification
 - At various levels: logic level, switch level, circuit level
- Formal verification
 - Used to verify the designs through formal techniques
- Testability analysis and Test pattern generation
 - Required for testing the manufactured devices

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

So, there are some other steps in the design flow also, these are not the only ones. Like simulation is very important step for verification. Like for example in this course we shall be extensively using simulation to check and verify the Verilog modules that we will be writing. Ok, we will also be informing you telling you how to do this simulation so that you can do or carry out the simulation yourself, right.

So, this simulation can be carried out at various levels of specification, logic level, transistor level, circuit level and so on. There is a step called formal verification, of course this will be beyond the scope of this course, where using some mathematical and formal techniques, you can check whether your designs are meeting the specifications or not. And again testability analysis test pattern generation is also very important. So, when you manufacture or designs some hardware, you will also have to test whether your final manufactured hardware is working correctly or not. Again this step is beyond the scope of this course we shall not be also talking about testability and testing here, ok.

So, with this we come to the end of this first lecture. In this lecture we have basically tried to give you an overview about what are the things we are expected to cover in this course and some basic concepts of VLSI design flow. Because understanding the VLSI design flow the process that is embedded there in, it will allow you to have a better understanding of how you can create a design using a so-called HDL, it can be either Verilog or VHDL as I said, so that the final hardware that will be generated in the process can be better in some

sense. So, over the course of this lectures that will follow, we shall be trying to address these issues.

Thank you.

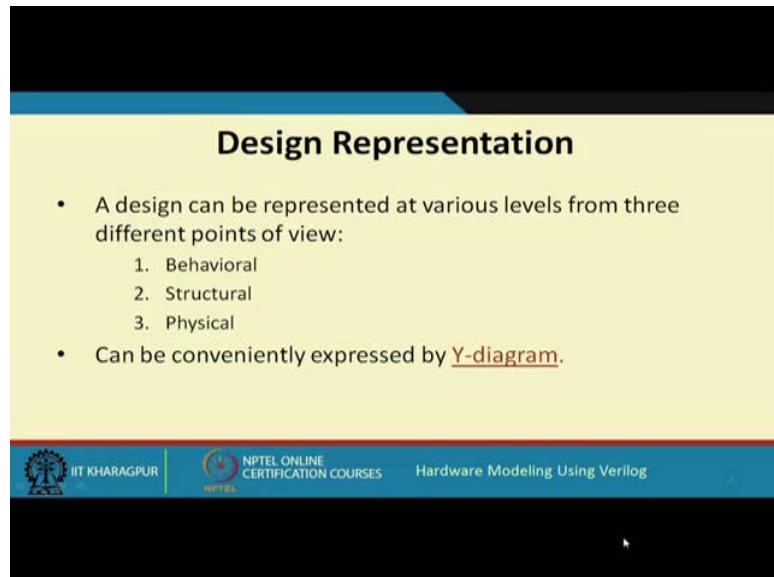
Hardware Modeling using Verilog
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 02
Design Representation

In this lecture we shall be talking about Design Representation. Now, earlier I had said that you can create a design using a hardware description language, Verilog with respect to this course. Suppose we create a design in some hardware description language like Verilog. Now, the question arises at what level do it typically create the design, or how do we create? How do we visualize a design?

So, when we talk about design representation, what you mean is that, given a design, how we can visualize the same design from different angles? Now, the way we shall be presenting here is that there are three distinct ways or three distinct angles from where you can visualize the same design and the implications will be a little different, let us see.

(Refer Slide Time: 01:22)



The slide has a black header bar and a blue footer bar. The main content area is yellow. The title 'Design Representation' is centered in bold black font. Below it is a bulleted list:

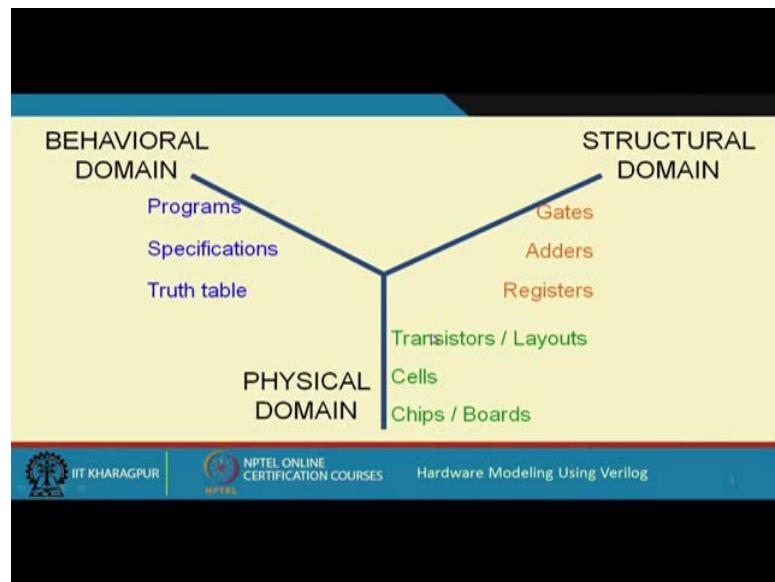
- A design can be represented at various levels from three different points of view:
 1. Behavioral
 2. Structural
 3. Physical
- Can be conveniently expressed by Y-diagram.

The footer bar contains the IIT Kharagpur logo, the text 'NPTEL ONLINE CERTIFICATION COURSES', and 'Hardware Modeling Using Verilog'.

So, design representation as I said this talks about how we represent a design. A design is nothing but you can think it as a black box that is trying to perform its intended operation. You have designed that block to carry out certain operation and that is your design, ok. Now, design representation, we are considering at three different levels, one you are calling it as behavioral, structural and physical.

Now, with respect to Verilog coding these terms behavioral and structural will be coming repeatedly and we will be understanding the exact differences and how we can code behavioral and structural designs in Verilog in a very efficient way and of course, the third representation is physical. Now, there is a very interesting way to just express this design views or viewpoints, it is with respect to something called a Y-diagram.

(Refer Slide Time: 02:39)



Now, this name has come because it has a shape of a Y. So, you think of an imaginary Y with the three design representations, in the three directions. Let us say, this is our behavioral domain, this is our structural domain and this is our physical domain. Now, just along each of these domains, like for example, in the axis of behavioral domain. Here we are talking about a design in terms of its behavior, like in a very high level case, you can think of an algorithm or a program, you can think of a specification, you can think of Boolean expressions, you can think of truth tables, state diagram and so on, these are all examples of behavior.

Now, when we talk about structural domain, now well we had explained in the last lecture what is a netlist and netlist is nothing but some building blocks and their interconnections. Now, this building blocks can define at various different levels, depending on that our structural domain specifications can vary. So, it can be at the register transfer level, it can be at the functional level, it can be at the gate level, transistor level and so on. And talking about the physical domain, here you are actually talking about the implementation. Like

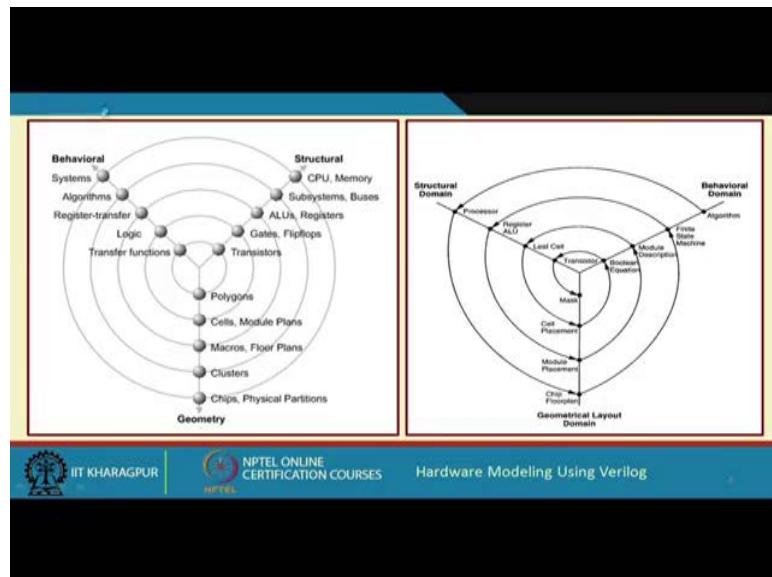
in the ultimate case, when the final design is complete, we would be getting our chips or a printed circuit board.

Before that our designs will be placed on a silicon layout in terms of cells. Now, what will those cells contain? They will contain some transistors, which will have some layout. So, these are all implementation or physical design related things, that our chip, the cells that are put on a chip and the transistors that make a cell. So, here we are looking from an angle which talks about the actual implementation, ok.

So, in one side was the behavior, one side was the netlist view and the other side was the actual physical implementation view. So, these are the three viewpoints, now there is some interesting ways to look at these Y, let us look at it once more from this point of view.

Here we have showed it in a slightly different way, behavior, structural, geometry and you see, we have drawn some concentric circles. The outer most circle indicates the most abstract design, so at the behavioral level, this outside circle indicates systems, is like in the highest level, my system may consist of two processors, one memory and one I/O module, this is my highest level design of the system.

(Refer Slide Time: 05:55)



So, this is my behavior, so, this in the structural level will be a netlist that consisting of interconnection of CPUs and memories as I have said the example and in the physical level, this will consist of some chips, a collection of chips one may be processor, one may

be memory and so on, ok. So, as you go to the inner circles you are carrying out synthesis, you are refining your design like at the next level your behavior is your algorithms. Here you have subsystems and buses, here you have some clusters, next level you have register transfer level blocks, you have ALU's and registers netlist level and here you have macros and floor plans.

Next level inside you have logic cells, gates and flip flops. So, here you have connection of gates and flip flops, here you have cells module plans and at the lowest level you have the transfer functions, transistors and polygons. So, as you go from the outer most circle towards the inner most circle, from all the three viewpoints your design is getting refined, from very high level or abstract, see from a very abstract level in the behavior, you will say the well I want a computer system.

In the structural level, you will say that well I need two processors and one memory more block. Now in the physical level you will say that well on the board I need four chips, these are the different ways to look at the same thing. So, as we move towards the centre of the circle each of these blocks will get further refined, they will get more detailed out and will be having some kind of a top down design process.

Now, if you think of it well we normally do not do it this way that at one level, we have all the three viewpoints, then we go to the next level, again look at all the three viewpoints. So, we really do not work exactly in that way, rather we work in an alternate way which is shown in the diagram on the right. So, here I am showing the behavior axis here, structural here and geometric here or the physical here. Now you see; here I am giving an example, we start with an algorithm at the behavioral level, so, highest level is algorithm.

So, we carry out some transformations in a systematic way which is represented you see as a helix here, from the outside you slowly move towards the inner most point of this of this structure. So, from the algorithm; you translate into processor which you translate into a chip floor plan and each of the blocks in a chip floor plan you translate into finite set machines; they will be translated into registers and ALU's, they will be translated the physical level into modules which will be placed module placement.

Then each of these modules; will be having a description at the behavioral level module description now each of the modules will be designed as a netlist we call them as a leap cell, they will be placed cell placement. Now, each of the cell will be expressed as Boolean

expressions they will be realize using transistors and finally, they will be realized using the layout masks.

So, you see this is, this helical structure is more natural in terms of the steps that a designer actually follows. So, although that concentric circle is a good way of representing, but the actual process that you follow is this helix, from the outermost point we slowly move towards the centre of the Y, ok, this is the so-called top down design process.

(Refer Slide Time: 10:07)

Behavioral Representation

- Specifies how a particular design should respond to a given set of inputs.
- May be specified by:
 - Boolean equations
 - Tables of input and output values
 - Algorithms written in standard HLL like C
 - Algorithms written in special HDL like Verilog or VHDL

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

Ok, so let us again come back to the behavioral representation, here we shall be showing you some examples in Verilog. So, just to repeat in the behavioral representation we specify, how a particular design should work, that means, given a set of inputs, what should be its output. So, here we are not telling that exactly how the design has to be implemented. Some examples of behavioral representation are Boolean expressions, truth tables table of input and output values.

So, you can write algorithms in some standard high level language like C that is also behavior or algorithms written in hardware description languages like Verilog or VHDL these are all examples of behavioral representation.

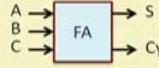
(Refer Slide Time: 11:01)

Behavioral Representation :: Example

Full Adder:

- two operand inputs A and B
- a carry input C
- a carry output Cy
- a sum output S

- Express in terms of Boolean expressions:
 $S = A \cdot B' \cdot C' + A' \cdot B' \cdot C + A' \cdot B \cdot C' + A \cdot B \cdot C = A \oplus B \oplus C$
 $Cy = A \cdot B + A \cdot C + B \cdot C$



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

Now, let us take an example a simple example of a full adder. So, a full adder we will have two inputs A and B and a carry input C. So, it will be generating a carry output and a sum like this; A, B, C are the inputs and S and Cy are the outputs. Well, now I am not trying to explain how a full adder works; you already know the basic definition of full adder; I am just writing the logic expressions of the sum and carry. So, the sum and carry expression in terms of A, B, C will be this: $A \cdot B' \cdot C' + A' \cdot B' \cdot C + A' \cdot B \cdot C' + A \cdot B \cdot C = A \oplus B \oplus C$ and $Cy = A \cdot B + A \cdot C + B \cdot C$; which is nothing but A exclusive OR B exclusive OR C.

$$S = AB'C' + A'B'C + A'BC' + ABC = A \oplus B \oplus C$$

$$Cy = AB + AC + BC$$

So, it is the exclusive OR of the three inputs that is sum and carry is nothing but AB OR AC OR BC. So, this is one way of expressing the behavior of a full adder, because here you are not specifying what kind of gates to use, how many gates to be used and so on. We are just writing down the expressions and we are telling that well this is your sum and this is your carry, this will be the behavior, right, this is an example.

(Refer Slide Time: 12:33)

- Express in Verilog in terms of Boolean expressions

```
module carry (S, Cy, A, B, C);
    input A, B, C;
    output S, Cy;
    assign S = A ^ B ^ C;
    assign Cy = (A & B) | (B & C) | (C & A);
endmodule
```

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

Now, this is in terms of Boolean expression the same thing you can express in Verilog as follows well I have not talked about the syntax of Verilog.

So, possibly this is your first look at a Verilog program, let us see what it contains. The syntax is somewhat similar to the language C, here every statement ends with a semicolon. This is like a function in C, this is called the module. So, Verilog program will consist of one or more modulus. So, it starts with this keyword module, it ends with endmodule, this is, this module is followed by the name of the module and the parameters. The parameters are well in terms of the hardware these are nothing but the input and the output ports of that block.

Now for a full adder, the inputs are A, B, C and the outputs are carry out and sum, right. So, this S, Cy, A, B, C are the five parameters, we declare, A, B, C as input like this, input A, B, C, output S, Cy. So, that the CAD tool will know that which are your input pins, input signals which are your output signals and there is an assign statement available in Verilog, here we are showing it using assign you can directly write down the Boolean expression.

This hat (\wedge) is the expression for Exclusive OR this indicates XOR. So, sum is A XOR B XOR C and this ampersand ($\&$) is AND, bar (\mid) is OR, carry is AB OR BC OR CA. So, you see in Verilog you can express Boolean expression, which is behavioral specification very concisely in this way using the assign statement, ok.

(Refer Slide Time: 14:42)

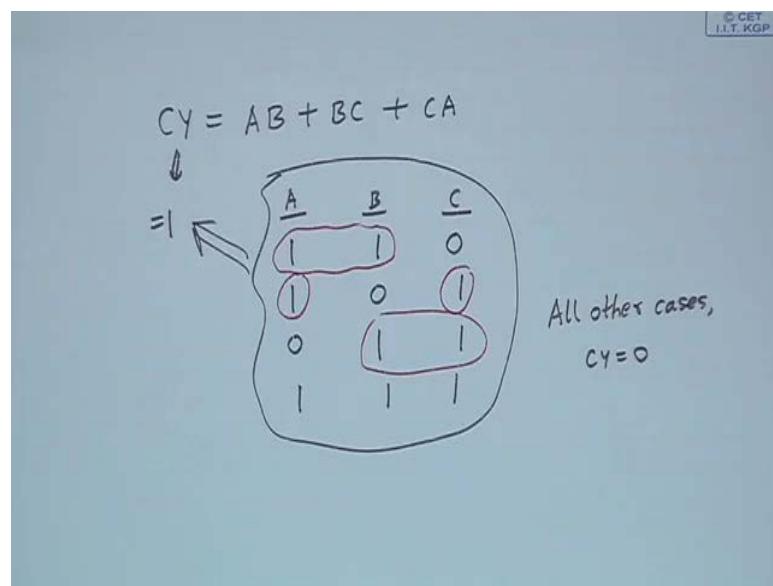
- Express in Verilog in terms of truth table (only Cy is shown)

```
primitive carry (Cy, A, B, C);
  input A, B, C;
  output Cy;
  table
    // A   B   C       Cy
    1   1   ?      :   1 ;
    1   ?   1      :   1 ;
    ?   1   1      :   1 ;
    0   0   ?      :   0 ;
    0   ?   0      :   0 ;
    ?   0   0      :   0 ;
  endtable
endprimitive
```

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

So, this is one way to specify a behavioral specification in Verilog. Now, this same design, the full adder I am showing only one of the output the carry, well just if we recall the carry is nothing but AB OR BC OR CA.

(Refer Slide Time: 14:55)



So, you tell me when the carry will be 1? The carry will be 1 for several conditions of A, B and C, when either A and B is 0, but C is 1, A, B is 1, C is 0, A and C is 1, B is 0 or B and C is 1, A is 0 or all these three are 1.

So, for these four cases my carry will be 1, for all other cases carry is 0. So, this I can express or as a as a truth table. So, how do I express? Well I need not have to specify all the rows; I can simply say that at least two must be 1; A B, A C or B C, at least two. If it is 1; then the carry will be 1; this is what you are specify here, you see; in this truth table.

So, this is again a way to express a behavioral specification in Verilog in terms of the truth table. So, there is a keyword called primitive. So, we use this primitive keyword here; so, I am only showing for carry. So, there are four parameters this is the output A, B, C are the inputs. So, I declare the input and output signals. So, I use a keyword table; so, here this is a comment just for showing that these are the values of A, B, C and this is Cy this is a comment line, just for convenience.

And the way you specify the input is, you specify 0 or 1 and question mark means don't care and colon separates inputs and outputs. The first line says; if A and B there 1, but C is don't care; then carry is 1; like you see here I means if A and B are 1, C is 0 then also carry will be 1, but if A and B are 1, C is 1; then also carry is 1. So, actually C is a don't care; it does not depend on C. So, this we can write in a concise way using the don't care notation.

Similarly, if A and C are 1; B is don't care or B and C are 1; A is don't care; then also carry will be 1. But on the other cases at least two of the inputs are 0; A B is 0 or A C is 0 or B C is 0; other is don't care; then carry will be 0. So, you see; if we use don't care then instead of the eight rows of the truth table; here we require only six, so our specification may be shorter, right.

(Refer Slide Time: 18:08)

Structural Representation

- Specifies how components are interconnected.
- In general, the description is a list of modules and their interconnection.
 - Called *netlist*.
 - Can be specified at various levels.

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

Now, let us move on to structural specification; now let us structural specification as I said that; it actually specifies how some modules are interconnected. So, when we say that I am defining something in a structural way; which means I will have to specify the modules and I will also have to tell how their actual interconnected, ok. Let us see an example on this. So, earlier we mentioned that; this kind of structural representation is also called the netlist. And netlist can be specified at various levels at a very high functional level, gate level, transistor level and so on.

(Refer Slide Time: 18:58)

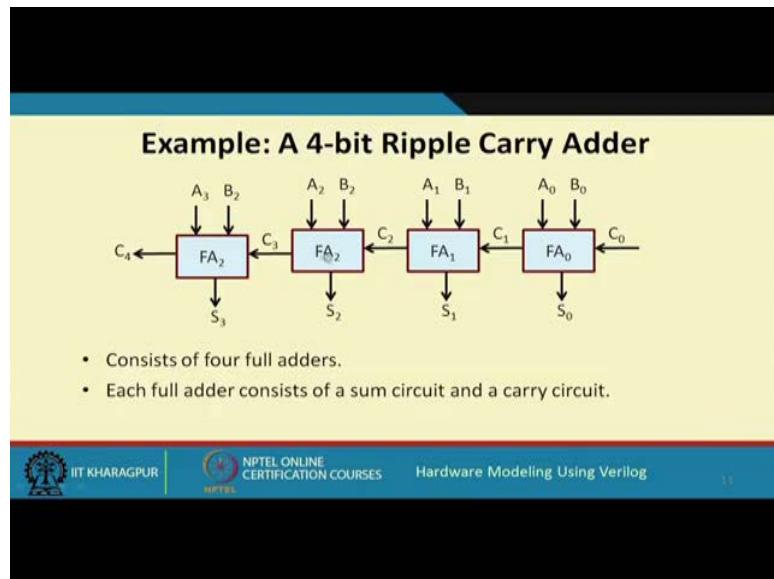
- At the structural level, the levels of abstraction are:
 - The module (functional) level
 - The gate level
 - The transistor level
 - Any combination of above
- In each successive level more detail is revealed about the implementation.

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

This will see later slowly. So, structural level; levels of abstraction is also we talked about earlier I mention gate level, transistor level or can be hybrid design also.

Some of the blocks can be the gate levels, some of the blocks can be at higher level, ok. So, as you move down; more and more details I revealed. Like a netlist; which is at a functional level; multiplexers, decoders, adders their number of blocks will be less. So, as you move down to the gate level netlist equivalent netlist, number of gates will be much larger. As you move down again to a transistor level netlist; number of transistor shall be even larger. So, as you move down; your size of the netlist will be increasing gradually. Let us take an example to illustrate structural design.

(Refer Slide Time: 19:53)

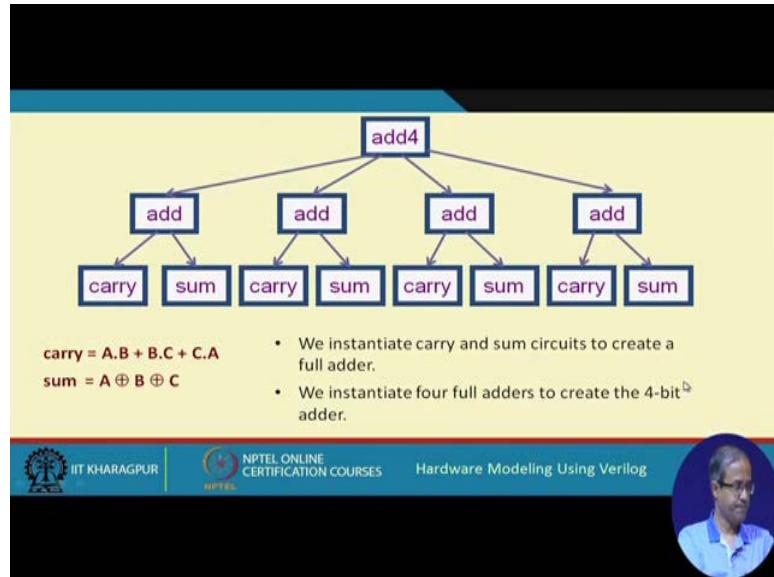


Now, this is a circuit which you must be familiar with; this is a 4-bit ripple carry adder. So, what we have done? Just to recall. So, 4-bit ripple carry adder consists of 4 full adders. So, I want to add two numbers A0, A1, A2, A3 and B0, B1, B2, B3; they are all 4 bit numbers. So, I feed the corresponding bits to the 4 full adders and C0 is my carry in. The carry out from the first full adder will be going as carry into the next carry out from here you will go to the carry in of the next; similarly, here and C4 will be the final carry out and S0, S1, S2, S3; will be the final sum.

So, if you look inside the full adders; where just now sometime back we saw, the behavioral specification of a full adder; we saw that there is a sum part, there is a carry part, there will be one circuit which will be computing the sum; there can be another circuit

disjoint or sheared may be which will compute the carry. So, each full adder we will consist of a sum circuit and a carry circuit.

(Refer Slide Time: 21:13)



So, conceptually speaking we show it like this. So, as if we have a 4-bit adder; we call it as a add4, 4-bit adder. 4-bit adder consists of 4 full adders add add add, this are full adder. Each full adder consists of a carry circuit and a sum circuit; carry circuit, sum circuit. So, this is how we can describe a circuit in a hierarchical way and also structural way. Like every adder will be a combination of carry and sum and this 4-bit adder will be a combination of 4 full adders. Now, this carry and sum, we already know these are the expressions, ok. So, let us see in Verilog how we can do this.

(Refer Slide Time: 22:03)

```
module add4 (s, cy4, x, y);
    input [3:0] x, y;
    output [3:0] s;
    output cy4;
    wire [2:0] cy_out;
    add B0 (cy_out[0], s[0], x[0], y[0], ci);
    add B1 (cy_out[1], s[1], x[1], y[1], cy_out[0]);
    add B2 (cy_out[2], s[2], x[2], y[2], cy_out[1]);
    add B3 (cy4, s[3], x[3], y[3], cy_out[2]);
endmodule

module add (cy_out, sum, a, b, cy_in);
    input a, b, cy_in;
    output sum, cy_out;
    sum s1 (sum, a, b, cy_in);
    carry c1 (cy_out, a, b, cy_in);
endmodule
```

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

So, here I am showing the top level module of a 4-bit carry look ahead adder. So, here the parameters are specified here; well here I am not going into the detail because all these details we shall be explaining again later. I am just showing you; how this Verilog code looks like. Just one thing you see here; here I have declared x and y are the inputs and this x and y input; I have declared as a vector 3 colon 0 means; it is a four-bit number, but the bits are number from the most significant side, 3, 2, 1, 0; this is the way to specify in Verilog 3 colon 0.

Carry-in is a single carry in; similarly output sum is also 4 bit; 3 to 0. See; we here let us show here I mean say we have an adder; we have also defined a full adder; add is a full adder. Full adder has parameters carry-out, sum and the three inputs. So, you see here in the 4-bit ripple carry adder; I have made four copies of this full adder, this is called instantiation.

Well in a C program; when you call a function. So, the control goes to the function; the function gets executed and again we come back. But in terms of hardware; if I call a full adder four times it means I am using four copies of the full adder, this is called instantiation. So, here four copies of the full adder will be instantiated and I am giving different names B0, B1, B2, B3 to the four copies and these parameters the way I have given the name; this defines the interconnection.

Like here for example, for B1 the carry-out Cy_out 0 this will be Cy_out 0. So, this will be connected to this Cy_out 2; Cy_out 1 will mean for B0, the carry-out will be connected to the carry-in of the next, for B1 the carry-out will be connected to the carry-in of the next, for B2 the carry-out will be connected to a carry-in of the next and so on because this is your carry-in the last parameter, right. And here this is a hierarchy design the full adder also we have defined in a structural way, we have put a sum and a carry. Sum is a module, carry is a module, we have instantiated them.

(Refer Slide Time: 24:58)

The screenshot shows a Verilog code editor with two modules defined:

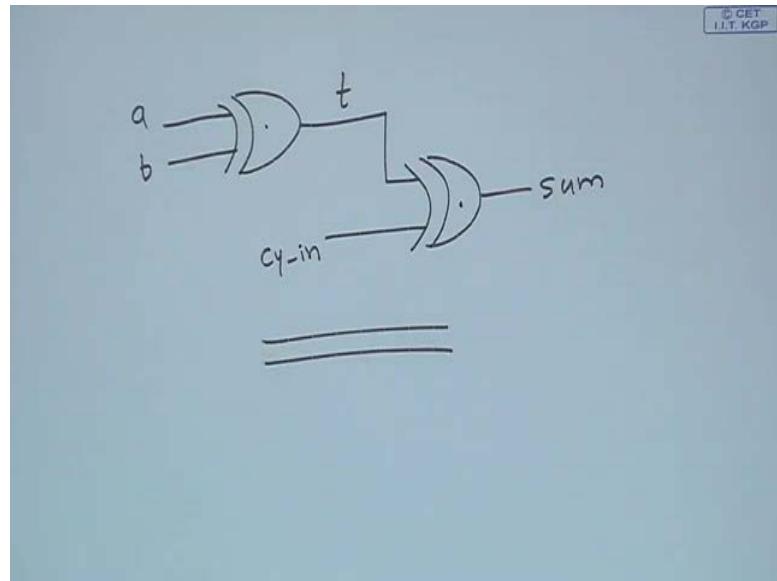
```
module sum (sum, a, b, cy_in);
    input a, b, cy_in;
    output sum;
    wire t;
    xor x1 (t, a, b);
    xor x2 (sum, t, cy_in);
endmodule
```

```
module carry (cy_out, a, b, cy_in);
    input a, b, cy_in;
    output cy_out;
    wire t1, t2, t3;
    and g1 (t1, a, b);
    and g2 (t2, a, c);
    and g3 (t3, b, c);
    or g4 (cy_out, t1, t2, t3);
endmodule
```

At the bottom of the slide, there are logos for IIT Kharagpur and NPTEL, along with the text "NPTEL ONLINE CERTIFICATION COURSES" and "Hardware Modeling Using Verilog".

And sum and carry similarly, can be described like this. So, I have defined this in a structural way; not in a behavioral way you see, sum, this A, B, C in the inputs; sum is the output and t is a temporary wire.

(Refer Slide Time: 25:27)



This XOR is a gate; what does it t, a, b means? t, a, b means the first one is the output second two are the inputs; which means that I have an XOR gate for the output is t and the inputs are A and B, the first line indicates this. Then next line indicates; there is another XOR sum, sum is the output, inputs as t and Cy_in. So, t is there this is another XOR gate, this is another input Cy_in and you generate sum. So, you actually specify this circuit; this is structural because you have specified two gates, XOR gates and how they are interconnected, right. So, this is a structural description of a sum.

Now, these gates XOR or for carry, this AND, OR these gates are already there as a part of the Verilog language. So, this we shall be studying slowly. So, I have just shown you example here just to show you how a Verilog module looks like. So, in the carry block in a similar way; we need four gates g1, g2, g3, this will do a AND of AB, AND of AC and AND of BC and the outputs are t1, t2, t3 and finally, you do a OR of t1; t2; t3 and out.

And you see the number of inputs to the gates you can vary like in the last case, the first one will be the output and the remaining three will be the inputs. So, this will be automatically taken like this, right.

(Refer Slide Time: 27:03)

Physical Representation

- The lowest level of physical specification.
 - Photo-mask information required by the various processing steps in the fabrication process.
- At the module level, the physical layout for the 4-bit adder may be defined by a rectangle or polygon, and a collection of ports.
- At the layout level, there can be a large number of rectangles or polygons.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog | 15

Now talking about the physical representation which is the lowest level; here as I said that when you finally, fabricate the chips; you define a large number of various polygonal shapes; for the different layers of fabrication, ok.

So, these are this specification of so-called photo masks that are required in the fabrication process. Now this 4-bit adder is the way we look at it; from the function or the structural point of view, but from the physical point of view there will be a large number of rectangles or polygons.

(Refer Slide Time: 27:48)

- Partial physical description for 4-bit adder in Verilog

```
module add4;
    input x[3:0], y[3:0], cy_in;
    output s[3:0], cy4;
    boundary [0, 0, 130, 500];
    port x[0] aluminum width = 1 origin = [0, 35];
    port y[0] aluminum width = 1 origin = [0, 85];
    port cy_in polysilicon width = 2 origin = [70, 0];
    port s[0] aluminum width = 1 origin = [120, 65];

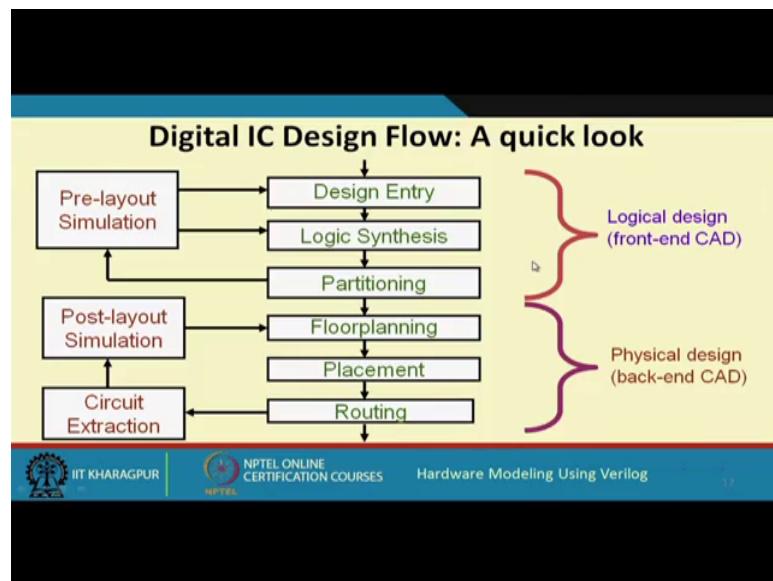
    add a0 origin = [0, 0];
    add a1 origin = [0, 120];
endmodule
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog | 16

So, this also can be specified in Verilog; I am just showing you an example this is not the complete description, just a partial description for the adder. Like you can actually define some coordinates 0, 35 is a coordinate; this is an aluminum wire of width one unit, this is the poly silicon wire of width 2 units and so on. So, in this way you can define various wires of different metals or materials, you can specify the widths basically their rectangular shapes, right.

Rectangles, there will be a large number of such rectangles will define you such specification. So, I am just showing you these primitives are also means available in Verilog. So, when you go down to the layout level; the same Verilog language can be used to specify your layout, ok. This is what I meant by saying is that you have a hardware description language and as the process of synthesis continues you transform one version of Verilog to another version of Verilog, that another version of Verilog to another define version of Verilog in that way you proceed and at the final step you get a description which is your final layout description, ok.

(Refer Slide Time: 29:09)



So, just a quick look at the digital IC design flow once more; starting from the specification, you design entry or this may be your specification in your Verilog or VHDL; logic synthesis and there are some steps where you are actually going for the physical design; this is called physical design. You go for partitioning, floor planning, placement routing steps like this.

And in the first steps; you go for design entry, logic synthesis, partitioning. These steps are typically called frontend design or logical design and these are called physical design or backend design. And there is some feedback connection also because at some step; you may find that it is not working properly to do simulation, you may have to go back and make some changes.

Like here also you can do some circuit extraction and simulation; you say that your delay is not coming proper, you may have to go back and make some changes. So, just a very rough view of the whole process of design; frontend design and backend design; this entire process has to be traversed by a designer to actually design a circuit in terms of the final layout description, starting from the behavior.

So, with this we come to the end of the second lecture. Well, in this lecture we have basically talked about design representations; the three ways you can visualize a design and we have seen some examples in Verilog; how Verilog can be used to capture the design at the different levels of abstraction; different views, behavioral structural and physical.

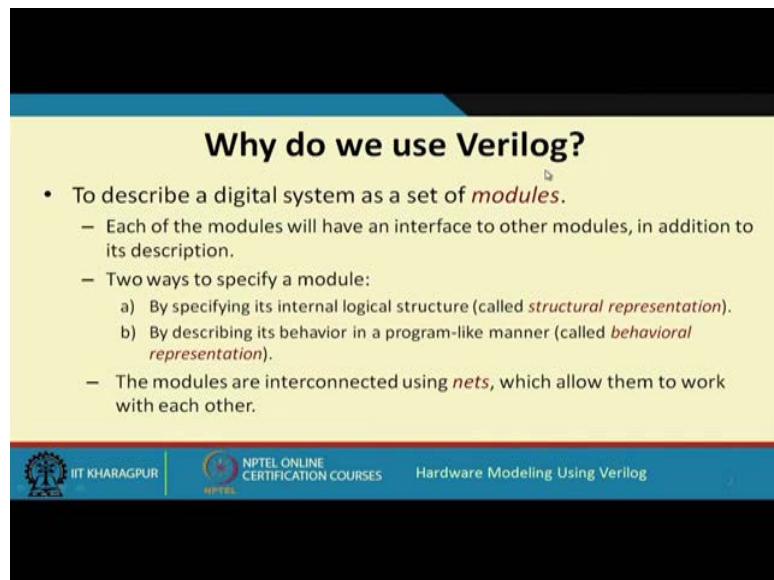
Thank you.

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

Lecture - 03
Getting Started With Verilog

Welcome back. In this third lecture, we shall be getting the first feel about Verilog: how to use it, and how to simulate designs using Verilog. As you can see that the title of this lecture is getting started with Verilog. So, let us see.

(Refer Slide Time: 00:46)



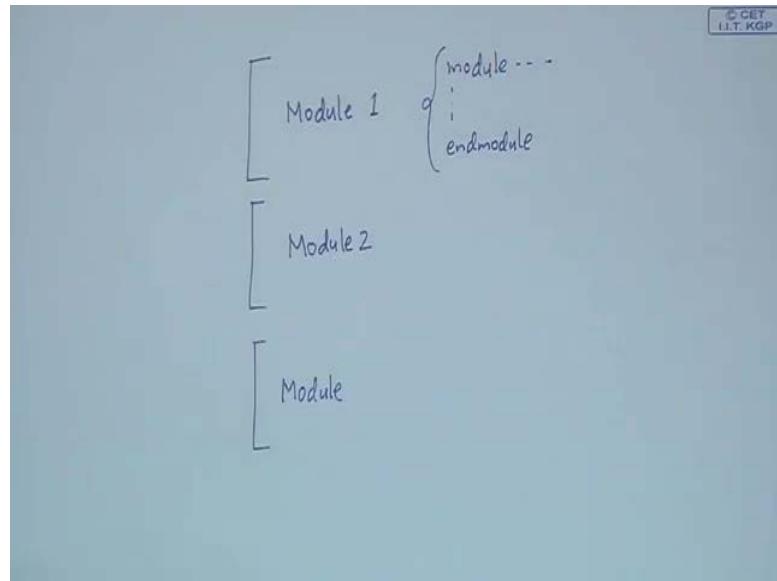
Why do we use Verilog?

- To describe a digital system as a set of *modules*.
 - Each of the modules will have an interface to other modules, in addition to its description.
 - Two ways to specify a module:
 - a) By specifying its internal logical structure (called *structural representation*).
 - b) By describing its behavior in a program-like manner (called *behavioral representation*).
 - The modules are interconnected using *nets*, which allow them to work with each other.

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

So, in our last lecture we have already seen some example; Verilog programs. So, I have just mentioned, the concept of Verilog modules. So, any description of a digital system that we are trying to create description, this has to be in terms of a set of modules. So, whenever you write a design description in a Verilog; so, it will be looking as follows.

(Refer Slide Time: 01:22)



There will be several modules; let us say there are three modules. Now, each of this modules will be starting with the module keyword and it will end with the endmodule. The same thing will be for the other modules also. So, this all the modules taken together will define your complete specification of the hardware; whatever you are trying to arrive at, right.

So, if you want to design some system or a circuit in terms of multiple modules; like in the earlier example, we saw in the last class where we talked about a 4-bit ripple carry adder. So, we saw we had multiple modules; a carry module, sum module; we use those modules to create a full adder module add. Then we used four copies of the add module to create a ripple carry adder module and so on. So, you can instantiate a module from other modules. So, there is no concept of calling one module from another; calling means instantiation in terms of hardware. So, if you call for example, from module 1; if you call module 2, there will be a copy of module 2, which will be inserted in module 1, ok.

Because you have to remember that ultimately we are designing some hardware. So, whenever invoking something one module is invoking another module; that is something called instantiation. It means create a copy, if I caller invoke two times there will be two copies created, ok, fine. So, these modules when you describe them; they will as you seen, they have some interfaces means in terms of the parameters. So, the way we specify the parameter values; the variables they will specify how these modules will be

interconnected, right. Now, we have also seen that we can specify a module in two different ways; you saw some examples in the last lectures.

So, we can either specify its internal logical structure; which is called structural representation. This we had already seen for the sum and carry modules of the full adder and also the full adder module and also the 4-bit ripple carry adder module. Those are all examples of structural description, where we had some modules; we also specified how the modules are interconnected; this is the so-called structural representation.

Secondly, we can also specify a module by describing its behavior. So, you recall the earlier example we took; where the carry and some expressed in the form of Boolean expression; that was a behavioral representation, right, fine. So, when we talk about structural representation; we also talk about the inter connections of the modules, they are interconnected by something called nets. Nets are just like wires the output of one module is connected to the input of another module, ok.

So, we shall see; later what are the different kinds of nets, different kinds of wires that are there in Verilog. We shall see this in detail later, ok, all right. So, what next?

(Refer Slide Time: 05:26)

What next?

- After specifying the system in Verilog, we can do two things:
 - a) Simulate the system and verify the operation.
 - Just like running a program written in some high-level language.
 - Requires a *test bench* or *test harness*, that specifies the inputs that are to be applied and the way the outputs are to be displayed.
 - b) Use a synthesis tool to map it to hardware.
 - Converts it to a netlist of low-level primitives.
 - The hardware can be *Application Specific Integrated Circuit* (ASIC).
 - Or else, it can be *Field Programmable Gate Array* (FPGA).

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

See, we have specified the system in Verilog; let us say. So, we have already specified our design in Verilog; now we can do one of two things, first is we can simulate the Verilog code and we can verify whether it is working correctly or not. This is just like running a

program which you do in a high level language like C. So, whenever we write a program; we immediately run the program, give some input data and check whether the output is coming correctly or not, right; this is how you verify.

So, here simulation means exactly a very similar thing. So, we simulate or we say that you run this Verilog description; with some input data that I have given and you tell me what the output is coming, ok. So, this simulation is just like running a program as I had said; written in some high level language. This requires us to specify the input and how we do it? We do it by writing something called test bench or test harness.

So, what is the test bench? Test bench is another Verilog module; which will actually be generating the inputs that will be applied to the module that I want to test, ok. So, here we will see; that how we can write this test benches. Test bench or test harness; specifies the inputs that have to be applied to my system which I have written in Verilog and I want to also see the outputs, ok. This is the one thing; I may want to do after I write this specification.

Now, you see as part of this course most of you will be doing exactly this. So here you are learning how to write or means how to specify some designs in Verilog. You will be specifying some design, you will be simulating and you will be trying to verify whether your design is working correctly or not, ok; this is through simulation. But as an alternative what you may do? You may use some kind of a synthesis tool to map it to a piece of hardware.

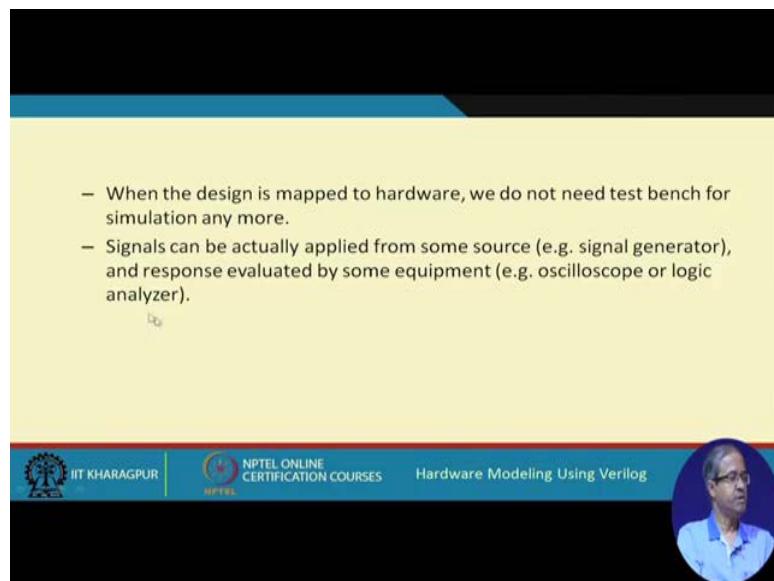
So, what is synthesis tool? The synthesis tool will be specifying your; it will be taking your specification which you have written in Verilog and it will be translating into ultimate hardware design; that will be your final hardware, ok, that what is called synthesis. So, the synthesis tool will convert your specification into a netlist of low level primitives. Now, when you say low level primitives again there are two alternatives available with you.

Some of you may say that well I want to design a VLSI chip; which means I want to design something called as an Application Specific Integrated Circuit or ASIC. So, here I will have to go through all the steps of the VLSI design cycles as I just mentioned earlier. After going that; I will have to send the layout specification that I have finally, there are some particular formats for the GDS2; I will have to send it to a fabrication facility and they will fabricate the chip and they will send the chip back to me.

This is one way you can utilize your design; what you want to do with it or the other one is that most of us do the second one; that your target hardware can be Field Programmable Gate Array or FPGA. So, what is the Field Programmable Gate Array? and FPGA is something which is programmable.

So, you can use it in your laboratory. So, you can create a design in Verilog; you can use a synthesis tool, it will be translating into some form which you can download on the FPGA chip by giving a command. Suppose you have designed a multiplier; you download that in the FPGA chip; your FPGA chip becomes a multiplier; that is your hardware. So, FPGA is nothing, but a programmable hardware you can make it behave in whatever way you want by downloading an appropriate programming data on it. So, these are the two ways you can utilize this synthesis tool.

(Refer Slide Time: 10:27)



So, when the design is mapped to the hardware; we do not need the test benches for simulation anymore. Because you already have the hardware; so, we can actually apply the voltage signals and see the output, ok. We do not need any test bench and simulation to do that; we have the actual hardware with us. So, here the signals can be actually applied from let us say some signal generator and the outputs maybe actually observed on let us say an oscilloscope or some logic analyzer, ok.

(Refer Slide Time: 11:02)

The slide is titled "Hardware Modeling Using Verilog" and is part of the "NPTEL ONLINE CERTIFICATION COURSES" offered by IIT KHARAGPUR. The content is organized into two main bullet points:

- Using ASIC as hardware target?
 - When high performance and high packing density is required.
 - When the manufactured hardware is expected to be used in large numbers (e.g. processor chips).
- Using FPGA as hardware target?
 - When fast turnaround time is required to validate the design.
 - The mapping can be done in the laboratory itself with a FPGA kit and associated software.
 - There is a tradeoff in performance.

So, when do we use this ASIC as a hardware target? Now, we use ASIC as a hardware target; when we need high performance and high packing density. You see this application specific integrated circuit is very optimized. So, whenever you use this ASIC, you are generating a hardware; which will be very much optimized in the sense that its area will be less, its speed will be higher and it will possibly be meeting all your delay and power related constraints, ok.

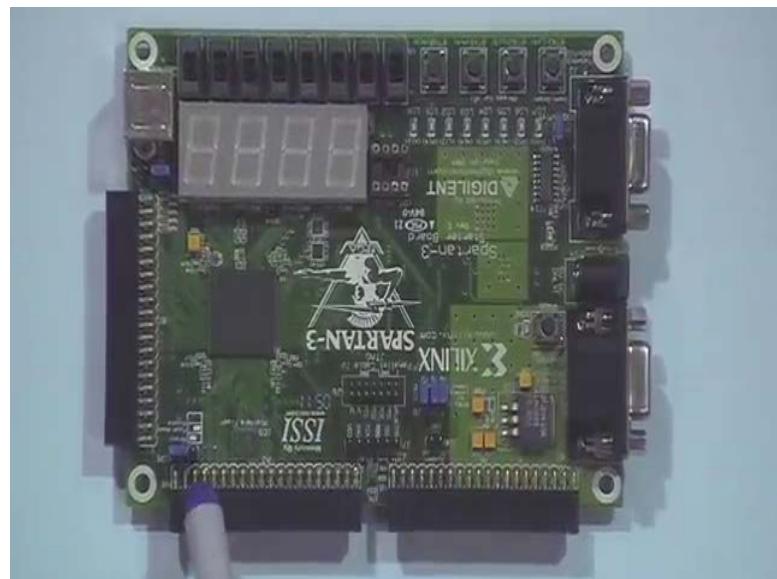
But there is a price to be paid; when you design an ASIC that two things, it is a very long drawn processes. It will take you months to design the chip send it for fabrication getting the chip back and secondly, it is very expensive. So, the fabrication facility will be charging you a lot; many many lakhs of rupees; let us say to get a chip fabricated, so it is very expensive.

So, there has to be some kind of cost effectiveness in that; so you can possibly afford to use that only when manufactured hardware is expected to be used in large numbers. So, one classical example is a processor chips; so, you think of the Intel processors; they manufacture the chips in millions all around the world. So, they come up with the design which is best possible in terms of its speed and performance. So, they will have to go for in ASIC designs, right, but when I created a design for my class, my laboratory; I cannot afford to spend so, much of money. So, I will be doing it in my lab and I will be using a Field Programming Gate Array or FPGA.

So, we use FPGA as the target when fast turnaround time is required; fast turnaround time is required means once you written a code in Verilog; it can be a few minutes; you do a synthesis, you download it on FPGA; your hardware is ready, right. So, this you can do in your lab itself, so what you need? You need something called FPGA board or FPGA kit and some associated software.

Trade off in performance means well because you are using a programmable hardware, your speed may not be as good as ASIC; means your area will also be bigger. Well it means, it will not be that optimized; well here I am showing you one example of an FPGA kit.

(Refer Slide Time: 14:02)



This is an FPGA kit you see; this is a board where there is small chip you can see here; this is your FPGA, well I can just keep it here you can see it from the top. This is your FPGA; this square thing you can see, this is your FPGA, right and you see there are other components in this; there are some switches, you can apply some inputs through this switches; there are some seven segment displays, there are some small LED's and there are some push button switches, right.

So, when you download some design on this FPGA. So, what will happen is that means, mean you can actually apply some inputs through this switches; through this push buttons and the output you can observe on the LED's or the seven segment displays and see

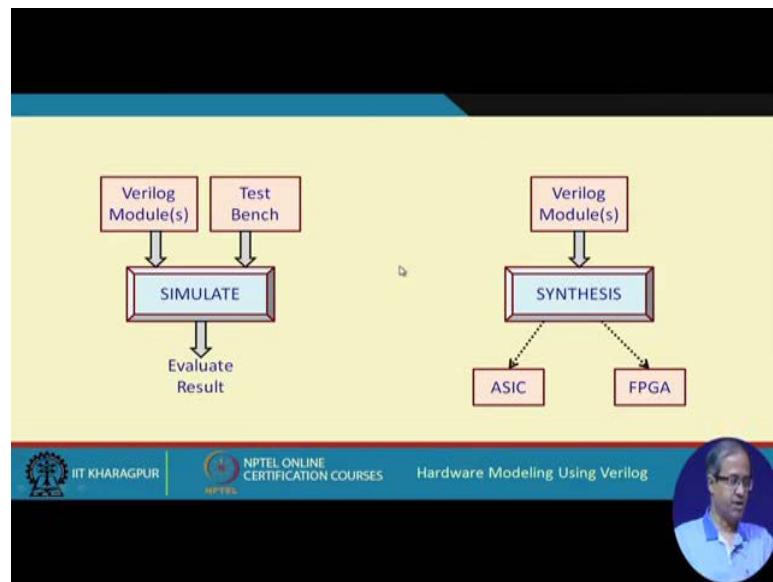
whether functionally it is correct or not. But if it is a more complex design, you possibly need to connect some external device like a logic analyzer or an oscilloscope.

So, you see that on this board; there are some other components are also there. Like here there are some serial ports, there are two serial ports out here and on this side there are some h connectors; you can connect some h connector through which you can connect to a processor. And here there are some connectors here; you can see, these connectors will be connected to a PC or a desktop or a laptop; from there the FPGA data will be downloaded; through this pins, right.

So, this is a very cheap cost effective device; this device for example, costs only a few thousands of rupees, right. So, in your lab you can have a device like this and if you have a device like this; you can very quickly come up with the design, put it on a FPGA and test it and see whether it is running or working correctly or not. Now, most of you or many of you possibly have already used FPGA or will be using FPGA at some point in time.

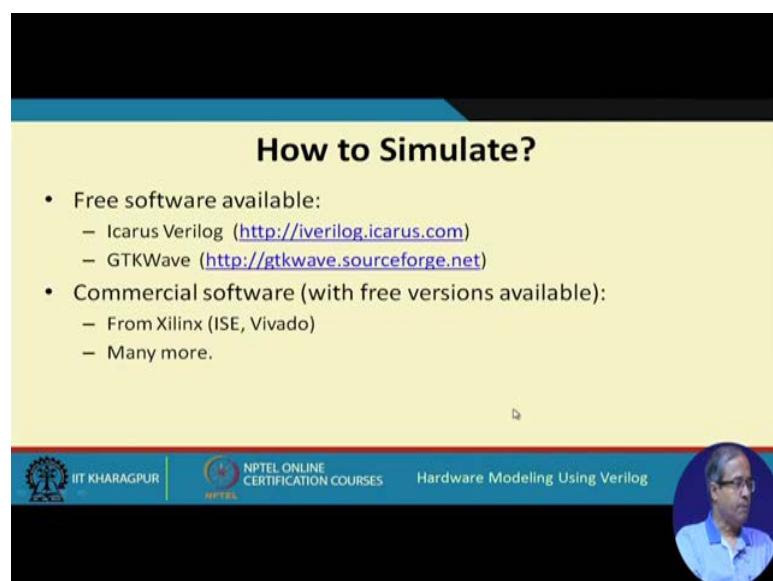
But, as I told you as part of this course; I shall not be teaching you how to use FPGA's or how to convert my Verilog description through synthesis and burn into FPGAs. Rather in this course; I shall be concentrating on explaining how this Verilog language is and what are the different ways to create a design or specify a design; based on certain constraints and requirements, ok. And we shall be mostly telling you and urging you to use simulation to verify that; whatever you have learnt, you have written a code whether that code is working correctly or not. Because simulation is much easier and simpler to run and check, I shall show you an example also, fine.

(Refer Slide Time: 17:24)



So, pictorially these are the two alternatives; so, if you decide to go for simulation, then you have the Verilog modules and you have a test bench module, you combine them, simulate and we evaluate the result; whether the result is coming as per your expectation or not. And if your target is synthesis; then you only need Verilog modules, you do not need test bench. and depending on your requirement you can either map it to an ASIC or you can map it to an FPGA; as I said, right.

(Refer Slide Time: 18:08)



Now, the question is how to simulate? Now, here I am showing you two links; there is, there will be tutorials which will be uploaded also on these site, you can basically look at them; that how to actually download and install them. See here, I strongly recommend you to download this Icarus Verilog or Iverilog in short; which is available on this link Iverilog dot Icarus dot com. This is a free software and this Verilog simulator; it runs on almost all platforms; windows, various versions of Linux, Mac everywhere, ok.

So, it is more or less platform independent you can run it anywhere you want and means after simulation if you want to see the results in a graphical way; is in wave form; there is another tool you can also download this is also free; GTK wave, you can download it from this site. So, again GTK wave is available on all platforms, right, but if you want to use some commercial software's; many of the commercial software's have a free version with limited capabilities available.

Like the FPGA manufactures Xilinx is one of the leading FPGA manufactures. They have the software tools called ISE, Vivado and a few others and there are many others vendors also. So, using any of these tools, you can carry out or do simulation, but in the examples that I shall be showing and illustrating. So, I shall be illustrating with Iverilog only; so, if you want to reproduce my examples. So, it is always good, it is always recommended that you install Iverilog on your machine, on your computer and you can also run the Verilog codes there and so means, you can verify whatever I am saying whether it is matching with your simulation or not, ok, alright.

(Refer Slide Time: 20:20)

How to Synthesize?

- For FPGA as target, specific software is required:
 - Xilinx ISE or VIVADO for Xilinx FPGA kits.
 - Similar software available from other FPGA vendors.
- For ASIC as target, commercial CAD tools exist:
 - Tool suite from Cadence.
 - Tool suite from Synopsys.
 - Several others ...

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



So, if you want to do synthesize; synthesize again if you use FPGA as target. So, you need specific software; like the FPGA board that I had shown that was a board which is manufactured by Xilinx; the FPGA chip was a Xilinx Spartan 3; FPGA chip, ok. So, for Xilinx; you need to use either a Xilinx ISE software or a Xilinx Vivado software for those kits. But for other FPGA vendors Altera, Actel; there are many several others; there are similar software available from the other vendors also.

So, for FPGA there is one constraint; if you use a particular type of a FPGA kit, you have to use this software that is provided by that particular means FPGA chip manufacturer; for Xilinx, you have to use only the Xilinx software, ok. But when you want to go for ASIC, you can use some commercial CAD tools; which are more generic in that sense.

That means, you are not stuck with the particular tool; there are tools from Cadence, there are tools for Synopsis and several other companies are also there. These are commercial tools; much more professional, much more elaborate and of course, much more expensive as well.

So, you cannot afford to buy for personal use. So, only for large institutions and laboratories; you can possibly afford to buy a license, ok, fine.

(Refer Slide Time: 22:07)

Scope of this Course

- We shall be discussing features of the Verilog language, and verifying the design through simulation.
 - How to design combinational and sequential digital circuits?
 - How to verify the functionality through simulation?
- We shall not be discussing the synthesis tools; however:
 - Shall discuss various tricks that help in synthesis.
 - Shall discuss the common design flow – first code the modules in behavioral design style, and then translate selected subset of modules to structural design style.

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

So, as I had said that; in this course, we shall be discussing various features of Verilog, ok. How to write Verilog modules? So, how to design combinational and sequential modules, circuits? and we shall also tell you that how to verify the functionality through simulation, but we shall not be discussing synthesis as part of this course; because it is slightly beyond this scope. But, we shall be discussing a few things which can help you in the synthesis; for those of you, who are interested in actually carrying out synthesis; may be now or maybe later.

So, we shall be discussing some tricks, some techniques in Verilog coding; which will help you in generating; you can say good design, the final hardware that you are mapping to, that will be a more efficient hardware, more efficient design. There are various tricks so we shall be seeing some of them. and also we shall be looking at the typical design flow which is followed; because you see through the examples you may have got an impression that writing a behavioral code is easier, writing a structural code is more complex. So, what is typically done is there for a complex design. First we code all the modules in behavioral fashion. Then one by one we replace those behavioral specifications by structural specification. So, not all of them so, a good subset of the modules are finally, translated into structures specifications, but the remaining one say for example, the control you need the control module that is often left in the behavioral form only, they will not be converted into structural, ok.

So, there is a trade off that has to be decided by the designer. So, if I have 100 modules, how many of them I want to convert it into the structural fashion, because you see once I convert it to a structural fashion. So, as a designer, I will have much better control about this structure of my design and also the performance. So, I can guarantee that my design will be good and it will perform according to specifications. Now if I give everything to the synthesis tool, ultimately it is a translator program, very complex program. So, often it generates a circuit which is correct, but in terms of efficiency it is seldom, it sometime happens in the, it is not so good. So, it is, it always helps if the designer gives some inputs to the tools so that the tool will understand that what to do and how to do right, fine.

(Refer Slide Time: 25:21)

How to Simulate Verilog Module(s)

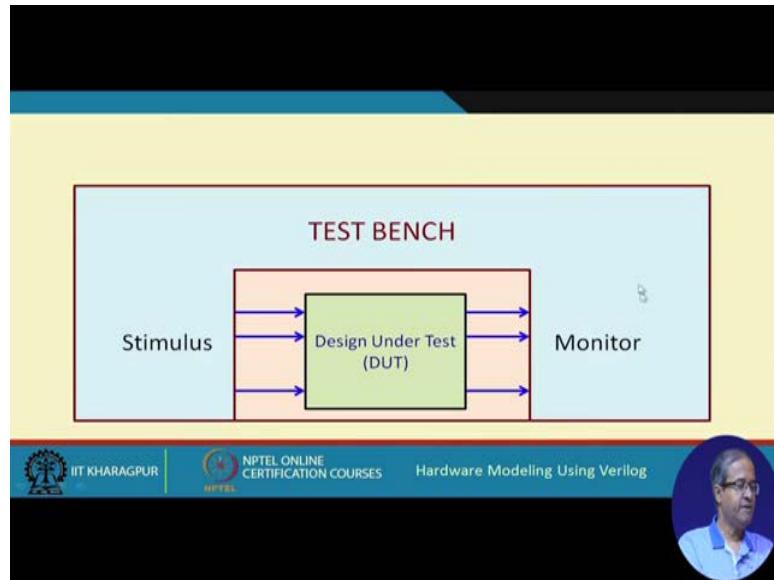
- Using a test bench to verify the functionality of a design coded in Verilog (called Design-under-Test or DUT), comprising of:
 - A set of stimulus for the DUT.
 - A monitor, which captures or analyzes the outputs of the DUT.
- Requirement:
 - The inputs of the DUT need to be connected to the test bench.
 - The outputs of the DUT needs also to be connected to the test bench.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog | 10

Now, the question is how to simulate Verilog module? Let us just explain this with the help of there was small example. So, as I said for simulation we need a test bench. So, let us say we are using a test bench to verify the functionality of a design, a module that module you are calling are referred to as Design-Under-Test or DUT. So, we have a design under test, we want to simulate it and verify that this is working correctly or not. So, in order to do that what the test bench will be doing. So, it will first be applying a set of inputs which are called stimulus. So, a set of stimulus will be applied to the DUT and the output of the DUT will be monitored. So, the test bench will also be capturing or analyzing the DUT outputs, right.

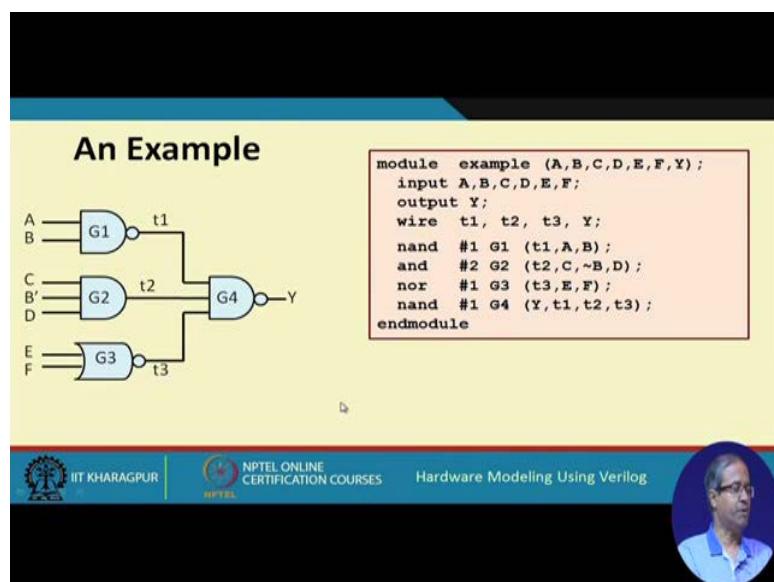
So, this is the requirement, the test bench has to feed the input of the DUT. and also the outputs of the DUT need to be connected to the test bench for doing this. So, pictorially I can visualize it like this, as if this whole thing is the test bench.

(Refer Slide Time: 26:39)



Now within the test bench, I have embedded my design under test. So, test bench has one part called stimulus; which is applying some inputs. There is another part called monitor, which is see observing and monitoring the outputs, right.

(Refer Slide Time: 27:05)



Now, let us take an example. A small example, there is a small combination circuits consisting a four gates: 2 NAND gate, 1 AND gate and 1 NOR gate. This is a simple structural description of this circuit using the primitive gates which are already available in the Verilog library: NAND, AND, NOR and NAND.

So, all the signals A, B, C, D, E, F and Y these are specified. Out of the A, B, C, D, E, F are the inputs; Y is the output, and this intermediate lines, these I am giving some names t1, t2, t3 and this Y you can declare as Y may not be mandatory. Now just ignore this numbers for the timing this NAND you see first NAND, G1 I am giving a name G1, t1, A, B; t1 is the output A, B; G2 and t2, C this is NOT, C NOT B, D; C, B NOT D this is G2; NOR t3, E, F; t3 NOR E, F; G4, Y is the output, G4, t1, t2, t3; and here for the sake of simulation purpose, I can specify the delay of the gates. Like I say that the delay of the NAND gate is 1 time unit, NOR is 1, NAND is 1, but AND is 2. Let us say I have defined like this, ok.

So, this is the way to specify, this kind of a structural netlist. Now one thing; there are 2 NAND gates right. So, instead of writing this NAND twice, so here we can also combine the declaration, I can write NAND once, both are delay 1, and I can write G1, t1, A, B comma G4 this and only a semicolon after that. So, I can combine several gates of the same time, type by writing it only once and combining them, ok, fine. So, this is the module we have written.

(Refer Slide Time: 29:18)

```

module testbench;
reg A,B,C,D,E,F; wire Y;
example DUT(A,B,C,D,E,F,Y);
initial
begin
$monitor ($time," A=%b, B=%b, C=%b,
D=%b, E=%b, F=%b, Y=%b",
A,B,C,D,E,F,Y);
#5 A=1; B=0; C=0; D=1; E=0; F=0;
#5 A=0; B=0; C=1; D=1; E=0; F=0;
#5 A=1; C=0;
#5 F=1;
#5 $finish;
end
endmodule

```

```

module example
(A,B,C,D,E,F,Y);
wire t1, t2, t3, Y;
nand #1 G1 (t1,A,B);
and #2 G2 (t2,C,~B,D);
nor #1 G3 (t3,E,F);
nand #1 G4 (Y,t1,t2,t3);
endmodule

```

example-test.v



Now, we are trying to write a test bench. So, what we are doing in the test bench? This is how the test bench will look like. This I am writing as the module testbench. This A, B, C, D, E, this is my example, right, module example. So, I am instantiating this here A, B, C, D, E, F are the inputs and Y is the output, right.

So, I have instantiated the example I am calling it DUT. So, A, B, C, D, E, F, Y all these are there. Now just ignore this line for the timing I will come back. This is my test bench. Now, in test bench we often use the key word initial. Initial means there is a block of statements, this block will be executed only once, begin end, right. So, what does this block contain? Monitor, dollar monitor, monitor is something like printf in C. So, here I am saying that you monitor the variables and you print certain things. what are the things to print? Dollar time means, this simulation time comma within quote A equal to percentage B means a bit or binary, ok, B equal to C equal to just the syntax is very similar to C. C, D, E, F comma then the actual name of the variables.

So, we are printing these variables. Now what monitor means that whenever anyone of these variables changes, then you print. Monitor is an intelligent print statements. So, it will not print every time, it will print only when at least one of the variables change, ok, fine. So, then what I say this hash 5 means these are time. It says after time 5, apply these inputs. Then after again another time 5 apply these inputs. Then again time 5 A equal 1, C equal to 0, it means the other inputs are not changing. Only you change A and you change C, but B is still 0, D is still 1, E is 0, F is 0. And again after time 5: you set F equal to 1 and again after time 5, dollar finish means you finish this simulation, ok. This is a very simple test bench and this is the meaning. Let us say the name of the file is example test dot v, and the original description of the module that you wrote the name of example dot v.

So, this example was instantiated here, right. This is how within a test bench, I can instantiate the module and apply the inputs. Now one thing, that inside the test bench initial block. So, whatever variables appear on the left hand side here A, B, C, D, E, F they have to be declared of type reg, reg is the register type variable that is why we have declared these are type reg and Y is the other variable, which is not on the left hand side as simply declared as wire, wire Y, right. Now this simulation results, if we simulate this design, then you can actually see that when the gates are changing the delays. So, I leave it as an exercise for you that you can see.

(Refer Slide Time: 33:01)

The screenshot shows a presentation slide with the following content:

- Simulation results:

```
0 A=x, B=x, C=x, D=x, E=x, F=x, Y=x
5 A=1, B=0, C=0, D=1, E=0, F=0, Y=x
8 A=1, B=0, C=0, D=1, E=0, F=0, Y=1
10 A=0, B=0, C=1, D=1, E=0, F=0, Y=1
13 A=0, B=0, C=0, D=1, E=0, F=0, Y=0
15 A=1, B=0, C=0, D=1, E=0, F=0, Y=0
18 A=1, B=0, C=0, D=1, E=0, F=0, Y=1
20 A=1, B=0, C=0, D=1, E=0, F=1, Y=1
```
- Command in iVerilog:
 - a) iverilog -o mysim example.v example-test.v
 - b) vvp mysim

The slide footer includes the IIT Kharagpur logo, NPTEL Online Certification Courses logo, and the title "Hardware Modeling Using Verilog". There is also a circular profile picture of a man.

That your simulation result will come like this, at time 0 all the variables are undefined, X means undefined. At time 5, you see at time 5, I am applying this: A1, B0, C0, D1, E0, F0. So, I have applied this, but still Y is undefined, see after time 3 your Y has changed; why because you see this circuit this is a delay of 2 plus 1, 3, this AND gate has a delay of 2 plus 1, after delay of 3 this Y will change, right. So, after time 8, Y will change, similarly this is the time. So, here you can just check, similarly that the times will be printed and the values will change.

So, this will be your simulation output and now the question is given these two files example dot v and example test dot v. So, how do I get this output well its very simple if you just a second if you just install Iverilog and if you just give a command like this Iverilog minus o just like in C compilation you can give an output file name minus o this is the output file name mysim then all the Verilog files example dot v example test dot v. Then you give another command vvp mysim. So, if you run this, if you run this vvp this output will be obtained, right, very simple.

So, this you can verify. Now, here I have made a little modification in the test bench. I have just added these 2 lines: dumpfile and dumpvars, rest is same, rest all are identical. Dumpfile means I am saying that well in the monitor is given it is printing all right, but I will also dump all these changes in a file called example dot vcd, vcd stands for value change dump and dumpvars 0 dot 0 comma test bench means test bench is the name of

this module. So, all the variables in this modules and all the variables that are included inside they will all be dumped, right now, just after this if you run it again that vvp. So, in this file example dot vcd will be created.

So, you just give a command like gtk wave example dot vcd, if you install gtk wave also just give this. So, what you will get, you will get something like this. This is the window of gtk wave. So, in this left corner you will see the name of your test bench file DUT you can see all the name of the wire. So, what I have done? Is that I have just pulled this, these values to this window one by one and this wave forms are immediately starting to appear. So, you can see this is the time on top 7 second, 7, 14, 21, 0. So, you can see that exactly the time at time 5 this is changing. A has become 1. So, at time 7, this Y has become 1; this exactly how the simulation output is coming. You can see it in the form of time for timing diagram. So, you can see this visually in the form of a wave form.

So, I think with this we just come to the end of this lecture. So, in this lecture we just saw, how we can actually write a Verilog module, how we can write a Verilog testbench, how we can simulate them, how we can see the output, and how also we can visualize the timing diagram. Now these experiments I shall not be showing on the time in my next lectures. I expect that you will be installing Verilog and gtk wave in your machines, and you will be actually running this codes yourself and seeing them working, ok.

Thank you.

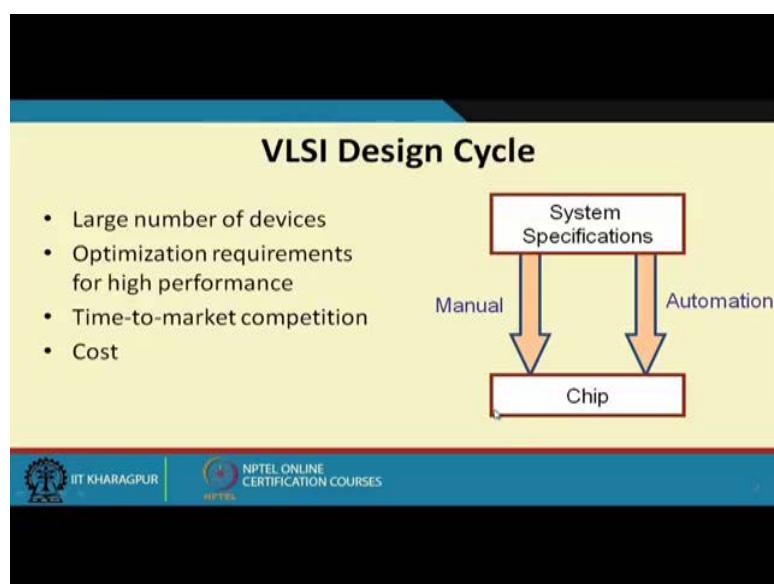
Hardware Modeling using Verilog
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 04
VLSI Design Styles (Part 1)

So, here we now take a break from Verilog and look at some of the so-called VLSI design styles. Well, when you talk about VLSI design styles, it means some of this I already talked about in the last lectures like I talked about ASIC application specific IC, I talked about FPGA, there is something called Semi-custom and Full-custom designs, there is something called Gate Array which falls between FPGA and ASICs.

So, will see some of the features of these design styles, the reason I would be discussing this is that, some of you may be actually designing this kind of circuits. So, you may want to see a correlation between the Verilog coding and the design that you will be doing and your final target hardware. So, there are a few differences. So, unless you understand these differences very clearly, it will be difficult for you to customize or modify your design as you move from one design style to another. So, we start with our discussion on VLSI design styles.

(Refer Slide Time: 01:47)



Now, VLSI design cycle, already I mentioned a little bit earlier that the VLSI design complexity has increased immensely over the years, manual design or synthesis is simply

out of the question, now you have to use computer aided design tools which means automation.

(Refer Slide Time: 02:20)

VLSI Design Cycle (contd.)

1. System specification
2. Functional design
3. Logic design
4. Circuit design
5. Physical design
6. Design verification
7. Fabrication
8. Packaging, testing, and debugging

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

So, there are many reasons for doing this time to market computation, cost optimization and so on. Talking about the VLSI design cycle, here I am showing you a slightly more detailed stepwise breakup. So, starting from the system specification; you can go to functional design which is the register transfer level design, logic design, gates flip flops circuit design in terms of transistors, physical design. Physical design is something which will again look at a little more, design verification, that whether our design is correct, finally fabrication and after fabrication packaging the chip, testing and debugging these are the overall steps in the VLSI design cycle.

(Refer Slide Time: 03:06)

Physical Design

- Converts a circuit description into a geometric description.
 - This description is used for fabrication of the chip.
- Basic steps in the physical design cycle:
 1. Partitioning, floorplanning and placement
 2. Routing
 3. Static timing analysis
 4. Signal integrity and crosstalk analysis
 5. Physical verification and signoff

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

But talking about physical design which is somewhere here; so, after your circuit netlist is done, we go for physical design before you can actually go for fabrication, right. So, physical design roughly says that I start with some kind of a netlist typically at the gate level or the transistor level and from the netlist; I ultimately translate it in to my manufacture hardware.

So, my manufacture hardware can be ASIC, it can be FPGA, it can be anything, but this step is like this, I start with a netlist at a sufficiently lower level gate level or transistor level and I map it in to my hardware, ok. Now in this steps of physical design, we again may have to go through a number of intermediate steps like there are steps called partitioning, floor planning and placement like you have to decide if it is a large design how to break it up in to smaller partitions and on the surface of silicon where to place them, how to plan my total chip floor plan and so on, then comes a very important step of routing.

So, how to interconnect this blocks that I have already placed, static timing analysis extremely important in the modern day context. So, after I have completed placement and routing, I need to check that whether my design is meeting my timing constraints or not, if not, I may have to go back and change these things. Signal integrity, cross stock analysis these are related to timing analysis, this are also very important and when everything is fine, I see that everything is meeting the requirements then I complete my physical

verification process and do a step called signoff. Signoff means I am satisfied with my physical design, now I can proceed to fabrication, ok, fine.

(Refer Slide Time: 05:22)

Various Design Styles

- Programmable Logic Devices
 - Field Programmable Gate Array (FPGA)
 - Gate Array
- Standard Cell (Semi-Custom Design)
- Full-Custom Design

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

Now, talking of the design styles, well, I shall be looking briefly in to this 4 design styles, there is the first category of programmable devices, I will talk about field programmable gate array FPGA and also simple Gate array, then for Standard cell design or Semi-custom design and Full-custom design. The last 2, they fall under the category of ASICs application specifies, fine.

(Refer Slide Time: 05:54)

Which Design Style to Use?

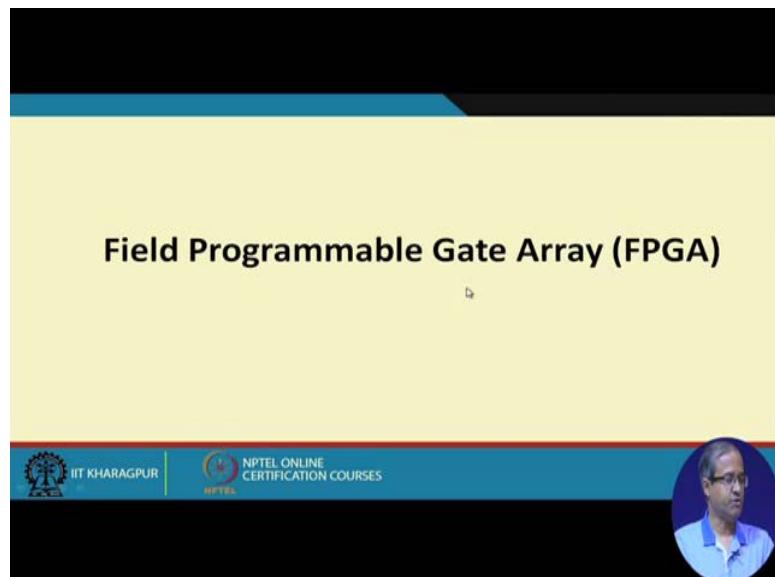
- Basically a tradeoff among several design parameters.
 - Hardware cost
 - Circuit delay
 - Time required
- Optimizing on these parameters is often conflicting.

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

Now, which design style to use; this is a matter of deciding on part of the designer, there is a tradeoff, there is a tradeoff between hardware cost performance and the total time required for the design. You see FPGA is easiest to design, but the circuit delay may be bad, but ASIC performance is very good, but hardware cost and time required will be very high, ok.

So, these parameters are often very conflicting. So, when you are going for a particular kind of design we have to look in to the bigger context that exactly for whom we are designing it, who are the potential customers and what are the main objectives that need to be satisfied. So, in that way we can decide on the optimizing criteria in a much better and concise way, fine.

(Refer Slide Time: 07:00)



So, start with FPGA field programmable gate array.

(Refer Slide Time: 07:04)

What does FPGA offer?

- User / Field Programmability.
 - Array of logic cells connected via routing channels.
 - Different types of cells:
 - Special I/O cells.
 - Logic cells (Mainly lookup tables (LUT) with associated registers).
 - Interconnection between cells:
 - Using SRAM based switches.
 - Using anti-fuse elements.

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

Now, FPGA as I mentioned earlier this offers user programmability or field programmability means we can do the programming in our lab sitting on a table, sitting in front of table, you can do it. Now what an FPGA is really? FPGA as I said just said it is a programmable device, but inside there is an array of logic cells, array means many thousands of logic cells, they are placed in a regular array and they are interconnected via routing channels. Now both this logic cells and the routing channels are programmable means their functionality can be modified.

Now, in addition to it, there are some other cells called I/O cells. Now this logic cells can be either I/O cells or they can be something called lookup table blocks and this routing channels interconnections, they are manufactured in different way; either using static RAM or using something called anti-fuse. Static RAM means inside, there are small memories from outside I can store some bit pattern in this memory 0s and 1s. So, if I store a 0 some switch will be open, if I store a one some switch will be closed. So, that way I can program my interconnection. Now, anti-fuse is something which is a little different it is one time. So, by passing a high current between 2 points; so, either I can blow out or I can connect a connection, now anti-fuse means normally there is no connection.

So, if a high current is flowing then the material will be fusing, it will be melting and a connection will be established.

(Refer Slide Time: 09:17)

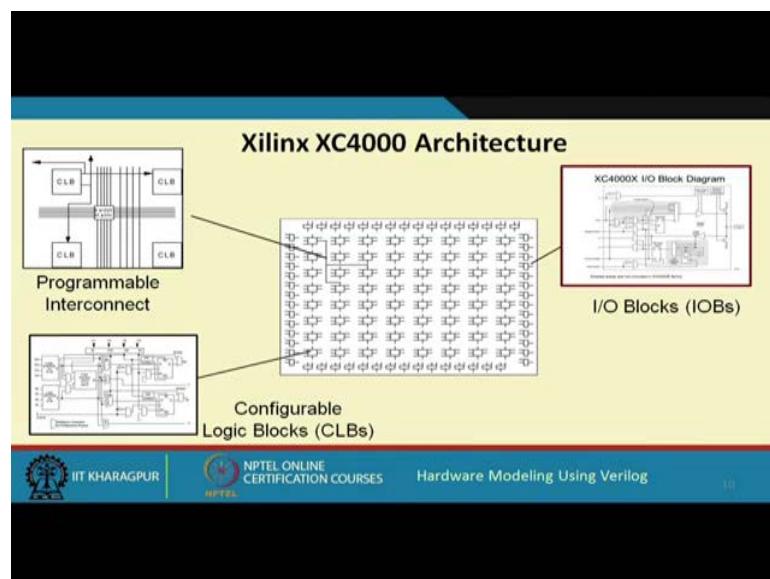
Ease of Use

- FPGA chips are manufactured by a number of vendors:
 - Xilinx, Altera, Actel, etc.
 - Products vary widely in capability.
- FPGA development boards and CAD software available from many sellers.
 - Allows rapid prototyping in laboratory.

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

So, there is some FPGA manufactures which also use this kind of empty fuse technology, fine. So, means; obviously, FPGA's are very easy to use this, this chips are manufactured by a number of vendors like Xilinx, Altera, Actel; their products vary widely in capability, there are FPGA chips now available which are very fast, very complex radiation hardened. So, you can map very complex and large designs in to those FPGA chips as well, ok and development boards and CAD software are available.

(Refer Slide Time: 09:55)

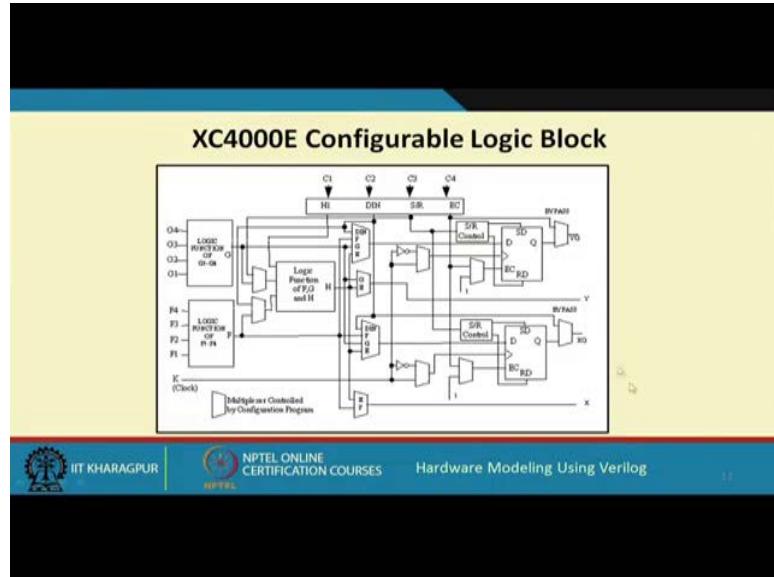


Now, here we have a very quick look at a particular FPGA architecture Xilinx XC4000. You see this is a bird's eye view of the whole chip, you see at the periphery there are some small rectangles these are the so-called I/O blocks or I/O cells.

This I/O blocks contains some multiplexers, flip flops, tri-state buffers; you see from external pins, there are signals which can be coming in or going out, some of these blocks can be input pins, some of these blocks can be output pin. So, by programming these blocks you can configure them, ok. Then you have these rectangular blocks in between, these are something called configurable logic blocks. So, again by programming this you can make them work in various ways. So, shall we should see a little later. So, how we can do this and in between the configurable logic blocks there is some spaces there are some programmable interconnects are there, by again programming the switches here I can connect, make connection from one block to the other as per my requirement.

So, whatever design I have, I can map them on to this fabric and I can program them in such a way that appropriate functionality and appropriate interconnections are made.

(Refer Slide Time: 11:41)



So, that whatever circuit I want to design I want to implement that gets implemented, implemented on the FPGA chip, this is a basic function. So, the configurable logic block it looks like this you see I am not going to detail, but the more interesting thing is that there are 2 blocks here, that 2 blocks here which are called lookup tables. So, as you can see there are 4 inputs and one output, this lookup tables can realize any 4 variable function.

So, I will just explain a little later how and there is also 2 flip flops in the output. So, by properly selecting this multiplexer, there are many multiplexers you can see. So, you can actually connect them in a variety of ways, right, some someway you can program them.

(Refer Slide Time: 12:35)

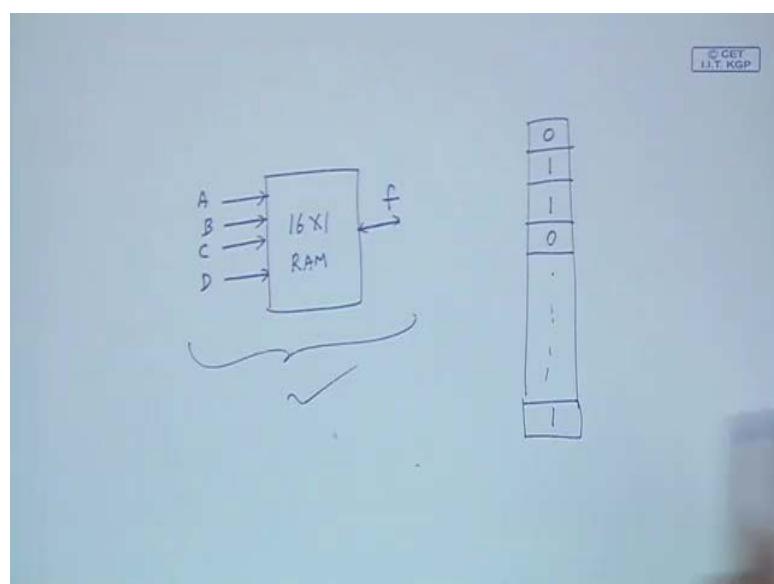
CLB Functionalities

- Two 4-input function generators
 - Implemented using Lookup Tables using 16x1 RAM.
 - Can also implement 16x1 memory.
- Two 1-bit registers
 - Each can be configured as flip-flop or latch.
 - Independent clock polarity.
 - Synchronous and asynchronous Set / Reset.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog | 12

So, CLB as I said, there are 2 4-input function generators here these 2, they are implemented using lookup table using 16 by 1 RAM.

(Refer Slide Time: 12:55)



So, how it is done; let us try to explain, suppose I have a 16 into 1 memory RAM, there are 16 words each containing 1 bit. So, there will be 4 address lines and then there will be

1 data output or input whatever you say. So, in this memory location there will be 16 locations, you consider a 4 variable function A, B, C, D. So, any 4 variable functions, if you look at the truth table, there will be 16 columns. Suppose I want to realize some function f of 4 variables, I construct the truth table and I just fill up these memory locations by the output column of the truth table. So, once I do this my function is ready. So, I apply an input that corresponding location will be selected and that corresponding output will be generated.

So, just by modifying this memory, I can implement any arbitrary function of 4 variables, this is how the programmability comes into the CLBs, right.

So, in addition to implementing 4 functions; 4 input functions, it can also be used as a memory if you require 16 by 1 memory and also inside you have seen that there are 2 flip flops, they are 2 1-bit register, they can be configured as a flip flop or a latch, clock polarity you can have leading edge triggered, falling edge triggered, set, reset all these facilities are there. Lookup table as I mentioned one lookup table is shown here.

(Refer Slide Time: 14:45)

Look Up Tables (LUT)

- Combinatorial Logic is stored in 16x1 SRAM Look Up Tables (LUTs) in a CLB.
- Capacity is limited by number of inputs, not complexity.
- Choose to use each function generator as 4-input logic (LUT) or as high-speed RAM.

The diagram illustrates a CLB (Configurable Logic Block) structure. It consists of a central 'CLB' block containing a 'LUT' (Look Up Table) and an 'FF' (Flip Flop). The 'LUT' receives four inputs (A, B, C, D) and has one output that feeds into the 'FF'. The 'FF' also receives control signals 'Rst-' and 'Clk-' and has an output labeled 'Q'. This output 'Q' is further processed by a logic gate (indicated by a triangle symbol) to produce the final 'Out' signal.

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

I am showing 1 lookup table, flip flop, multiplexer. So, this lookup table can implement any function of 4 variables which gives its power, as I said, ok.

(Refer Slide Time: 15:05)

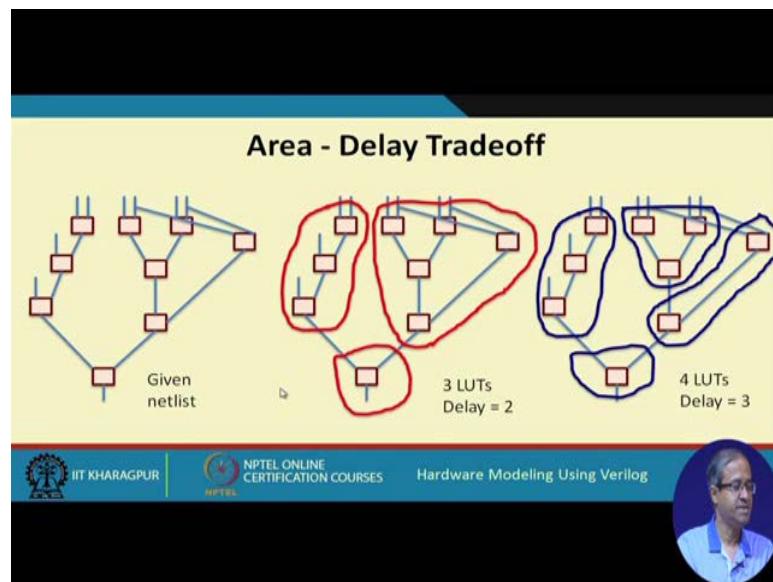
LUT Mapping: An Example

- A function: $f = A'B + B'C'D$
- The mapping process:
 - Create the truth table of the 4-variable function.
 - Load the output column into the SRAM corresponding to the LUT.
 - Apply the function inputs to the LUT inputs.
- Any 4-variable function can be realized.
 - Netlist to LUT mapping is an interesting design tradeoff.



So, just one small example; so, if you are given a means function, we create the truthtable of the function just as I said, load the output column of the truthtable in to the SRAM. So, my function is ready. So, I apply A, B, C D to the input, I get f as the output of the memory.

(Refer Slide Time: 15:35)

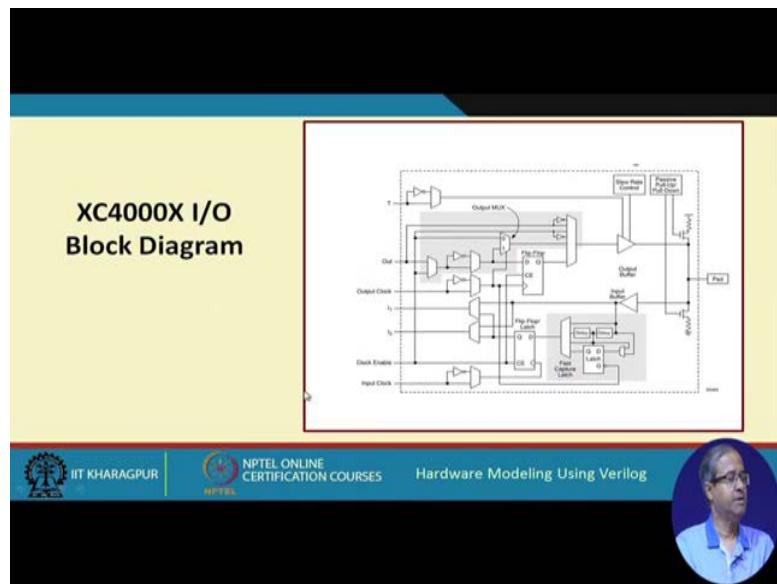


So, any 4 variable function can be realized and mapping there can be area delayed like means, I am just showing an example suppose these are some gates, some gates and these are the inputs, lines are inputs, this is a given netlist. So, I want to map them in to those lookup tables. So, one way of mapping may be like this, this can be 1, this can be 1, this

can be y; you see if, I take this part of it, this whole thing there are total 4 inputs, you see 1, 2, 3 and 4. So, overall this is a 4 input and 1 output circuit; this part similarly, this whole thing is 1, 2, 3, 4 and 1 output.

So, any sub circuit with any number of gates with maximum up to 4 inputs and one output can be mapped to the LUT and here whatever remains I can map into 1 one more LUT. So, so here there is only 2 inputs, ok. So, there are 3 LUTs which are required here, delay is 2, this is 1 level of delay, 2 level of delay. Now you think of an alternate mapping; let us say I take this as one, let us; I take this as 1, there is also 4 inputs; 1 output and I take this as 1; 1, 2, 3 input and 1 output and this as 1. So, here there will be 4 LUTs, delay will be 3 because this will be one delay this output is coming here, this will be another delay and then this 3. So, this is a bad mapping this is a better mapping, right. So, this LUT mapping is a big problem, this a challenge.

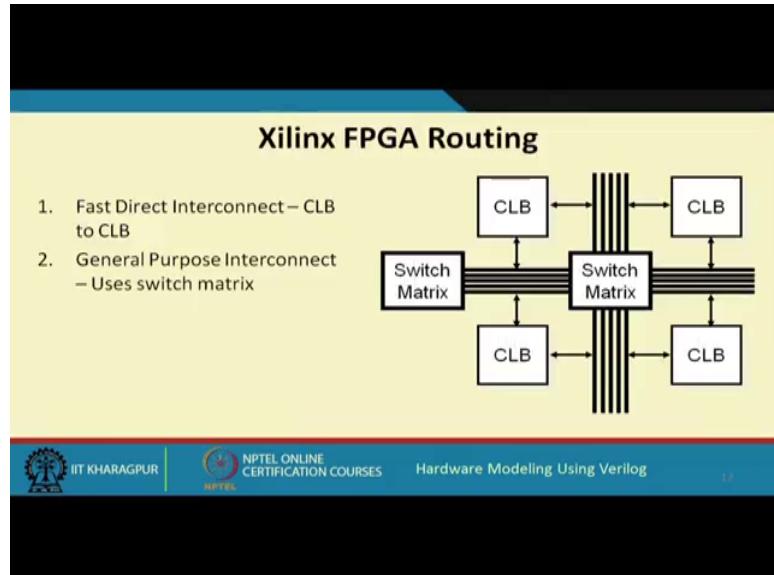
(Refer Slide Time: 17:25)



So, the I/O cells are also quite complex as you can see the I/O blocks, there are also latches there. There are flip flops, these I/O pads, the tri-state controls, buffers pull ups, there are a lot of things you can again; there is some RAM, SRAM inside by loading them you can configure the output pins that whether it is an input pin or an output pin whether you require tri-state control and so on, whether it is a stroked output, you need clock leading edge, falling edge all these things you can program. So, these are extremely flexible cells.

So, by specifying a few bits you can specify exactly how these, this individual I/O blocks will work.

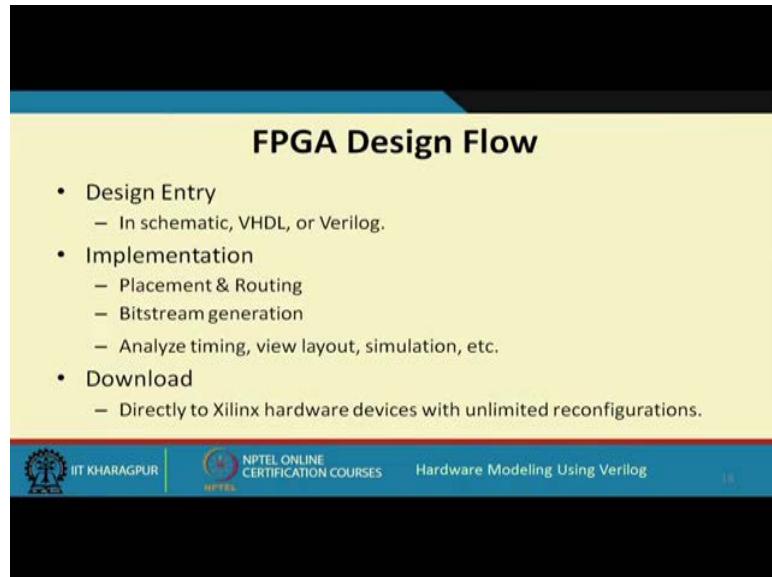
(Refer Slide Time: 18:17)



So, and talking about the routing, see this CLBs are there inside which those LUTs are there and in between, there is some space, there are some switch matrixes with some finite number of wires that are connected and this switch matrix is also programmable like one wire of this can be connected to a wire here, say, a wire here can be connected to a wire here and the CLBs are connected to these wires.

So, by suitably programming the switch matrix and connecting the CLBs to these; so, we can make any kind of connection say as the output of this CLB can be connected to the input over this CLB, ok. So, this is also programmable, routing, interconnection.

(Refer Slide Time: 19:05)



The slide is titled "FPGA Design Flow". It lists four main steps: Design Entry (in schematic, VHDL, or Verilog), Implementation (placement & routing, bitstream generation, timing analysis), and Download (directly to Xilinx hardware devices). The slide is part of an NPTEL online certification course on Hardware Modeling Using Verilog, specifically module 18.

- Design Entry
 - In schematic, VHDL, or Verilog.
- Implementation
 - Placement & Routing
 - Bitstream generation
 - Analyze timing, view layout, simulation, etc.
- Download
 - Directly to Xilinx hardware devices with unlimited reconfigurations.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog | 18

So, then FPGA you say everything is programmable, the logic is programmable, logic in terms of the lookup table, interconnection is program means interconnection is programmable in terms of the switch matrixes and also the I/O blocks can be programmed, they are also programmable. So, when I say in that board, I have shown earlier that I am downloading some data on the FPGA board where actually downloading all this programming data depending on my circuit netlist the synthesis software which is there, it will be generating this programming data and if I download it on my chip I will be getting my desired functionality on that chip, ok, fine.

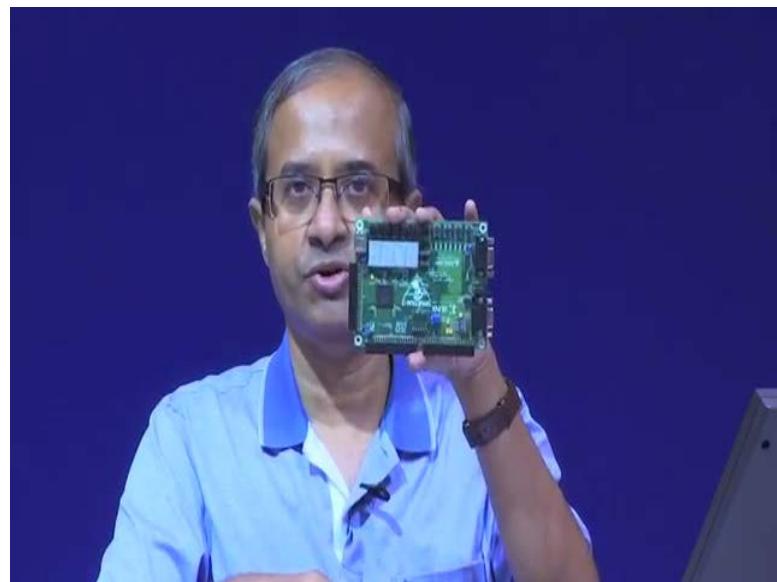
So, FPGA design flow, if you just look at it. So, it consists of design entry to start with, you can do it in Verilog then the software that is provided by the vendor like I talked about Xilinx, ISE or Vivado; they do placement and routing, partitioning, but here it is different, here partitioning means partition in to sub circuit with maximum 4 inputs and one output because each partition I will be mapping to one of the LUTs and this LUTs will be placed into one of this CLBs configurable logic blocks.

Now, if you talk about the switch matrixes, you see there is a finite number of wires, right, but if you have a very bad kind of a placement, you have placed them in such a way that lot of wires need to be connected across a switch matrix, but maybe one matrix is allowing only 4 wires to be laid. So, you cannot complete the routing, see you may have to change the placement and again try the routing.

So, FPGA placement and routing is also a big challenge. So, if we have a good routing, if you have a good placement, then routing will be easy. If a placement is not good may be during routing it will fail, you may have to move some blocks around again try and may ultimately will be getting out some solution. So, this placement and routing, bit stream generation is that programming.

So, whatever you want to program ultimately it is generated in the form of stream of bits. And these software, they allow you to analyze the timing behavior layout that in that within the layout which part is heavily utilized, which part is not utilized of course, you can also do simulation and check with a function it is correct or not then finally, you can do the download, you can directly map it to the Xilinx hardware device, like just talking about that, that FPGA both that I showed you earlier again.

(Refer Slide Time: 22:15)



So, here I told that there is a port here some pins through which you connect a cable to the PC and you can download that bitmap file here to the FPGA board and this FPGA chip will be automatically configured in that way, right, ok. So, with this, we come to the end of this lecture and in the next lecture we will look at the other design styles namely Gate array Semi-custom and the Full-custom design styles.

Thank you.

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

Lecture - 05
VLSI Design Styles (Part 2)

So, in this lecture, we continue with a discussion on design styles in VLSI, if you recall in our last lectures, we had been talking about the FPGA field programmable Gate Array design style and what are its specific features and rules during the design.

(Refer Slide Time: 00:42)



Now we continue our discussion in that line. So, the first design style that we will be talking about today is something called Gate Array.

(Refer Slide Time: 00:50)

Introduction

- In view of the speed of prototyping capability, the gate array (GA) comes after the FPGA.
- Design implementation of
 - FPGA chip is done with user programming,
 - Gate array is done with metal mask design and processing.

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

Now, Gate Array is something that comes right after FPGA, see, the two extremes of the spectrum are FPGA on one side and a pure ASIC on the other side. Now you recall what you had said. So, when you want to manufacture an ASIC, we have to send our design to the fabrication facility, they will be carrying out all the steps of fabrication, they will be manufacturing the chip and there will be sending it back to us, but on the other extreme we have this so-called field programmable Gate Array or FPGA where the entire programming and customization can be done by the user in the lab, ok.

So, the turnaround time is extremely fast, the FPGA kits as I said will cost you not more than few thousands of rupees; at least the simpler ones. So, it is very cheap very fast and you can very quickly map some design into hardware. Now in that spectrum Gate Array is something which falls between the 2 extremes, I mean you may say it is a little closer to FPGA. So, let us see the salient features, now the main difference from FPGA is that you see FPGA, the design customization which we sometimes called programming is entirely done by the user, but for the Gate Array, it is a 2 step process, user is not doing well here again we are sending it to the fabrication facility; to the fab, but the effort is much less, why we shall explain.

(Refer Slide Time: 02:43)

The slide has a yellow header and footer. The header contains the text '(Refer Slide Time: 02:43)'. The main content area contains a bulleted list:

- Gate array implementation requires a two-step manufacturing process:
 - a) The first phase, which is based on generic (standard) masks, results in an array of uncommitted transistors on each GA chip.
 - b) These uncommitted chips can be customized later, which is completed by defining the metal interconnects between the transistors of the array.

The footer contains the IIT Kharagpur logo, the text 'NPTEL ONLINE CERTIFICATION COURSES', and the course title 'Hardware Modeling Using Verilog'. There is also a circular profile picture of a man.

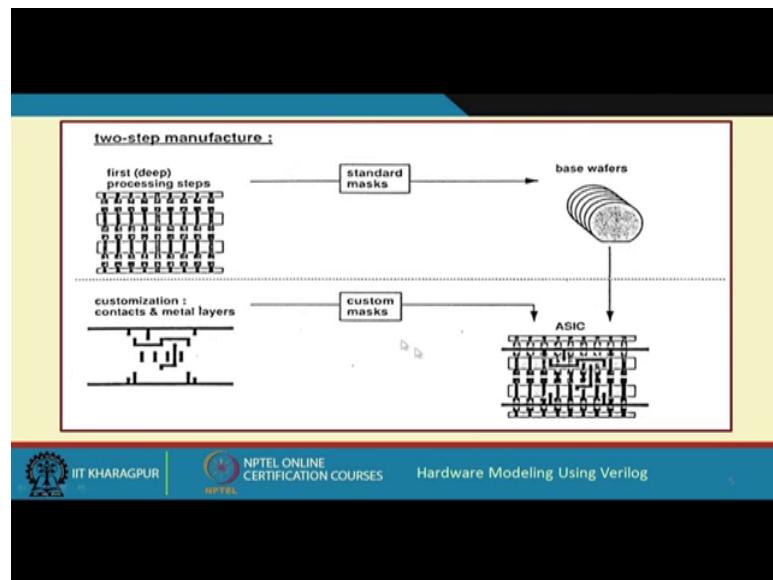
So, for Gate Array implementation, we actually require a 2 step manufacturing process.
So, when we say manufacturing; that means, we are trying to build something.

Now, in a Gate Array chip which will ultimately implement or realize some functionality, there are 2 steps. First step is independent of the function being realized and the second step is a step of customization. So, you see the first; in the first step, you can complete the first step for all the designs in a means irrespective of what is the function you want implement and once you have done that for specific designs, you will have to separately carry out step 2.

So, the first phase which I am saying, this is based on some generic masks during fabrication. So, actually what we are doing here we are actually fabricating a large number of transistors on the chip, but what we are doing, we are not interconnecting the transistors, we are simply fabricating a large number of transistors which is the first phase. Now in this second phase, where you are doing the customization, we are completing the metal interconnects, we are interconnecting the transistors in a way which will help us in realizing the functionality of a circuit, ok.

These are the 2 phases of manufacturing; first phase is a generic say step which is independent of the design, while this second step is design specific where we carry out the interconnects, we interconnect the transistors to form gates.

(Refer Slide Time: 04:53)



And again interconnect the gates to form our desired functionality, ok. So, pictorially the process will look like this, now in the first step which is shown on top. So, you simply create a large number of transistors using standard masks. So, what we get is a number of silicon wafers where the same large array of transistors is fabricated. Now you see when you are fabricating something in a fab, it will cost you money. So, when you are fabricating in millions your means; your total cost will be distributed or divided among the number of, number of items you are manufacturing. So, because you are manufacturing these base wafers in large numbers the total cost of the first step is getting distributed among all these designs. So, that the per design cost of step one becomes very less.

Second step is the actual customization where you complete the metal interconnections. So, the transistors which are fabricated, you interconnect them in some way, right. So, after this, you get your final design.

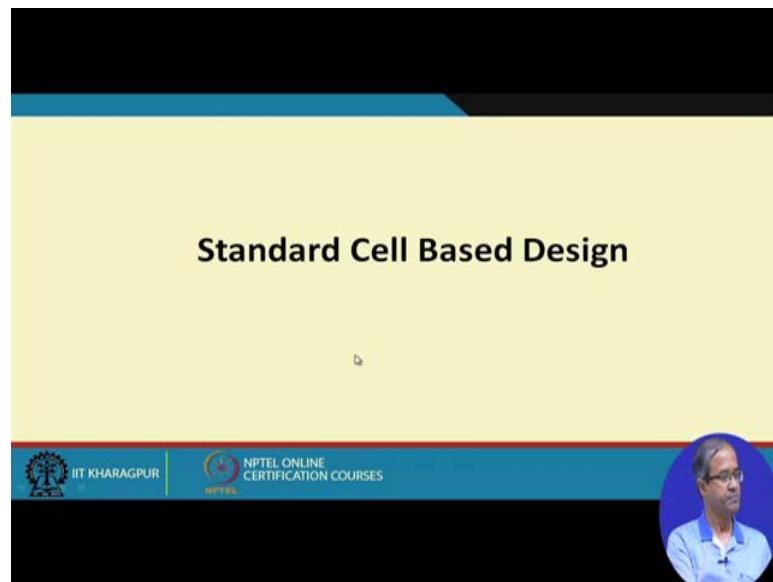
(Refer Slide Time: 06:13)

- The GA chip utilization factor is higher than that of FPGA.
 - The used chip area divided by the total chip area.
- Chip speed is also higher.
 - More customized design can be achieved with metal mask designs.
- Typical gate array chips can implement millions of logic gates.

So, some features of this design style; number one because we are manipulating or working at the level of transistors, clearly the chip utilization factor will be much higher as compare to FPGA because in FPGA chip, in the field programmable Gate Array. If you recall, there was something called configuration logic block the CLBs, inside CLBs, there were the lookup tables or LUTs. Now there apart from the LUTs, there were a large number of other circuitries as well lot of multiplexers, some flip flops whether you need them or not those circuits are also there. So, maybe out of that you are utilizing only 60 percent or 70 percent of the hardware, the rest you are not actually using, ok.

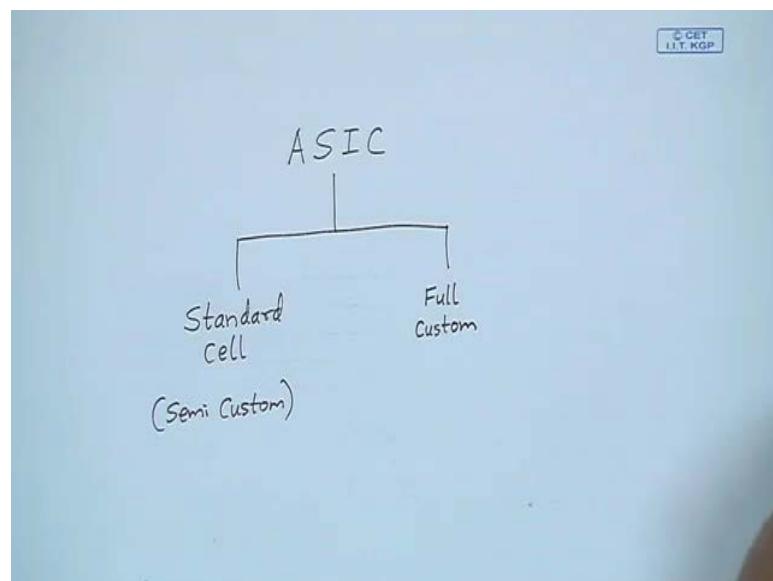
So, in that respect; when you go for this Gate Array kind of design style where you are working at a much lower level, at the level of transistors. So, you can just utilize the transistors in a much better way. So, wastage will be very less here, ok and because you are working at a much lower level, your chip speed in general will also be much higher, ok, because the delay will be less because all those, you have all those complicated circuitries as there in FPGA, they will not be there in Gate Array style. So, a typical Gate Array chip can implement millions of logic gates or even more, ok. So, this is the idea.

(Refer Slide Time: 07:48)



So, next let us move on to a Standard Cell based design style. Now Standard Cell is one way to implement ASICs. So, under ASIC what we are trying to say is that.

(Refer Slide Time: 08:05)



When we talk about application specific integrated circuit or ASIC, there are 2 broad ways of going about the manufacturing. One is called Standard Cell based, this is sometimes also called Semi-custom and the other alternate style is Full-custom. These are the 2 broad approaches. Now in this approaches, the basic idea is that you have to go through all these steps of fabrication. So, your fabrication cost is not changing, right. So, what is changing

will be your design cost. Now in case of Standard Cell, you are using some pre designed cells from the library. You are picking them up and putting them in your design that is your Standard Cell or Semi-custom design whereas, for Full-custom design you are theoretically designing everything from scratch. So, naturally it will take much long time to design a reasonably complex circuit, ok.

(Refer Slide Time: 09:27)

Introduction

- One of the most prevalent design styles.
 - Also called semi-custom design style.
 - Requires developing full custom mask set.
- Basic idea:
 - Commonly used logic cells are developed, and stored in a standard cell library.
 - Typical library may contain a few hundred cells (*Inverters, NAND gates, NOR gates, AOI gates, OAI gates, 2-to-1 MUX, D-latches, flip-flops, etc.*).

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

So, let us see the features of the Standard Cell based design style. So, this is one of the most widely used design style for manufacturing ASICs today. So, as I said, this is also called Semi-custom because it is not Full-custom, lot of the design cells are already pre designed and available to us we are reusing them, but as I said, we have to go through all these steps of fabrication which means we have to develop the full set of masks that are required for fabrication. So, the basic ideas, I had said that you use some simple logic cells which are very commonly used and store them in a library which is typically called the Standard Cell library.

Now, when you store them in the library actually what we store is a highly optimized layout. So, it is not that it is a gate level netlist, we are taking the gate level netlist putting it on our design and later on will be thinking about layout not exactly that we are getting the layout directly, you can put it straight on to silicon, ok, that is idea. So, the typical library for Standard Cell may contain few 100 cells. So, cells may contain simple gates

like Inverters, NAND, NOR or more complex gates like AND-OR-INVERT, OR-AND-INVERT, multiplexer, latches, flip flop, this kind of cells, fine.

(Refer Slide Time: 11:03)

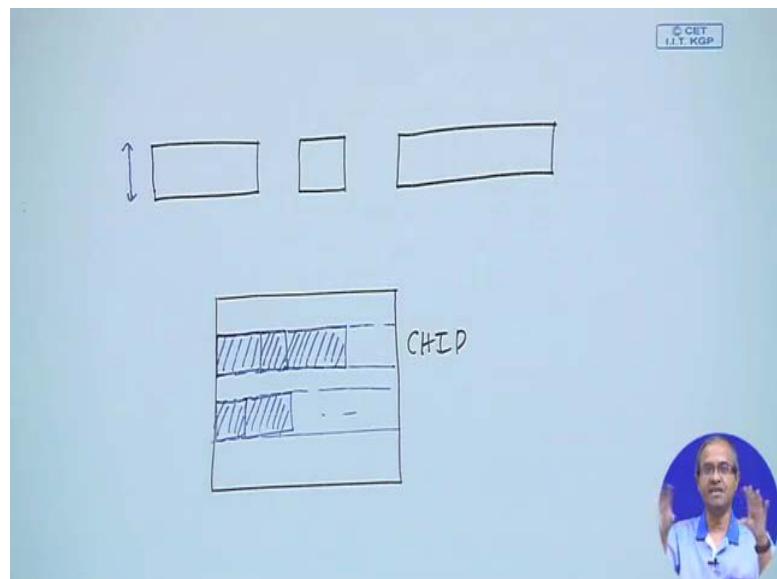
Characteristic of the Cells

- Each cell is designed with a fixed height.
 - To enable automated placement of the cells, and routing of inter-cell connections.
 - A number of cells can be abutted side-by-side to form rows.
- The power and ground rails typically run parallel to upper and lower boundaries of cell.
 - Neighboring cells share a common power and ground bus.
- The input and output pins are located on the upper and lower boundaries of the cell.

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

Now, there is something very unique about these cells, the cells are designed with a fixed height, but the widths may be different. So, the idea is as follows, I am just trying to show you.

(Refer Slide Time: 11:21)

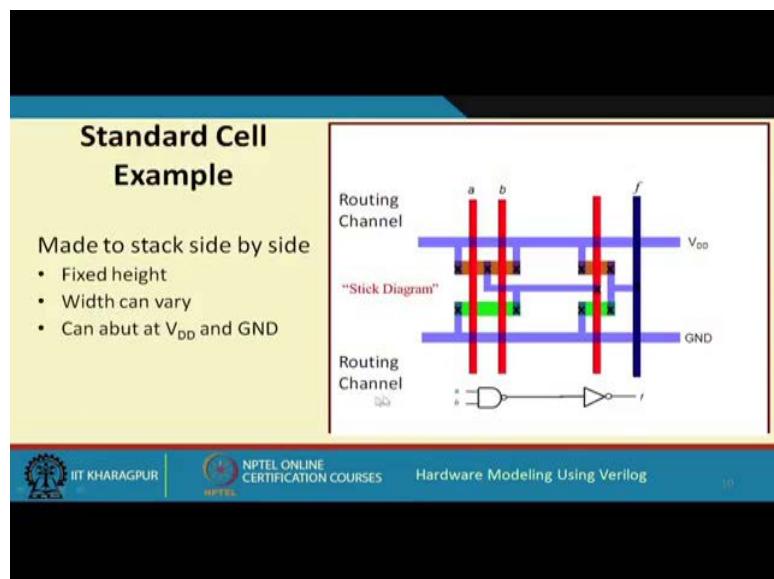


Suppose I have a cell like this which geometrically has a shape of this form, let us say I have another cell which has the same height.

But width is different, there is another cell same height, but the width is again different. So, when we put these cells on a chip suppose this is our final chip. So, here we place all these cells in some well defined rows, suppose this is one row, the height of a row will be equal to the height of this cell and within a row, what we do? We just place the cells one after the other, they will be physically touching each other, right like this. So, you see the layout also becomes very regular all these cells will be arranged in terms of rows. So, some more cells will be placed here, there be some cell here and so on. So, in Standard Cell, the layout looks something like this, very regular where everything is placed in terms of rows. So, in terms of the design; in terms of the routing, in terms of the floor planning, the task of the designer becomes much simpler here, right.

So, just as I said, each cell is designed to the fixed height, the power and ground rails the lines run parallel to the upper and lower boundary, I will just show an example, shortly, the reason is that if you just connect 2 of the adjacent cells side by side, if you just connect them together the power lines, VCC line will also VDD line will touch and also the ground line you will touch.

(Refer Slide Time: 13:35)



So, you will be getting continuous signal for the supplies, power supplies and typically the input and output pins are look at on the upper and lower boundaries. Let us take an example. So, here I am showing an example of a layout. Well, this is a way in which we show a layout. This is called a Stick diagram. So, the blue lines are the metal connections,

red lines are the poly silicon connections, green and brown are the diffusions, here are the n type transistor, here the p type transistors and this black crosses are the, this are interconnections. So, you see here we have implemented 2 gates; 1 NAND gate, 1 NOT gate, this part is the NAND gate, this part is the NOT gate, the 2 inputs a, b.

So, you see the inputs a, b are available vertically here. So, the transistors that make this NAND gate, that 2 transistors here and 2 transistors here and this is the output of this gate. This output of the first gate feeds, the input of the second Inverter, there is an n transistor here, p transistor here. So, you see for both these, here we are showing that the 2 cells are touching each other. So, VDD is running on top ground is running on bottom. So, when they are brought together and touch each other, this VDD and ground lines become continuous. So, this is just an example, ok, fine.

(Refer Slide Time: 15:01)

The slide has a yellow header bar with a blue triangle on the left. The main title is "Floorplan for Standard Cell Design". Below the title is a bulleted list of design requirements:

- Inside the I/O frame which is reserved for I/O cells, the chip area contains rows or columns of standard cells.
 - Between cell rows are channels for routing.
 - Over-the-cell routing is also possible.
- The physical design and layout of logic cells ensure that
 - When placed into rows, their heights match.
 - Neighboring cells can abut side-by-side, which provides natural connections for power and ground lines in each row.

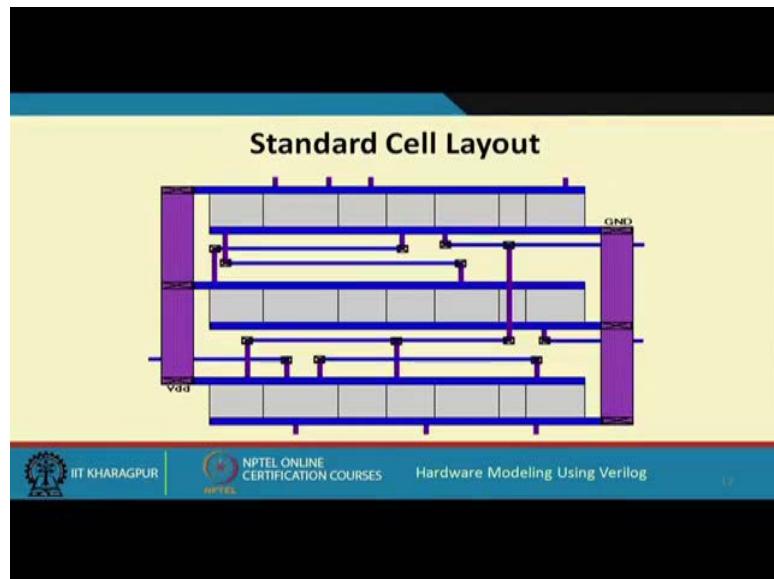
At the bottom, there are logos for IIT Kharagpur and NPTEL, followed by the text "NPTEL ONLINE CERTIFICATION COURSES" and "Hardware Modeling Using Verilog".

So, just as I had shown you. So, when you think of the floor plan of Standard Cells. So, the chip area actually will consist of some rows of Standard Cells or column whichever way you think of.

Now, the design and layout of the cells will; obviously, ensure that when you place them side by side their heights will match and in neighboring cells can abut side by side, this is because of the restriction that you have imposed that all cells have to be of the same height because they are of the same height, you can put the cells side by side and within a row, then any number of cells which can fit in a row, you can put them side by side just touching

each other. So, the VDD will be on top, ground will be on bottom and all the signal lines will be running vertically, ok, they will be interconnected later on using some space which are available within the rows those are called channels, ok.

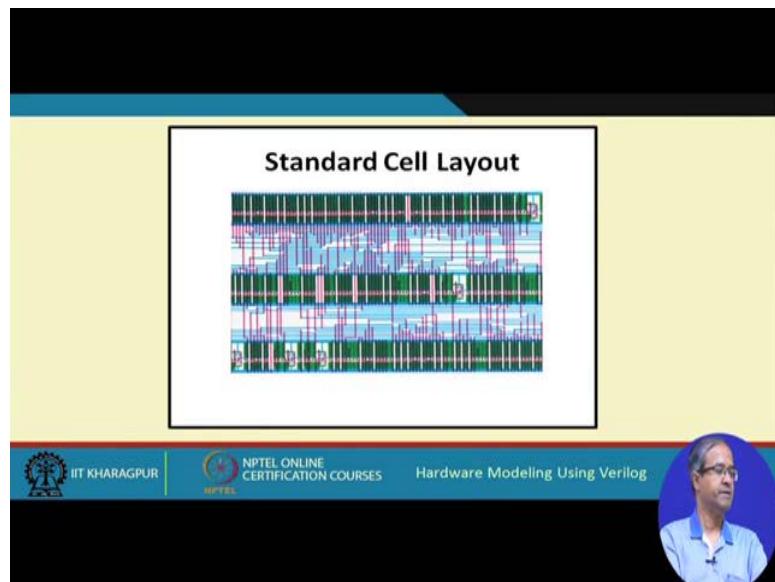
(Refer Slide Time: 16:09)



So, this is a typical Standard Cell layout which shows 3 rows; you see this is one row, this is one row this is one row.

So, the different Standard Cells are shown, you see here; these are the cells and you see some of the interconnections are shown. So, this space between the rows, this part is called Channel, you see some of the interconnections are made like this, ok, well it may so happen that one point here, a pin here wants to be connected or needs to be connected to a pin here. So, what do we do? We have to cross a row. So, there are some special cells also available which are called Through cells. So, what it does? It simply provides the connection path from the top to bottom nothing else. So, you can use one of such cells to make a connection from here to here and then to here, right and the power supply and the ground lines can be fed from 2 side. This is your VDD, it feeds the VDD lines and the other side of ground, it feeds the ground lines. So, this is a typical layout structure of a Standard Cell based design.

(Refer Slide Time: 17:30)



So, a typical Standard Cell layout for a moderately complex design is shown here, these are the cells and the interconnections you see they are quite complex, right, fine. Now let us come to Full-custom design.

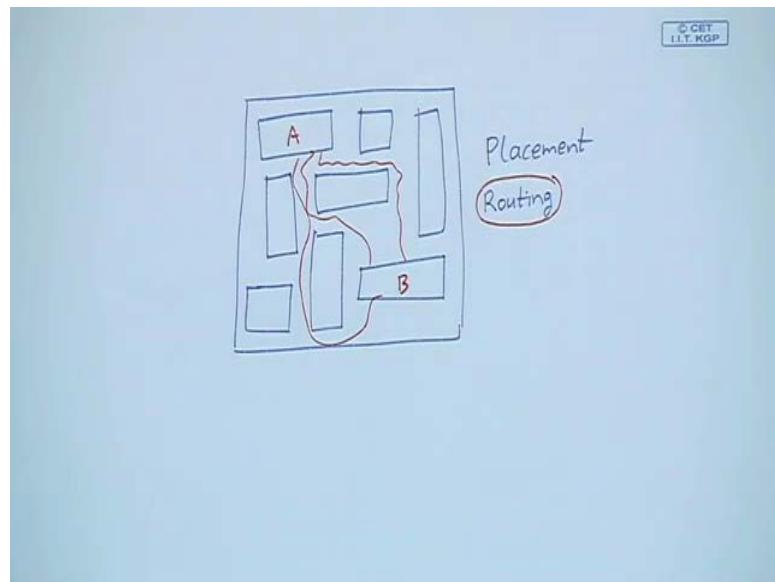
(Refer Slide Time: 17:46)



Now, for the Standard Cell or the Semi-custom design, what we had assumed, we had assumed that these cells are already available to us, their layouts are available, their geometries are very well defined, their heights are same the relative vertical positions of the VDD and ground lines are the same. So, I can simply put the cells side by side, but in

Full-custom design we are not making any such assumption we are saying that our basic blocks in the design can be of any arbitrary complexity, any arbitrary size, some can be small, some can be big, you can place them in to the chip area in whatever way you want like suppose this is your chip.

(Refer Slide Time: 18:37)



So, it is possible that one block is has a shape like this, one block has a shape like this, one has a shape like this, one is like this, one is like this. So, they can come in any arbitrary shapes and sizes, ok.

So, you can see that here the problem is much more difficult, here some very important problems pertain to placement of the blocks; how to place the blocks or in what way should I place the blocks. So, that my interconnection which is called routing becomes easier because say I may do one thing, I may place in such a similar in one way, let us say a block A and a block B here, they have a large number of connections between them. So, all these connections have to be brought either from here or from here or from some other direction. So, if you do not have sufficient space here for the connections, you may see that your routing fails. So, if your routing fails, you may have to go back, you may have to modify the placement and again retry, ok. So, this is not a very easy task, this is a quite difficult task, alright.

(Refer Slide Time: 19:58)

Introduction

- Standard-cells based design is often called semi custom design.
 - The cells are pre-designed for general use.
- In the full custom design, the entire mask design is done anew without use of any library.
 - The development cost of such a design style is prohibitively high.
 - The concept of design reuse is becoming popular to reduce design cycle time and cost.

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

So, this Standard Cell based designed which you had seen, this is called Semi-custom design, semi because the cells that we are using those are pre designed, but the way we are placing the cells that is up to us, right, but in the Full-custom design, we are theoretically, will no one will do it this way today, theoretically we are not using any library. So, the entire design is done from scratch, but; obviously, for the modern day VLSI chips where the number of transistors can run into the, into the order of billions, if you say that I will be doing everything from scratch. So, it really becomes impractical. So, it will take you years and years to design a particular chip, right. So, as I have mentioned here, the development cost of such a design style may become prohibitively high therefore, all the designers today, they use a concept called design reuse like I am just take an example.

Suppose you are coming up with a new, is a new design of a processor, you said I am developing a new CPU, I want to manufacture it, fabricate it. So, should I design everything from scratch well, I may see that well, I require a 32-bit fast multiplier which I had already designed earlier for some other processors. So, why not I take the same design from there and use it here. This is the concept of design reuse. So, today nobody designs everything from scratch, whatever is available, these are sometimes called IP cores, Intellectual Property cores; they are taken from different sources, they might be put together in a same place and you can integrate them in a suitable way, ok, fine. So, this is what is called design reuse, which is becoming very popular nowadays this; obviously, reduces the total time for design and also the cost they are related, fine.

(Refer Slide Time: 22:19)

- The most rigorous full custom design can be the design of a memory cell.
 - Static or dynamic.
 - Since the same layout design is replicated, there would not be any alternative to high density memory chip design.
- For logic chip design, a good compromise can be achieved by combining different design styles on the same chip.
 - Standard cells, data-path cells and PLAs.

So, with respect to Full-custom design, there is one kind of chip which always requires Full-custom design that is the memory cell because memory is the, you can say the one design where you pack the maximum number of transistors in a chip, well you look at 2 chip side by side, one a processor, one a memory chip. In a memory chip because the layout of the transistors the way you are putting them are extremely regular and you have millions of such rows and columns you are using.

So, the design becomes very compact. So, you can put in much more number of transistors as compared to a processor design where your layout may not be that regular. In a memory, there will be a large number of rows, large number of columns with some transistors or some cells sitting at the junctions, very regular kind of layout, right. So, memory is one such design where Full-custom design is religiously practiced. So, if you design one cell, the layout of one cell you can simply replicate it a billion times.

Let us say; so, it becomes easy, but for a design like a processor where there is a lot of logic then you can have a lot of mix and match, some part you may implement in Standard Cell, some part you can have the blocks which are pre designed, some parts you can use something called programmable logic arrays and so on, ok.

(Refer Slide Time: 24:06)

The slide contains the following bullet points:

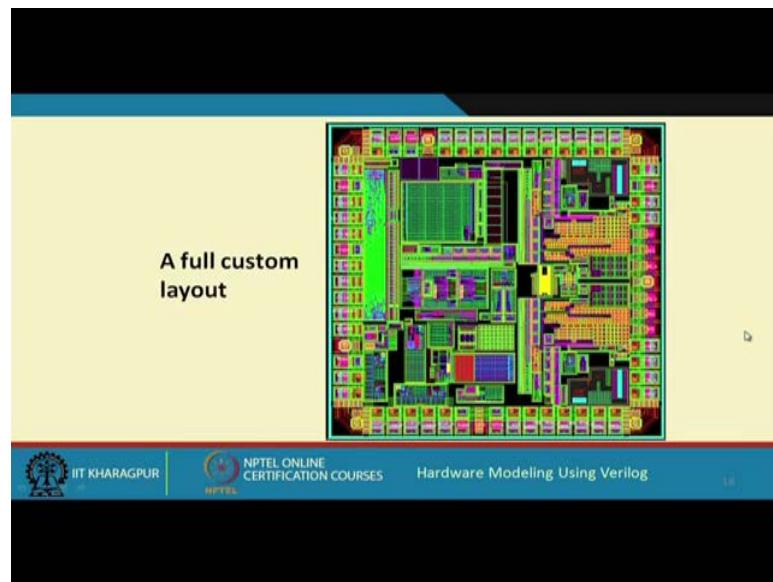
- In real full-custom layout in which the geometry, orientation and placement of every transistor is done individually by the designer.
 - Design productivity is usually very low (typically 10 to 20 transistors per day, per designer).
- In digital CMOS VLSI, full-custom design is rarely used due to the high labor cost.
 - Exceptions to this include the design of high-volume products such as memory chips, high-performance microprocessors and FPGA masters.

At the bottom of the slide, there is footer information: IIT KHARAGPUR, NPTEL ONLINE CERTIFICATION COURSES, and Hardware Modeling Using Verilog. The slide number 12 is also visible.

So, if you want to go for a Full-custom layout, which is religiously, you are trying to start from scratch, where orientation and placement of every transistor has to be done individually by the designer then you think. So, statistically it has been found that the design productivity for this kind of a design philosophy is typically 10 to 20 transistors per day per designer. So, even if a company puts in 1000 engineers for this design, you just imagine for a one billion transistor design how long will this take, right, because of the high labor cost and time, this is actually never done.

So, memory, very high performance microprocessors like the one's INTEL manufacture for example, which are sold in millions and FPGA chips, the masters they are also sold in large quantities, these are some designs while you can put in some effort in order to improve their performance, ok.

(Refer Slide Time: 25:19)



So, this is just an example of a Full-custom layout. So, the way it looks like, you see there are different regions, they are all marked in different colors you can see, this is one regular region, this is some region where there is not much regularity and so on, ok.

(Refer Slide Time: 25:42)

Comparison Among Various Design Styles				
	Design Style			
	FPGA	Gate array	Standard cell	Full custom
Cell size	Fixed	Fixed	Fixed height	Variable
Cell type	Programmable	Fixed	Variable	Variable
Cell placement	Fixed	Fixed	In row	Variable
Interconnect	Programmable	Variable	Variable	Variable
Design time	Very fast	Fast	Medium	Slow

At the bottom, there is a footer bar with the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, the course title "Hardware Modeling Using Verilog", and a page number "19".

These are the I/O cells, input output cells. So, just to compare among the design styles, we just now talked about FPGA, Gate Array, Standard Cell and Full-custom. So, we have ordered the designs in this way because FPGA is easiest, Full-custom is the most difficult. So, we are comparing this with respect to these parameters, cell size, cell type, placement,

interconnect and the total time for design. Self-size in FPGA, everything is prefabricated right this CLB, the LUTs, they are fixed Gate Array also the transistors of the cells they are fixed. Standard Cells; cells are not fixed, but the heights are fixed, but Full-custom, you have complete flexibility, they are variables.

Cell type in FPGA, you have the LUTs, but you can implement anything by programming them. So, this is programmable, but in case of Gate Array there are some Gate Arrays where there are a large number of NAND gates which are built. So, the cell type is also fixed, they are NAND function, but Standard Cell, there are various different kinds of cells, Full-custom also there can be different kind of cells. Cell placement in FPGA and Gate Array are fixed because the cells are already there, in Standard Cell you have some restriction, cells can be placed along rows, but in Full-custom, you can place them anywhere you want, similarly for the interconnect, for FPGA well again this is programmable, but for the other 3 styles, interconnect has to be done based on the user requirement, they are variable. Design time, FPGA is very fast, Full-custom is the slowest, Gate Array is here, Standard Cell is here, Gate Array of course faster than Standard Cell.

So, this is how the whole thing works. So, with this we come to the end of this lecture. Well, over the last 2 lectures; this one and the previous one, we have talked about the different design styles, well, most of you, most of the time will probably be using the FPGA based design style only and there as the programming vehicle, you will be using some harder description language like Verilog or VHDL as I had said, there are other languages also available, many people they also use those languages.

So, in our next following lectures, we shall now be moving in to the details of the Verilog language; some syntax, some semantics and we shall be illustrating them with examples and we shall be progressing to more and more complex features and complex designs.

Thank you.

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

Lecture – 06
Verilog Language Features (Part 1)

So, we now start our discussion where will be talking about some of the features of the Verilog language. So, the title of this lecture is titled Verilog language features; part one. Well, talking about Verilog, you have already seen some examples of sample Verilog programs. So, what we had seen is that the essential component of a Verilog program or Verilog code is a module, just like in a high level language like C; you have function as the basic building block. So, the program will consist of one or more functions. So, in a Verilog program or code implementation, it will consist of one or more modules.

(Refer Slide Time: 01:12)

Concept of Verilog “Module”

- In Verilog, the basic unit of hardware is called a *module*.
 - A module cannot contain definition of other modules.
 - A module can, however, be *instantiated* within another module.
 - Instantiation allows the creation of a *hierarchy* in Verilog description.

```
module module_name (list_of_ports);
    input/output declarations
    local net declarations
    Parallel statements
endmodule
```

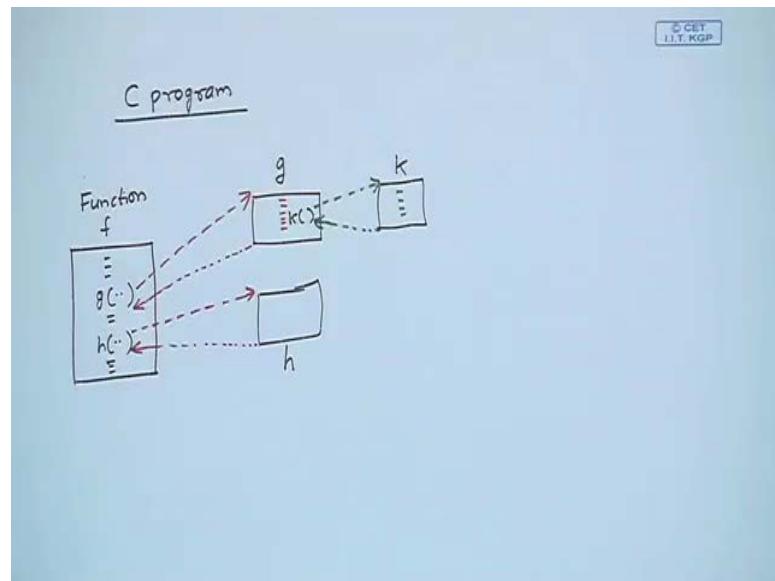
 IIT KHARAGPUR |  NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

So, the concept of Verilog module is important. So, as I had said the basic building block or basic unit of hardware in Verilog is called a module, there are some restrictions like a module cannot contain definitions of other module, well, this is some similarity with a language like C. Let us say in C, whenever you define a function, you cannot define another function inside that function. So, the function definitions have to be disjoint. So, in a similar way, here also the module definitions have to be disjoint, this is one. So, a

module cannot contain definition of other modules. A module can be instantiated within another module.

So, here we have a mark difference between how a program like C works and how Verilog works.

(Refer Slide Time: 02:29)

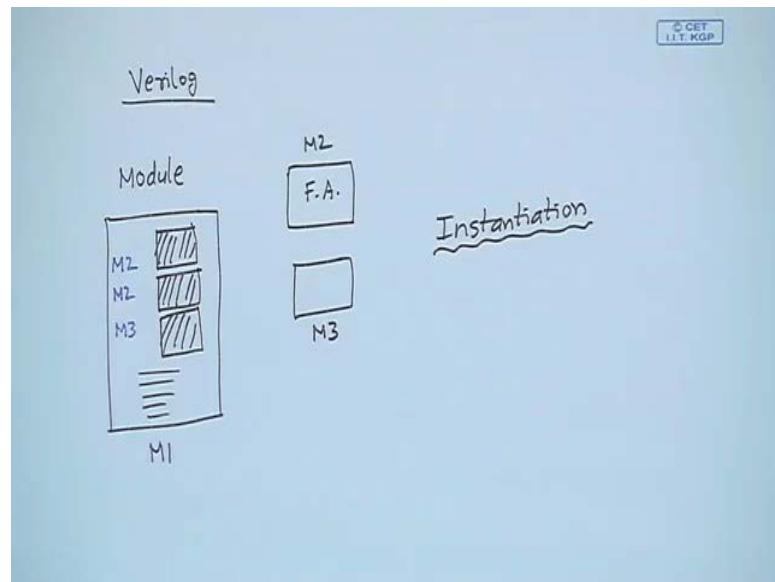


So, let me try to just explain, well in a C program, let us say if I talk of a high level language like C, let us say there is a function f, this is the body of the function, there is a function g, there is another function h. So, within the body of the function, I can call g with parameters, I can call h, ok, and so on. So, in a typical program execution, whenever function is called, what happens; that when this function call is encountered, control will be transferred to this function. The code of this function will be executed and then control will be returned back to this statement following this g.

Similarly, when this h is encountered, again control will be transferred to this function h, it will be finished and then again it will return back. Now, this can happen for nested call also, suppose there is another function. Let us say k where from inside g you have called k, right. So, when you call k from inside g, again a similar thing will happen, k will be called, it will be executed and then it will be returned, ok. So, if you look at the thread of execution, during execution you encounter g, you go to g, you execute, encounter k, you go to k, finish it, then come back, then come back.

So, this is like a last in first out, the last function to be called will be the first one to return right. So, in C program, irrespective of how many times you call the function, your size of the total program does not increase, just you are simply calling one function from another function, right. So, your program code is not increasing in that sense, but in Verilog the concept is slightly different because here we are talking about hardware. So, in Verilog, we have the concept of modules.

(Refer Slide Time: 05:00)

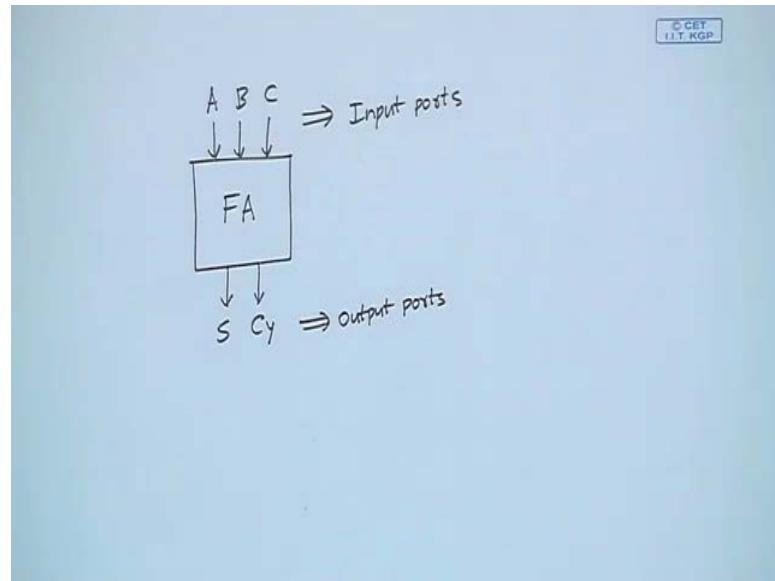


Let us say, I have a module here, let us call this module is M1. So, what I do from inside this module well I am invoking, let us say just a following the terminology of C, I am calling a module M2, I am calling it 2 times, I am calling another module M3, let us say. So, my M2 module is already defined, my M3 module is already defined. Now, what will happen here whenever these modules are invoked. So, it is not exactly like calling a function in C, like here for example, you are invoking M2 2 times. So, what will happen inside this module, 2 copies of M2 will get embedded. Suppose let us say, this is a full adder. So, if you call it 2 times. So, 2 copies of full adders will be put inside this master module, let us say M3 is some other module you call it, this M3 will also be embedded. Now in addition to these, this module M1 can contain other statements also. So, this process of calling a module and the module getting inserted in the design in the hardware this is called instantiation.

Because we are talking about hardware, there is no concept of calling hardware and returning, well if you call it 2 times it means you are saying that I want 2 copies of that hardware. So, 2 copies are instantiated. So, M3 is called once means I need one copy of M3 just like that, ok. So, recall the example that we showed you earlier of a 4-bit ripple carry adder, there we had instantiated a full adder 4 times. So, that 4 copies of those full adders were included, means in our ripple carry adder design, ok, fine. So, this is what I just mentioned. So, a module can be instantiated within another module and this instantiation process, this allows the creation of a hierarchy, let again talking about the ripple carry adder example you see, ripple carry adder, the highest level as I said will consist of 4 full adders.

Now every full adder will consist of a sum and carry circuitry and every sum and carry circuitry if we implement using a structural description it will consist of some gates. So, this is a hierarchical description. So, the top level, we are seeing a full, just an adder, 4-bit adder, next level we are seeing 4 full adders, next level we are seeing 4 carry circuitry and 4 sum circuitry and in the lowest level the gates, right. So, a Verilog module in terms of its syntax contains the key word module to start with and it ends with endmodule. And after module will come the module name just like, just like in C, you have a function name, module name and within parenthesis list of parameters, these are actually technically called ports and at the end there is a semicolon, this has to be given. Now this is ultimately some hardware we are trying to design, let us say I am building a harder, if this is a full adder.

(Refer Slide Time: 09:11)



So, you know that for a full adder there are 3 inputs A, B, C, this is a full adder and there are 2 outputs sum and carry out. So, this A, B, C, S and Cy, these are called ports, these 3 are so-called input ports and these 2 are so-called output ports.

So, in the declaration, first you have to declare all the input output ports then inside your design you may be requiring some temporary connection like in the gate level netlist. This sum and the carry, the description that was shown some of them you recall, there some temporary signal lines wire were used, those are these local nets and after that there will be some statements, now here we are calling them as parallel statements. Well, why we will call them parallel? you see in Verilog, there are certain constructs which you will be learning one by one.

Now, you see ultimately we are describing some hardware, we are describing one piece of hardware say h1, another hardware h2, those are all part of the module. Now, in terms of its operation h1 and h2 are both working in parallel because they are actually circuits, they will always be fed with some inputs and they will be generating some output. So, it is not that you cannot say each one that well you wait till h2 is complete then only you start. So, you cannot tell that to hardware like that because ultimately it is a circuit and circuit will respond whenever some inputs are changing, right. So, these are so called parallel statements which you shall see later in some more detail.

(Refer Slide Time: 11:17)

The slide shows a Verilog module definition:

```
// A simple AND function
module simpleand (f, x, y);
    input x, y;
    output f;
    assign f = x & y;
endmodule
```

To the right, a note explains the behavioral description and synthesis options:

This is a behavioral description. The synthesis tool will decide how to realize f:

- a) Using a single AND gate
- b) Using a NAND gate followed by a NOT gate.

Let us take a simplest of simple example just an AND function we have already seen examples like this earlier using the assign statement. So, we have this module description, name of the module is simple AND and the port list is f, x and y and you specify x and y as inputs, input is a keyword, output is another keyword, f is output and assigned f equal to x this is logical AND. Well, if you just mention x, y and f, it will mean, these are all one bit variables; that means, they are logic signals. So, when I say ampersand. So, logically speaking you may see that well I am talking about an AND gate, but well I should say, we are talking about an AND function not necessarily an AND gate because in this case.

So, you are specifying the behavior, you are saying that the output will be the AND function of x and y, but when you give this description to the synthesis tool, synthesis tool will be actually generating the hardware. Now, synthesis tool if it wants it can generate a single AND gate, but alternately it can generate a NAND gate followed by a NOT; that is also AND. You see this second one may be more you can say more probable because in CMOS technology, it is much easier to design and implement NAND, NOR and NOT gates rather than AND and OR gates. So, this depends on the synthesis tool. So, you really do not know for sure that what will be your final gate netlist that this synthesizer will be generating, ok.

(Refer Slide Time: 13:15)

The slide contains the following text and diagrams:

This is also behavioral description.

- One possible gate level realization is shown.
- t1 and t2 are intermediate lines; termed as wire data type.

Verilog HDL code:

```
/* A 2-level combinational circuit */
module two_level (a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire t1, t2; // Intermediate lines
    assign t1 = a & b;
    assign t2 = ~c | d;
    assign f = ~(t1 & t2);
endmodule
```

Logic diagram:

So, let us take another example, this is a 2 level combinational circuit. So, there are 5 parameters, the module name is 2 level. So, A, B, C, D are inputs, output is f, you see here we have declared some intermediate nets t1 and t2, ok, and there are 3 assign statements, t1 is doing an AND, t2 is doing a OR and then NOT basically NOR and f1 is doing AND and NOT basically NAND. So, let us show the natural implementation, t1 equal to a and b, suppose we implement it using a NAND gate, t2 equal to NOR of C and D, we implement using basically an NOR gate and finally f we implement using a NAND gate. So, this is a 2 level design and using assign, we have defined it in some kind of a structure because we have also mentioned the interconnection. This t1 we have used here, we have used here on the right hand side, t2 we have assigned, t2 we have used on the right hand side, ok. So, this intermediate wires t1 and t2, they are serving the purpose of connecting these 3 gate outputs t1, t2 and finally, this f, ok.

This is again just one possible gate realization because the synthesis tool can generate some other realization also for the same function. So, just one thing you remember that whenever you are using assign statement for describing some design, well, just one thing let me tell you, the assign statement for most practical purposes are used to specify only combinational circuits, but later on I shall give you an example where assign statement can also be used to specify a sequential circuit, this we shall see later, right. So, whenever you are using assign statements one thing you remember that you are actually talking about a behavioral description, right, ok.

(Refer Slide Time: 15:41)

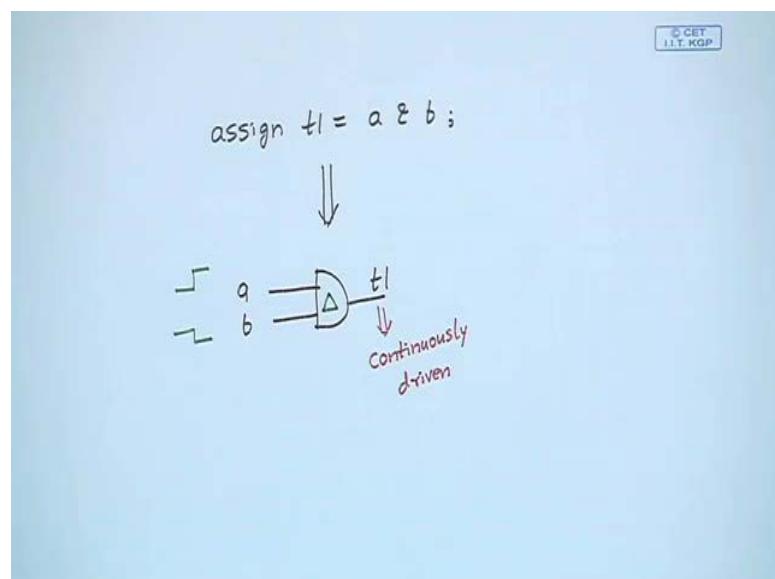
• Point to note:

- The “assign” statement represents continuous assignment, whereby the variable on the LHS gets updated whenever the expression on the RHS changes.
assign variable = expression;
- The LHS must be a “net” type variable, typically a “wire”.
- The RHS can contain both “register” and “net” type variables.
- A Verilog module can contain any number of “*assign*” statements; they are typically placed in the beginning after the port declarations.
- The “*assign*” statement models behavioral design style, and is typically used to model combinational circuits.

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

Now, there are some points to note for the assign statements, first it is that the assign statement represents something called continuous assignment. So, what you mean by continuous assignment? where the variable on the left hand side. So, let us say I have an expression like this assign some variable equal to some expression, let us say in a previous example, assign t1 equal to a ampersand b, this is the expression. So, the variable on the left hand side will get updated whenever the expression on the right hand side changes.

(Refer Slide Time: 16:26)



So, what I mean is that that when I write, let us say assign t1 equal to, let us say a ampersand b. So, possibly this will be mapped in to a hardware like this. So, we say this output signal t1, this is continuously driven. So, what is the meaning of this? continuously driven means whenever this input signal a and b changes, let us say a changes from 0 to 1 or b changes from 1 to 0, some change is there. So, whenever the input changes that will cause an immediate change in the output value depending on the functionality of the gate, of course there will be a delay of the gate, after the delay, this will change. So, there is nothing that says that well, all the changes have to be synchronized with a clock. So, when a clock comes only then you change the output nothing like that the output signals t1 is continuously being driven by the functionality of the gate, the output of the gate, whenever the input of the gate changes, the output will also change and immediately the value of t1 will change, ok, this is the idea, continuously driven.

So, some constraints are there, well of course, net and register type variables, we shall be studying later. So, just assume for the time being that there are 2 kinds of variables net and register and wire which were used, this is an example of a net type variables. So, this constraint says that this left hand side; this variable must always be a net type variable, but the right hand side can be either a register type or a net type, left hand side cannot be register, ok, this is the constraint. Now as the previous example shows. So, a module can contain any number of assignment statements, this assign statements and as a matter of convention this assign statements are placed towards the beginning of the code, right after the declaration of the ports. So, after your port line declarations and the temporary variables wires, you place this assign statements after that. And this is what I have said repeatedly that assign models the behavioral design style and is typically used to model combinational circuits, but you can also model sequential circuits using assign, this we shall see later.

(Refer Slide Time: 19:47)

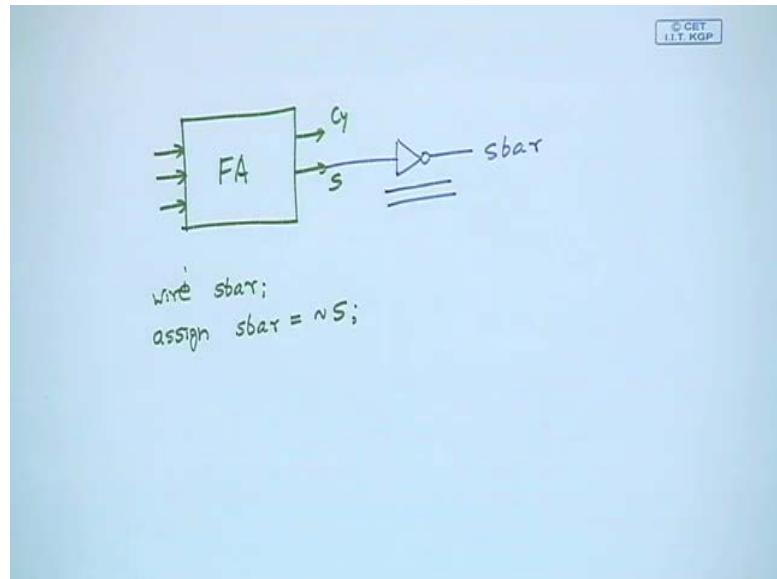
Data Types in Verilog

- A variable in Verilog belongs to one of two data types:
 - a) Net
 - Must be continuously driven.
 - Cannot be used to store a value.
 - Used to model connections between continuous assignments and instantiations.
 - b) Register
 - Retains the last value assigned to it.
 - Often used to represent storage elements, but sometimes it can translate to combinational circuits also.

Now, we have just now talked about net and register, let us see what these are now. So, in Verilog, whenever you declare a variable just like in C whenever you declare a variable, you declared it as either an integer, float, character and so on. So, in the same way whenever you declare a variable in Verilog, you have to declare them with a particular given type. So, broadly speaking data types fall under 2 categories, one is net other is register. Net, just in the example that we took earlier sometime back, net is continuously driven typically the output of a gate or output of a functional block that is connected to a net type variable. So, whenever that output of that block changes, the variable that is connected to it, changes in a continuous way that is why you call it continuous assignment, ok.

But we cannot use a net to store a value, well whenever it changes, you assign something again in the next time when you see its value will depend on the input of that gate or that block. So, it will not hold the value for a long time, ok, just like a flip flop or a latch, you cannot store a value for a longer duration. Typically, a net is used to model connections between continuous assignments and instantiations, what do mean by here?

(Refer Slide Time: 21:14)



Let us say, suppose I have done an instantiation in a module, let us say. So, there is a fulladder I have instantiated. So, there are 3 inputs and 2 outputs, this is Carry and this is S. So, what I say in my module, I have written somewhere that assign, let us say, Sbar equal to NOT of S, where I have defined Sbar as wire. So, actually what this will do? this will take the output from this instantiated full adder and this assign statement will just be an inverter, it will be driving Sbar and this assignment will be continuously driven, whenever S changes Sbar will change immediately, right, this is the basic idea.

Similarly, the register type variable, you can assign a value and the register is supposed to retain or store the value, you can use it again later, ok. Now as the name register implies well, you may be tempted to think that a register is just like a hardware register that is constructed using flip-flops, well do not confuse the two. This is a register type variable in Verilog and when you talk about hardware registers that is actually the implementation flip-flops, ok. So, here, even if you declare a variable of type register; so, it is often used to represent storage elements, all right.

(Refer Slide Time: 23:19)

(a) Net data type

- Nets represent connection between hardware elements.
- Nets are continuously driven by the outputs of the devices they are connected to.
 - Net "a" is continuously driven by the output of the AND gate.
- Nets are 1-bit values by default unless they are declared explicitly as vectors.
 - Default value of a net is "z".



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

But there can be instances where a register can translate into combinational circuits also, ok. So, this you have to remember. Now coming to the net data type, we have already explained in some detail. Net actually represents connection between hardware elements and suppose this output, a is a net, this is continuously driven from the output of a gate, they are continuously driven by the output of some device which they are connected to.

So, whenever the input of this gate changes, this gate output will change and the value of a will also change immediately.

So, when you declare a net using say wire for example, say wire a. So, by default it will be taken as a 1-bit value, but we shall see that we can also declare vectors where instead of 1-bit value, we can use a collection of lines like I can declare a variable to be a 4-bit variable, 8-bit variable, 16-bit and so on. So, I can define something called a vector and use these vectors in my Verilog description or coding also, ok, this will see. And one thing when you declare a net, but you have not initialized yet, so, by default the value is considered to be in the high impedance state z, refers to high impedance or the tri state, right.

(Refer Slide Time: 24:49)

The slide is titled "Hardware Modeling Using Verilog" and is part of an NPTEL online certification course at IIT Kharagpur. It lists several Verilog net data types and their characteristics:

- Various "*Net*" data types are supported for synthesis in Verilog:
 - wire, wor, wand, tri, supply0, supply1, etc.
- "*wire*" and "*tri*" are equivalent; when there are multiple drivers driving them, the driver outputs are shorted together.
- "*wor*" and "*wand*" inserts an OR and AND gate respectively at the connection.
- "*supply0*" and "*supply1*" model power supply connections.
- The Net data type "*wire*" is most common.

So, it means in Verilog, there are a large set of net data types that are supported, some of the important ones I am showing here - wire, this is wired-OR (wor), wired-AND (wand), tri-state, supply0 and supply1, this wire and tri they are equivalent. So, when the multiple drivers driving them, the driver outputs are shorted together. I will just show an example later. This wired-OR and wired-AND, there also outputs are shorted together, but there will be a OR and AND function at the junction, well, again I have an example, here I shall show. And sometimes you may need to specify that which is your power supply line VDD and ground. So, there are 2 special data types supply0 and supply1 that indicate connections to ground and connection to VDD, net type wire of course, is most commonly used.

(Refer Slide Time: 25:57)

The slide shows two side-by-side Verilog code snippets. The left snippet, titled 'use_wire', declares 'f' as a wire type and uses assign statements to map it to A&B and C|D. The right snippet, titled 'use_wand', declares 'f' as a wand type and also maps it to A&B and C|D. Below each code block is a note about the resulting function.

```
module use_wire (A, B, C, D, f);
    input A, B, C, D;
    output f;
    wire f;
    // net f declared as 'wire'

    assign f = A & B;
    assign f = C | D;
endmodule
```

For $A = B = 1$, and $C = D = 0$,
f will be indeterminate.

```
module use_wand (A, B, C, D, f);
    input A, B, C, D;
    output f;
    wand f;
    // net f declared as 'wand'

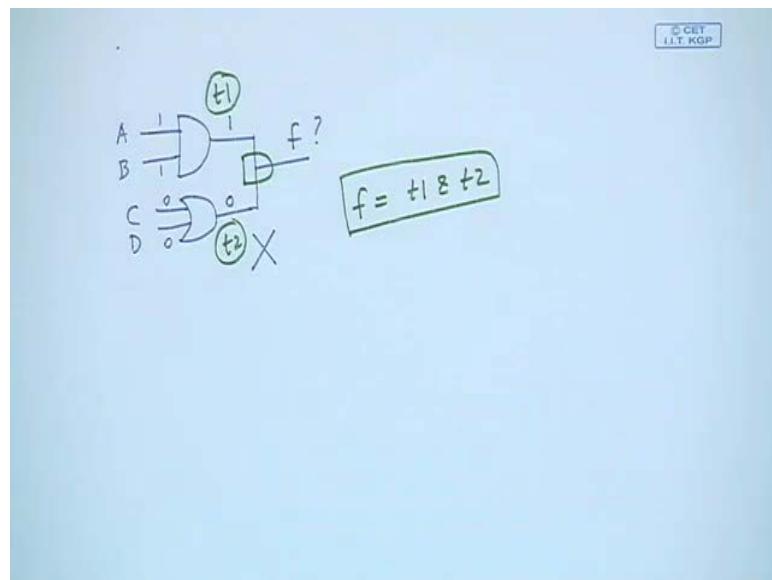
    assign f = A & B;
    assign f = C | D;
endmodule
```

Here, function realized will be
 $f = (A \& B) \& (C | D)$

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

Let us take some examples, here I have a module, 5 ports are there, A, B, C, D are input, output is f. Now here I am declaring f as type wire since output it, here I can specify the type of the output separately and you see I have declared or defined 2 assign statements, both are assigning value to f; f equal to A AND B, f equal to C OR D.

(Refer Slide Time: 26:38)



So, actually this is something one AND gate which is A, B; there is another OR gate, inputs are C and D, both output are f. So, actually what you are saying that as if the outputs are shorted together and you are calling this f.

But you see this is not a normal practice and this is something which is prohibited in design. So, if I apply A1, B1 and C0, B0. So, one gate is trying to make the output 1, other gate is trying to make the output 0. So, what should be f? It will be indeterminate, right. So, for certain condition the output f can be indeterminate, this is a wrong design, you should not use this, but whenever you want to tie the outputs together, you can use this either wired-AND or wired-OR. You see instead of wire, I use a wired-AND here, what is wired-AND means, meaning. So, wired-AND means that in this circuit there is an implied AND gate here at the junction. So, so if you call this 2 lines as t1 and t2.

So, f will actually be t1 AND t2. So, now, the output f is very well defined. So, if you declare output f as a wired-AND then an implicit AND operation will be realized at the connection. So, if you declare it as wired-OR then a wire, then here OR operation will be done. These are all means, operations or operators which have a direct correspondence with the hardware. So, in hardware we have this concept of wired-AND and wired-OR connections depending on the logic family, if you tie the output of 2 gates together for example, for CMOS, it works like a wired-AND, if you connect the outputs of 2 CMOS gates together the output will be 0, if any one of them is 0, that is how it works.

(Refer Slide Time: 28:44)

```
module using_supply_wire (A, B, C, f);
    input A, B, C;
    output f;
    supply0 gnd;
    supply1 vdd;
    nand G1 (t1, vdd, A, B);
    xor G2 (t2, C, gnd);
    and G3 (f, t1, t2);
endmodule
```

supply0 and supply1 have the greatest signal strength.

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

So, let us take another example where we have also used supply lines, supply0 and supply1. I have defined a variable called VDD which is connected to supply1, a variable called ground which is connected to supply0 and we have instantiated 3 gates NAND,

XOR, AND. Here for some of the inputs, I have used VDD and ground also. So, this I can do if I want.

(Refer Slide Time: 29:18)

Data Values and Signal Strengths

- Verilog supports 4 value levels and 8 strength levels to model the functionality of real hardware.
 - Strength levels are typically used to resolve conflicts between signal drivers of different strengths in real circuits.

Value Level	Represents
0	Logic 0 state
1	Logic 1 state
x	Unknown logic state
z	High impedance state

Initialization:

- All unconnected nets are set to "z".
- All register variables set to "x".

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

Just an illustration that how you can use the supply wires. So, data value and signal strength means, Verilog supports this 4 logic values 0, 1, x and z. 0 is logic 0, 1 is logic 1, x is unknown which is not initialized it can be 0, it can be 1, but z is high impedance state. Now when you initialize, so, all unconnected nets, the wires they are initialized to z, but if you declare some variables of type register, registers are initialized to x. Well in addition to the values, there are strength levels also, see strength level actually, they will talk about something like this if I connect 2 signals together.

So, which of the signal is stronger if my first signal is stronger, then the final output will be dominated by the stronger signal that is the concept of signal strength. So, in Verilog, there are 8 signal strengths which are defined, while we should be talking about this later because this signal strength comes in to the picture particularly when you talk about MOS transistors.

(Refer Slide Time: 30:25)

Strength	Type
supply	Driving
strong	Driving
pull	Driving
large	Storage
weak	Driving
medium	Storage
small	Storage
highz	High impedance

Strength Increases ↑

- If two signals of unequal strengths get driven on a wire, the stronger signal will prevail.
- These are particularly useful for MOS level circuits, e.g. dynamic MOS.

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

So, in circuits is the MOS transistors, there are various points in the circuits where the signal strengths are different. So, if I connect 2 points together, their signal values are different, strengths are different, then the highest strength signal will dominate. So, just to know which one is higher which one is lower? So, this table just shows you that strength increases in this reaction. So, if two signals of unequal strengths are driven on a single wire, this stronger signal will prevail, right.

So, with this we come to the end of this lecture. So, we shall be continuing with our discussion in the next lecture as well.

Thank you.

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

Lecture - 07
Verilog Language Features (Part 2)

So, in the last lecture, if you recall, we were talking about some of the features of the Verilog language in particular, we are discussing the net type variable. So, what are the different types of net type variables; now we continue our discussion here in this lecture.

(Refer Slide Time: 00:40)

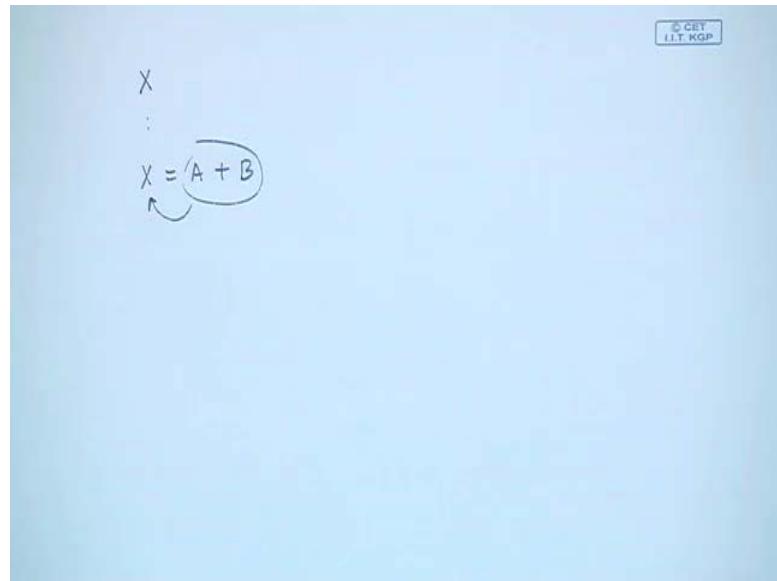
(b) Register Data Type

- In Verilog, a “*register*” is a variable that can *hold* a value.
 - Unlike a “*net*” that is continuously driven and cannot hold any value.
 - Does not necessarily mean that it will map to a hardware register during synthesis.
 - Combinational circuit specifications can also use register type variables.
- Register data types supported by Verilog:
 - i. `reg` : Most widely used
 - ii. `integer` : Used for loop counting (typical use)
 - iii. `real` : Used to store floating-point numbers
 - iv. `time` : Keeps track of simulation time (not used in synthesis)

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

So, here we shall be first talking about the other data type that we talked about in addition to net, the register data type. Now register data type by its very semantic, it means that it is a variable which can hold a value, like suppose I have defined a register X, then I can write a statement like X equal to a something, let X equal to A plus B.

(Refer Slide Time: 01:13)



So, I can write something; something will get assigned to X, ok. Now, this assignment will not be done using an assign statement; not continuous assignment because I told in the last lecture that in an assign statement the left hand side must be a net type variable, it cannot be a register type variable, ok.

So, whenever you use a register type variable, there you are actually storing a value, it is supposed to hold the value till you are using it again, ok. So, a register is a variable that can hold a value and this is different from a net which is continuously driven and cannot hold any value because if the line that is driving the net changes. The net variable immediately changes, it cannot hold it, but I also mentioned that this does not mean that a register variable will always map to a hardware register. This may or may not be true, there will be some cases which we shall be illustrating with examples where you may be using a register variable in a Verilog code, but ultimately your circuit that we synthesized will be a combinational circuit, right.

So, the following register data types are supported in Verilog, the most widely used is something called reg, this is short form for register, then you have integer, real, and time. Integer is typically used for application where you want to count something, like number of times you are looping and so on and real as the name implies, this is used to store some floating point number with fractional parts and time is a special variable which is used to keep track of simulation time.

Now, you say let me just emphasize one thing here that we shall be talking about many features of Verilog, which are simulation only features that make sense only when you are doing simulation. When you are carrying out synthesis, those features does not mean anything and the synthesis tool will be simply ignoring them.

Let me take some examples, see whenever we instantiate a gate, we had seen some examples where we had specified some delays of these gates, right, like you can use using that hash (#) symbol, I can right hash5, hash2, hash1, but the synthesis tool will be ignoring that those will be only for simulation purposes because in during synthesis whenever an AND gate is actually implemented. So, whatever is the actual delay of the AND gate that will be the delay, right, not 1 or 2 or 3 whatever you are saying. Similarly, this time, this time is something that relates to the time of simulation, this is never used during synthesis, such variables will be ignored during synthesis, ok.

(Refer Slide Time: 04:41)

• “*reg*” data type:

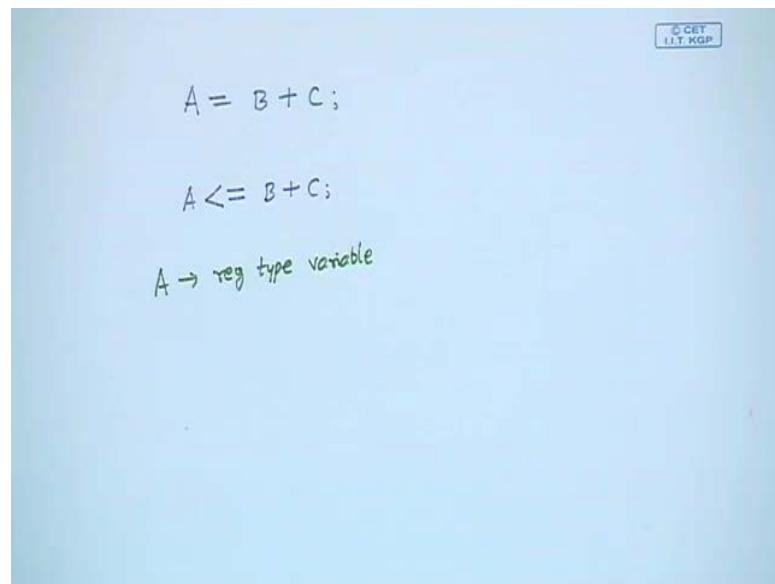
- Default value of a “*reg*” data type is “x”.
- It can be assigned a value in synchronism with a clock or even otherwise.
- The declaration explicitly specifies the size (default is 1-bit):

```
reg x, y;           // Single-bit register variables
reg [15:0] bus;    // A 16-bit bus
```
- Treated as an unsigned number in arithmetic expressions.
- Must be used when we model actual sequential hardware elements like counters, shift registers, etc.

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

Let us start with reg, the first thing is that in the reg data type. So, just after you declare, but you have not initialized it to any value. So, the default value will start with undefined or x. A reg value can be assigned in several different ways, we shall be seeing that they can be assigned in synchronism with a clock or even without a clock, you can do it.

(Refer Slide Time: 05:14)



Now, there are two kinds of assignments, we should be learning like for example, a reg type variable A, you can either write an assignment like this or you can write an assignment like this, less than equal to. So, their meanings are different, we shall be explaining these things later, ok, but the point to notice that here A is a reg type variable. So, for these kind of assignments the left hand side must be a reg type variable, right.

So, some of the example declarations are shown here reg x, y. Well if you simply specify the name of some variable like here x and y, they will be by default taken to be 1-bit variables, single bit variables. So, x is a 1-bit variable, y is a 1-bit variable. Well you can also define a vector, we shall be talking about vector separately again, this is one example reg within square bracket 15 colon 0 bus (reg [15:0] bus), this means I am declaring a variable bus where there are 16 bits, 0 is the least significant bit, 15 is the most significant bit. So, I can define a 16-bit bus variable called bus using a statement like this. Now a reg type variable when it is used in an arithmetic expression, it is treated as an unsigned number without any sign and another point is that when you are actually implementing some hardware register base design like counter, shift register, etc. then you must use reg, there is no other alternative.

(Refer Slide Time: 07:14)

The screenshot shows a slide from a presentation. On the left, there is a code block containing Verilog code for a simple counter. On the right, there is a description of the counter and a list of bullet points. At the bottom, there are logos for IIT Kharagpur and NPTEL, and the title "Hardware Modeling Using Verilog" next to a video thumbnail of a speaker.

```
module simple_counter (clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @ (posedge clk)
    begin
        if (rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```

32-bit counter with synchronous reset.

- Count value increases at the positive edge of the clock.
- If “rst” is high, the counter is reset at the positive edge of the next clock.

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

So, here there is one example which uses reg, of course, here there are many constructs which I have not discussed yet, but this example you can appreciate what it is, here we are trying to implement a counter, there are 3 ports: clock, reset and count. Clock and reset are the input and count is the output, as you can see, count is a 32-bit vector 31 to 0 (31:0); that means, this is a 32-bit counter. Well, ignore this reg for the time being and this is the main body of the counter description, there is a statement called always, always at positive edge of the clock (@posedge clk). So, in English, it means this, always whenever there is a positive edge of the clock coming, you do this, what you do? you check if, if the reset signal is one on high or not, if it is high, you initialize count value 32'b0 means it is a 32-bit number, value is 0. So, you are initializing it to 0.

If reset is one count is 0, else count equal to count plus 1; you do this, every time clock edge, positive edge is coming. Now as I said such expression with equal to, so, these are examples, where you are using that, but you can have only a reg type value in the left hand side that is why, I have to separately declare the same variable count, which was in the output, also as a reg because if you do not do this, your compiler will give you an error that left hand side variable type is wrong, right. So, here the count value is increasing at the positive edge of the clock and if reset is high the counter will be reset whenever next clock comes, it will reset to 0.

(Refer Slide Time: 09:26)

The slide shows a Verilog module definition:

```
module simple_counter (clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @ (posedge clk or posedge rst)
    begin
        if (rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```

Next to the code, there is a description of the module:

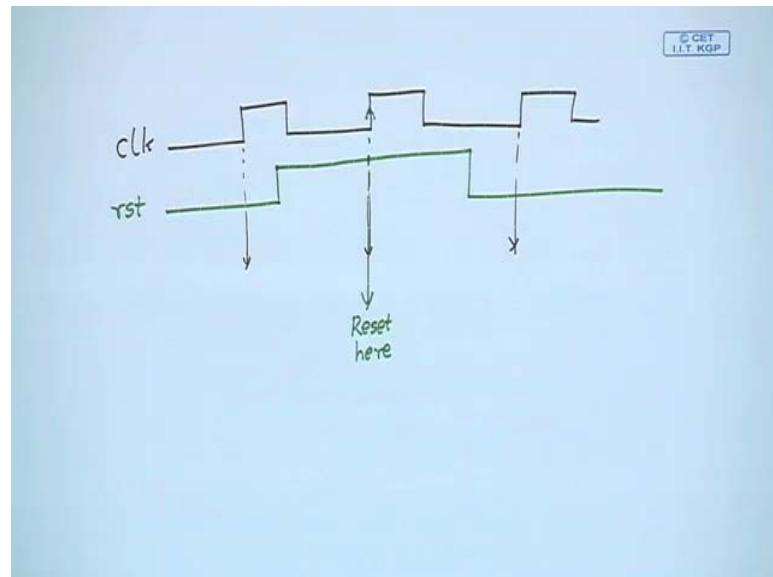
32-bit counter with asynchronous reset.

- Here reset occurs whenever "rst" goes high.
- Does not synchronize with clock.

At the bottom, there are logos for IIT Kharagpur and NPTEL, and the text "NPTEL ONLINE CERTIFICATION COURSES" and "Hardware Modeling Using Verilog". There is also a portrait of a man in a blue shirt.

Now, there is another way of specifying this, look at this previous one here, this always block was executing begin end whenever the positive edge of clock was coming.

(Refer Slide Time: 09:41)



So, you think of a scenario like this. So, I have a clock, clock is coming like this, this is one positive edge, this is another positive edge, this is another positive edge, positive edge means 0 to 1; 0 to 1 positive edge. So, this always block will get executed here, here and here, but suppose my reset signal, I have made it high here. Let us say I have made it high here and have kept it high for some time and then I made it low. See, here even

though I have made the reset signal high here, but according to our previous design the counter will be reset only here whenever the next positive edge of the clock comes.

So, counter will be reset here because the always block will be executing only when posedge is coming. So, even if your reset has been activated here, it will be waiting for some time before the counter is actually reset. But there can be application where you may want that whenever reset is going high, you reset the counter immediately, right. So, our next design which is a slight modification of this one, what we do? we change this always statement, we add another clause.

What I say; we say always at positive edge of the clock or positive edge of reset. So, if the reset signal goes from 0 to 1 that will also cause the begin end to execute and it will see that reset has become one, it will immediately reset, it to 0. So, this is an example of asynchronous reset, which means resetting is not happening in synchronism with the clock, ok.

(Refer Slide Time: 11:46)

- “*integer*” data type:
 - It is a general-purpose register data type used for manipulating quantities.
 - More convenient to use in situations like loop counting than “*reg*”.
 - It is treated as a 2’s complement signed integer in arithmetic expressions.
 - Default size is 32 bits; however, the synthesis tool tries to determine the size using data flow analysis.
 - Example:

```
wire [15:0] X, Y;
integer C;
Z = X + Y;
```
 - Size of Z can be deduced to be 17 (16 bits plus a carry).

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

So, you can have both these descriptions in Verilog as you want. Next comes the integer data type, the integer is a more general purpose type, which, where you can use it to store any arbitrarily integer values mainly for counting some situations like loop counting. So, reg may not be very convenient there because whenever you are counting something, the count value may sometimes become negative, but as I said in reg, a variable declared of

type reg is always considered to be an unsigned quantity, it cannot have a negative value, ok, fine.

So, integer data type another feature is that it cannot have a size defined to it. So, the default size is 32-bits, but the point to note is that, but if you are doing synthesis again, not for simulation. If you are doing synthesis, the synthesis tool will try to apply some intelligence. It will do some kind of data flow analysis and try to guess that what should be the value of the integer should I keep whole 32-bits or less than 32-bits should also be ok. Like you take an example, suppose, I have declared 2 wires x and y vectors of size 16 each 0 to 15; 16 bit, 16 bits and I declare an integer of size C.

So, as I said that whenever you declare an integer, you cannot specify the size, then we write. Sorry this not Z this is C, let me correct this, this is C. So, here if I write C equal to x plus y, this is also C, yeah. So, if I write C is equal to x plus y, what will happen? Two 16-bit numbers are added. So, the synthesis tool will see that x and y are 16-bit numbers you are adding it. So, this sum can be maximum 17-bits, 16 bits plus a carry.

So, when it realizes the hardware for C, it will not be implementing it as a 32-bit counter rather, it will be using it only as a 17-bit register, 17-bit is sufficient in this particular case, ok. This is what I mean is that this synthesis tool will try to determine the size by carrying out some data flow analysis wherever possible, ok.

(Refer Slide Time: 14:32)

The slide has a yellow header bar with a blue gradient at the bottom. The main content area is white with a black border. At the bottom, there is a footer bar with the IIT Kharagpur logo, NPTEL logo, and course information.

- “*real*” data type:
 - Used to store floating-point numbers.
 - When a real value is assigned to an integer, the real number is rounded off to the nearest integer.
 - Example:

```
real e, pi;
initial begin
    e = 2.718;
    pi = 314.159e-2;
end
integer x;
initial x = pi; // Gets value 3
```

Talking about real data types as the name implies, they are used to store floating point numbers; numbers with fractional parts and just like in a high level language. So, whenever you assign a real value to an integer, there is little difference. Normally in a high level language like C the number is truncated and the integer part is stored, but here it is rounded off. So, a small example, suppose I define 2 variables e and pi as real and in the initial block which you have used seen in the test benches, this initial is normally used in the test benches.

So, in an initial block let us say I have initialized e to 2.718 and pi to, this is just a notation. So, I could have written 3.14159, but I have written 314.159 into 10 to the power minus 2 (314.159e-2), e minus 2 (e-2), e minus 2 means 10 to the power minus 2, right. Now, suppose I declare an integer x and somewhere again inside an initial block, I write x equal to pi. So, pi was 3.14, say to be rounded up and only the value 3 will be assigned to X, right; this is what I mean here, ok.

(Refer Slide Time: 16:08)

- “*time*” data type:
 - In Verilog, simulation is carried out with respect to a logical clock called simulation time.
 - The “*time*” data type can be used to store simulation time.
 - The system function “\$*time*” gives the current simulation time.
 - Example:

```
time curr_time;
initial
...
curr_time = $time;
```

Now time, time is a data type, which is used to keep track of simulation time. So, when simulation is carried out, well you can use that Iverilog that I had mentioned, even you can say means, you can use Iverilog or any other tool which are available to simulate a Verilog design. So, when you do simulation there is a notion of simulation time, you specify time, you specify gate delays, lot of things you do. So, just a small example, in the Verilog code,

you can define a variable, let us say current time is a variable of type time and this special system function dollar time is there.

So, anything starting with dollar is a system function, this system function actually gives you the current simulation time. So, in the initial block somewhere, if you want to assign the current time to this variable, you can write a statement like this curr_time equal to dollar time. So, you can use the time data type in some applications where you actually want to find out that to execute a block how much time it has taken, it took. So, you can print the time before you can print the time after and see the difference, ok, like that, fine.

(Refer Slide Time: 17:43)

Vectors

- Nets or "reg" type variable can be declared as vectors, of multiple bit widths.
 - If bit width is not specified, default size is 1-bit.
- Vectors are declared by specifying a range [range1:range2], where *range1* is always the most significant bit and *range2* is the least significant bit.
- Examples:

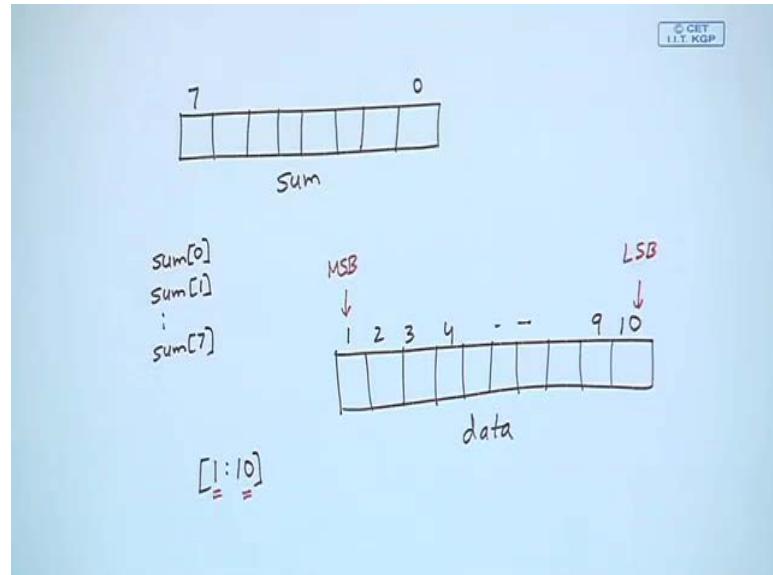
```
wire x, y, z;           // Single bit variables
wire [7:0] sum;          // MSB is sum[7], LSB is sum[0]
reg [31:0] MDR;
reg [1:10] data;         // MSB is data[1], LSB is data[10]
reg clock;
```

Now, let us come to vectors, you have seen single bit variables. Now let us see how can we group such single bit variables to create something called vectors. So, nets or reg types variables both of them, they can be declared as vectors. Vectors are nothing, but multiple bit quantities and if you do not specify this bit width then by default it is taken to be one bit. So, some examples we have already seen, vectors can be declared by specifying a range within square bracket range1 colon range 2 ([range1: range2]), the convention is that the first one, range1 is always the most significant bit of the number and the rightmost one range2 is the least significant bit.

Well, I will explain what it means; let us see some examples, suppose I write simply wire x, y, z; x, y, z are simple variables without any vector notation, these are single bit

variables. So, if I write wire 7 colon 0 sum (wire [7:0] sum); what does this mean? This will mean that I have an 8-bit variable called sum, this is my sum, there are 8-bits.

(Refer Slide Time: 19:13)



The index is 7 is the most significant, 0 is the least significant, the individual bits I can access by writing $\text{sum}[0]$, $\text{sum}[1]$ up to $\text{sum}[7]$, this is just like an array, this just like an array, array of bits, ok. Similarly, another example MDR, this of course I have defined of type reg. Well, now the point is a range1, range 2 can be any arbitrary values. So, it is not necessary that range1 must be greater than range2, like in this example I have shown reg 1 colon 10 data (reg [1:10] data). So, how many numbers are there? 1 to 10, there are 10 numbers. So, actually here also we are declaring a register of size ten, but how? this is data, let us see. So, the name of the variable is data and the way we have declared, the range1 was 1, range 2 was 10. So, the index values are defined like this 1, 2, 3, 4 up to 10. So, in this case.

So, if you use this data in an arithmetic expression, this particular bit will be treated as the most significant bit, this particular bit will be treated as the least significant bit. So, by specifying your range1 and range 2 in a suitable way, you can choose to swap LSB & MSB, like here in this case, the 7 was the MSB and 0 was the LSB, but if you write 1 to 10, 1 become MSB, 10 becomes LSB. So, this convention has to be kept in mind, the first value of the range will be the bit on the left, the second value on the range will be the index, bit index of the further on the right.

(Refer Slide Time: 21:47)

- Parts of a vector can be addressed and used in an expression.
- Example:
 - A 32-bit instruction register, that contains a 6-bit opcode, three register operands of 5 bits each, and an 11-bit offset.

```
reg [31:0] IR;           opcode = IR[31:26];
reg [5:0] opcode;        reg1 = IR[25:21];
reg [4:0] reg1, reg2, reg3; reg2 = IR[20:16];
reg [10:0] offset;       reg3 = IR[15:11];
                        offset = IR[10:0];
```

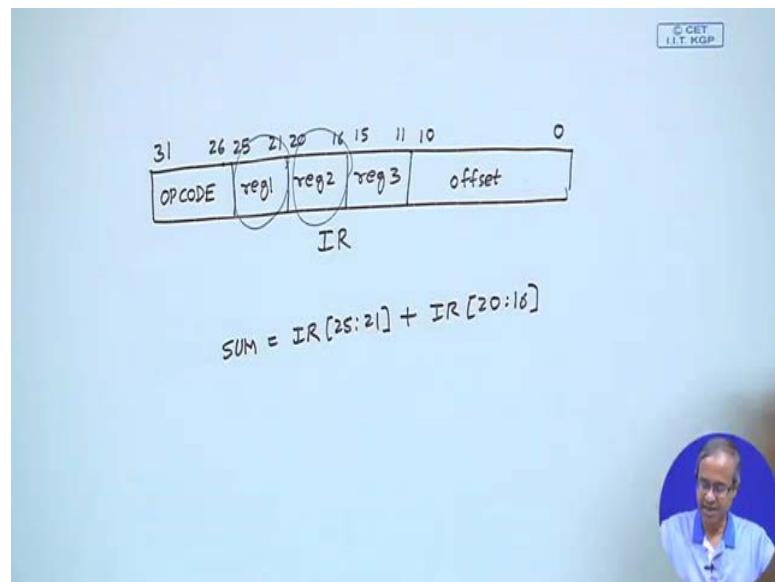
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

So, that will be your word. So, whenever you are using it, you should keep that in mind, ok, all right.

So, after you have seen this vector, let us see how we can use sections of a vector in an expression. So, what do you mean by sections of a vector? suppose, I have declared a vector of size something, let us say 32. So, the example I have shown is the vector of size 32 and I want to extract 5-bits from the middle of the vector and assign it to some variable. Now, that can I do it? Yes, I can do it. So, I can take out any cross section of a vector by specifying the starting and the ending index and I can assign it to some variable, I can use it in an arithmetic expression whatever I can do I want, let me take an example. So, here I am taking an example of an instruction encoding. So, the example I have taken is a 32-bit instruction register that contains a 6-bit opcode, 3 register operands 5-bits each and an 11-bit offset.

So, what I mean is something like this suppose, I have a 32-bit instruction register.

(Refer Slide Time: 23:09)



So, bits, let us say from 0 to 31. So, what are the different fields? So, I have the first 6 fields are the opcode; first 6 fields means bit number 31 to bit number 26, then I have 3 register operands. So, I call them reg1, reg2, reg3, these are all 5-bit fields. So, the bit number will be 21 to 25, 16 to 20 and 11 to 15 and the last field, is a field which is called offset. Offset is an 11-bit field: 0 to 10. Suppose my instruction register has this format, there are 5 fields, right. So, what I can do is something like this, I can declare this register IR 32-bit and I can declare the individual fields also, opcode is a 6-bit field; reg1, reg2, reg3 are 5-bit fields: 4 to 0, offset is an 11-bit field, then I can simply write opcode equal to IR 31 colon 26 (opcode = IR [31:26]). So, you see here. So, we mentioned 31 to 26 opcode will be assigned, 31 colon 26.

Similarly, reg1 will be 25 to 21, you see reg1 will be IR 25 to 21 (reg1 = IR [25:21]), reg2 will be 20 to 16 (reg2 = IR [20:16]), reg3 15 to 11 (reg3 = IR [15:11]), offset 10 to 0 (offset = IR [10:0]). So, in this way, I can use this cross sections in an assignment statement even mean, I can use them in an expression, like let us say, I can write an expression like this also, next I can write sum equal to let us say IR 25 to 21 plus IR 20 to 16 (sum = IR[25:21] + IR[20:16]). So, I can write like this also. So, here what I am meaning is that 25 to 21 means this number (reg1), 20 to 16 means this number (reg2); I am adding these two 5-bit numbers and I am storing the result in another variable sum, such things I can also do, right. So, I can take out a cross section from a given vector and I can do any kind of manipulations on it, ok.

(Refer Slide Time: 26:00)

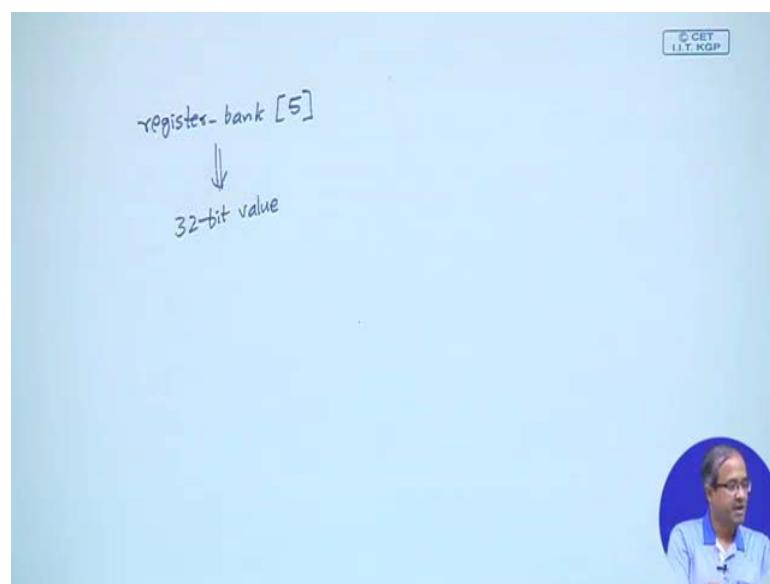
Multi-dimensional Arrays and Memories

- Multi-dimensional arrays of any dimension can be declared in Verilog.
- Example:
`reg [31:0] register_bank[15:0]; // 16 32-bit registers
integer matrix[7:0][15:0];`
- Memories can be modeled in Verilog as a 1-D array of registers.
 - Each element of the array is addressed by a single array index.
 - Examples:
`reg mem_bit[0:2047]; // 2K 1-bit words
reg [15:0] mem_word[0:1023]; // 1K 16-bit words`

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

Now, talking about multi-dimensional arrays and memories, now we can declare multi-dimensional arrays in Verilog of any dimension. Let us take two examples here reg 31 to 0 register bank again 15 to 0 (reg[31:0] register_bank[15:0]), what does this mean? First is reg 31 to 0, this is the declaration of a register of width 32, 32-bit register. Now we are saying that I have a register bank where there are 16 elements. So, we are actually talking about sixteen 32 bit registers, right. So, and when you access this element will be accessing them using the two index values, like here if I just write.

(Refer Slide Time: 27:04)



Let us say, register_bank, if I just write 5, register_bank 5 (register_bank[5]), this will mean, a 32 bit value, the content of the register number 5 because here I have declared register_bank as an array of size 16; 0 to 15. So, when I say register_bank 5. So, it is one of those elements and each of this element is of type reg 31 to 0; that means, a 32-bit number.

But, but here for integer for example, integer is already a 32-bit by default. So, I need not have to specify this. So, I write matrix [7:0][15:0], this is a 2 dimensional matrix with 8 rows and 16 columns. Now will see later that when we use, let us say memories in a design. So, one way there are, other ways also we have seen. So, one way is to declare a memory as a two dimensional array, multi-dimensional array. So, some examples of memory declaration is this. So, you can simply write reg some name mem_bit 0 to 2047 (reg mem_bit[0:2047]), this is a 2 kilo bit memory, total number of location is 2048 and each location contains one bit, but if you declare it like this, similar to this register_bank (reg[15:0] mem_word[0:1023]). So, there are 0 to 1023; one kilo words, each word is of size 16. So, you can declare a memory like this.

(Refer Slide Time: 29:00)

Specifying Constant Values

- A constant value may be specified in either the *sized* or the *unsized* form.
 - Syntax of sized form:
`<size>'<base><number>`
 - Variables of type integer and real are typically expressed in unsized form.
 - Examples:

<code>4'b0101</code>	// 4-bit binary number 0101
<code>1'b0</code>	// Logic 0 (1-bit)
<code>12'hB3C</code>	// 12-bit number 1011 0011 1100
<code>12'h8xF</code>	// 12-bit number 1000 xxxx 1111
<code>25</code>	// signed number, in 32 bits (size not specified)

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

So, specifying constant values, you can specify constant values like this in either sized or un-sized form, the syntax is, you specify a size then a single quote followed by a base then a number, here I have shown some examples of the base I have shown as binary or

hexadecimal h. 4'b0101 means a 4-bit binary number 0 1 0 1. 1'b0 means logic 0, 1 bit. 12 hexadecimal means, a 12-bit hexadecimal number, 12-bit B 3 C (12'hB3C).

So, this is B, 3, C. Well out of this, I can have some x also, don't care also, 8xf, this is 8, x means all xxxx, f (1000 xxxx 1111) and if I write only 25, this is a sign number and by default, it will be put in 32-bits. So, whenever you are using integer or real numbers you typically do not use the size specifier; they are typically expressed in un-sized form, right.

(Refer Slide Time: 30:19)

Parameters

- A parameter is a constant with a given name.
 - We cannot specify the size of a parameter.
 - The size gets decided from the constant value itself; if size is not specified, it is taken to be 32 bits.
- Examples:

```
parameter HI = 25, LO = 5;
parameter up = 2b'00, down = 2b'01, steady = 2b'10;
parameter RED = 3b'100, YELLOW = 3b'010, GREEN = 3b'001;
```

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

Now, talking about parameters, we shall be seeing parameters again later. So, parameter is something similar to hash define that you use in a C program, it is like a constant, we define a constant with a given name and in the program whenever we use that name, we replace that name by that constant. So, this parameter is something like that.

But when you declare parameter, we do not specify the size, the size automatically get deduced by the constant values that we are defining like, we have given some examples, it will be clear. The first example, I am saying parameter HI equal to 25 (HI = 25), LO equal to 5 (LO = 5). So, in the program whenever there is an HI, this HI will be replaced by 25 and whenever there is LO, LO will be replaced by 5, similarly next example up, down, steady.

So, I am declaring 2-bit number 00 means up (up = 2b'00), 01 is down (down = 2b'01), 10 is steady (steady = 2b'10). So, if we use parameter in the program it will make your

program more readable. So, instead of writing 00 and 01; if you write up and down, it will be easier to understand, right. So, that is the main purpose of using this kind of constructs parameter. Similarly, for a traffic light controller system, let us say you can define RED as a 3-bit number 100 (RED = 3b'100), YELLOW as 010 (YELLOW = 3b'010), GREEN as 001 (GREEN = 3b'001).

(Refer Slide Time: 31:57)

```
// Parameterized design:: an N-bit counter
module counter (clear, clock, count);
    parameter N = 7;
    input clear, clock;
    output [0:N] count; reg [0:N] count;

    always @ (negedge clock)
        if (clear)
            count <= 0;
        else
            count <= count + 1;
endmodule
```

Any variable assigned within the "always" block must be of type "reg".

Ok, this are some examples. And just one Verilog complete module, I am showing that uses a parameter, this is an N-bit counter.

So, I am declaring it as an N, ok, clear, clock, count. clear and clock are the inputs, count is the output. This is 0 to N and because I am using it on the left hand side, I am also declaring it as reg. So, this counter counts at a negative edge of the clock. So, it is negedge. So, very similar to the previous one; if clear, count, you see here, we use the other kind of assignment less than equal to (\leq). So, the difference will explain later, ok, 0, else count equal to count plus 1. So, here if you just change this one line in the program, your module can change from a means, from a 7-bit counter to something else not 7-bit, 8-bit counter, N plus one bit, 0 to N, parameter N equals 7 means it is an 8-bit counter. So, if I make it 15, it will become a 16-bit counter, ok.

So, you change just one line, ok, that is the advantage of using parameters. So, with this we come to the end of this lecture again we shall be looking at some more features of the Verilog language in our next lectures.

Thank you.

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

Lecture - 08
Verilog Language Features (Part 3)

So, you know earlier lectures, we have seen some examples where you have tried to illustrate some structural design constructs by using instantiation of gates. So, in the present lecture, we shall first see what are the various types of gates that are available as part of the Verilog language which you can directly use and instantiate in our design.

(Refer Slide Time: 00:50)



(Refer Slide Time: 00:53)

Predefined Logic Gates in Verilog

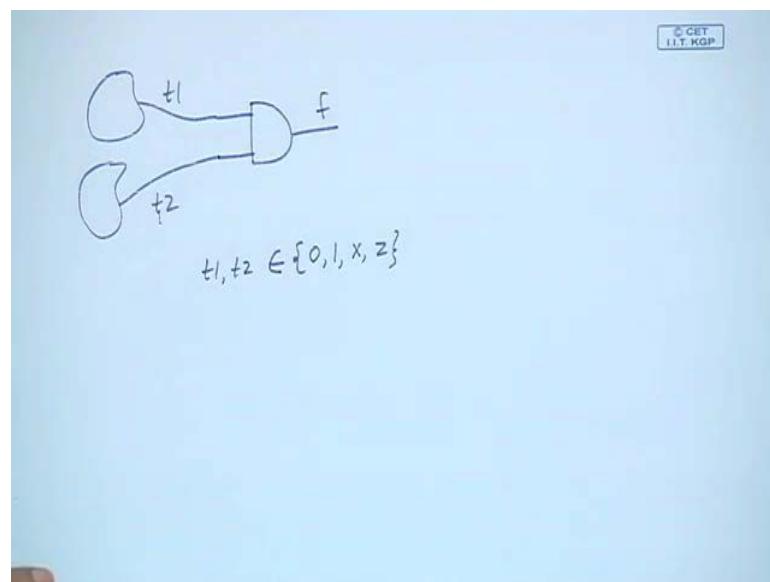
- Verilog provides a set of predefined logic gates.
 - Can be instantiated within a module to create a structured design.
 - The gates respond to logic values (0, 1, x or z) in a logical way.

2-input AND	2-input OR	2-input EXOR
$0 \& 0 = 0$	$0 0 = 0$	$0 ^ 0 = 0$
$0 \& 1 = 0$	$0 1 = 1$	$0 ^ 1 = 1$
$1 \& 1 = 1$	$1 1 = 1$	$1 ^ 1 = 0$
$1 \& x = x$	$1 x = 1$	$1 ^ x = x$
$0 \& x = 0$	$0 x = x$	$0 ^ x = x$
$1 \& z = x$	$1 z = x$	$1 ^ z = x$
$z \& x = x$	$z x = x$	$z ^ x = x$

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

So, this is the title of our lecture and we start with our discussion on predefined logic gates in Verilog. So, we have already seen some of these gates AND, OR, NOT, NAND, NOR, EXOR. Verilog provides predefined logic gates as I said and we have also talked about earlier that in Verilog will support 4 logic values state; 0, 1, undefined (x) and high impedance (z).

(Refer Slide Time: 01:34)



Now when you talk about gates for example, I use a AND gate. So, the inputs of the AND gates are coming from say, two other hardware blocks. Let us call this t1, let us call this t2. Now this t1 and t2, they will be belonging to this set 0, 1, x or z ($t1, t2 \in \{0,1, x,z\}$).

So, now when I say that this is an AND gate, I also have to define; what is the behavior of this AND gate. If some arbitrary input combinations are coming on t1 and t2; what should be the value of f. Ok, this actually follows from intuition; let us see. Let us look at an AND gate from the definition of an AND gate, 0 and 0 is 0, well if one of the input is 0, other is 1, it is also 0, when both the inputs are 1, then it is 1.

Now let us look at the behavior with x, one of the input is x and the other input is 0 or 1. So, if one input is x and the other is 1 for AND operation you cannot say what is the output, but if the one input is 0, then you can definitely say that the output will be 0. Similarly, for the high impedance state, some other things see if one of them is 1, other is the high impedance state, then the output can be an indeterminate level, I call it x and if one is z, other is x, output is also x.

So, for OR operation, it is again similar, 0 0 is 0, 0 1 is 1, 1 1 is 1 and for don't care if one of the input is 1, then the output is definitely 1, but for 0, you cannot say, for this case is again x. For XOR similarly, 0 0 is 0, 0 1 is 1, 1 1 is 0 and for other cases you really cannot say for XOR function, if one of the input is x or z, the output will be undefined. So, for all the gates, you can define the behavior in terms of the four valued logic 0, 1, x, z like this.

(Refer Slide Time: 03:54)

The slide has a yellow header bar with the title "List of Primitive Gates". Below the header, there is a table of Verilog code for primitive gates:

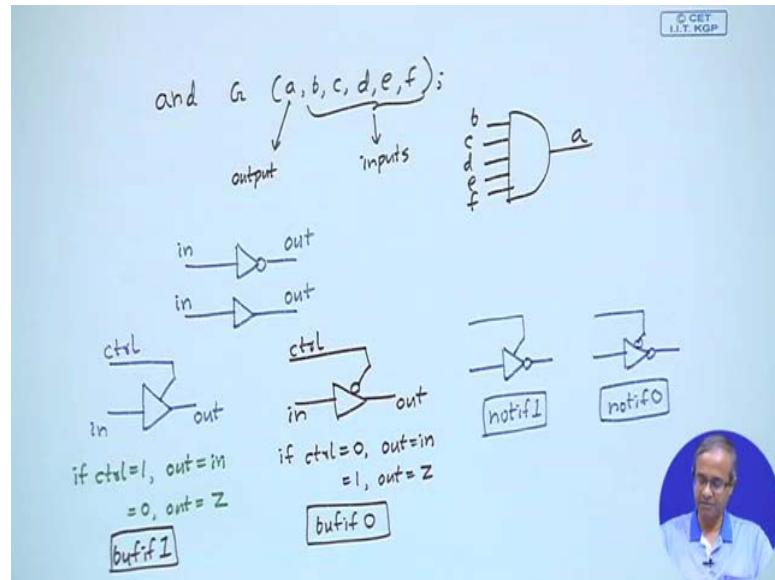
and G (out, in1, in2);	bufif1 G (out, in, ctrl);
nand G (out, in1, in2);	bufif0 G (out, in, ctrl);
or G (out, in1, in2);	notif0 G (out, in, ctrl);
nor G (out, in1, in2);	notif1 G (out, in, ctrl);
xor G (out, in1, in2);	
xnor G (out, in1, in2);	
not G (out, in);	
buf G (out, in);	

In the bottom right corner of the slide content area, there is a note: "There are gates with tristate controls".

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the text "Hardware Modeling Using Verilog".

Now, talking about the list of gates this is the complete list, the basic gates are this AND, NAND, OR, NOR, XOR, XNOR, NOT. There are buffers and there are some couple of more NOT gates. So, I will explain this, for the normal AND, NAND, OR, NOR, XOR, or XNOR, the number of inputs can be arbitrary. Like for example, for AND I can write.

(Refer Slide Time: 04:35)



An instantiation and gate is G, so, I can write like this a, b, c, d, e, f (and G (a,b,c,d,e,f));, this will mean that the first variable that I am using here, this will be my gate output and all the remaining variable, there are 5 here, this will be my inputs. So, actually I am defining a 5 input AND gate, well a is the output and b, c, d, e and f are the inputs. So, this way depends on how many variables I am using in the parameter. So, the number of inputs of the gate will be defined accordingly, right.

So, there is a flexibility. NOT is simple, NOT will have a single input and a single output. So, the input is in output is out. Come to buf; buf is just a buffer sometimes you need to means isolate signals, this out equal to in, the logic value does not change, but the signal values are restored, this is a buffer. Now there are some tri-state version of buffers and NOT, bufif1. Let us say, let us see this first bufif1 is something like this, it is a buffer which is selected by a control signal like this. So, this is ctrl, this is in, this is out. So, the behavior is, if your control is one then out will be equal to in, if control is 0 then output will be tri-state, it will be in the high impedance state, right, this is your bufif1, this gate.

Similarly, you have another gate called bufif0; bufif0 is similar, the only difference is that the polarity of the control is different, there is a negation here. So, it is just the reverse, if control equal to 0 then out equal to in, if control equal to 1 out equal to high impedance state, this is bufif0. Similarly, there are equivalent versions for NOT's, NOT's are very similar, I am just showing the diagram and not writing expression instead of the buffer this will be the NOT. So, the output will be in bar and there will be a control, this is notif1 and there is another version, NOT with the control inverse, this is notif0. So, this kind of tri-state control gates are also available, if you want to use it in a design, right.

(Refer Slide Time: 08:31)

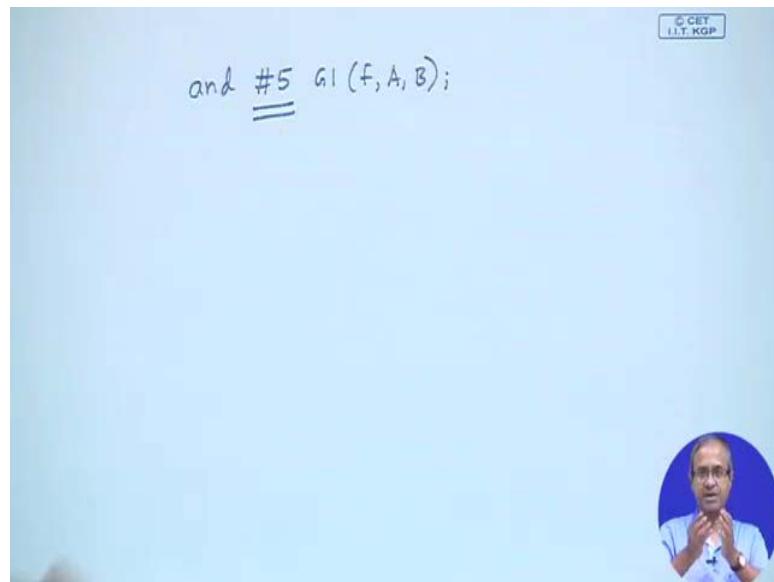
- Some restriction when instantiating primitive gates:
 - The output port must be connected to a net (e.g. a wire).
 - An “output” signal is a wire by default, unless explicitly declared as a register.
 - The input ports may be connected to nets or register type variables.
 - They have a single output but can have any number of inputs (except NOT and BUF).
 - When instantiating a gate, an optional delay may be specified.
 - Used for simulation.
 - Logic synthesis tools ignore the time delays.

So, when you are instantiating these kind of primary gates or primitive gates, there are some restrictions you need to remember. Like for example, here the outputs, the first argument or the parameter is the output, the output port must always be connected to a net, it cannot be connected to a register, ok.

So, an output signal in a module is a wire by default. So, you can directly use it if you want. Now the input ports of the gate, like for example, this n1 and n2, these can be anything, they can be connected to either nets or register type variables. This I have already said, that they have a single output, but can have any number of inputs, the basic gates except NOT and buffer. Well and I have seen in the examples earlier that when you instantiate a gate; you can specify an optional delay, like suppose you specify an AND, you can write AND hash 5 G1 f A B (and #5 G1 (f,A,B);), like this.

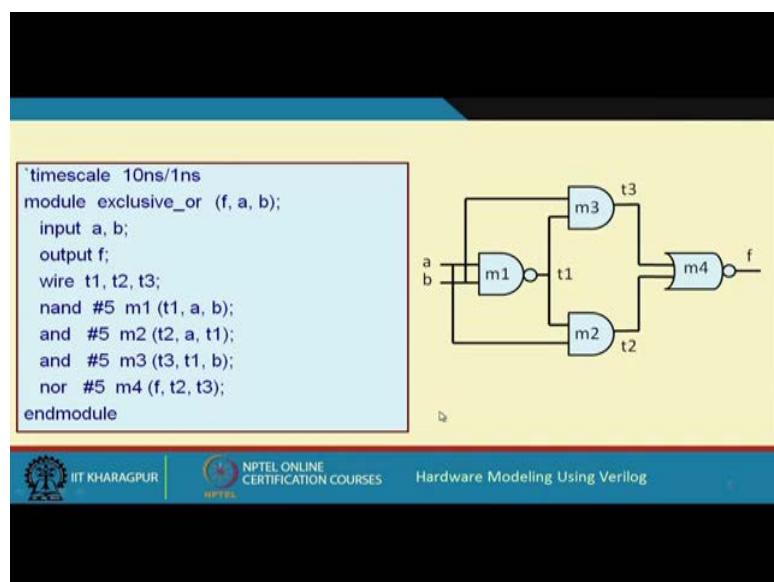
Now this delay is again only for simulation purpose, for synthesis purpose this delay does not make any sense, right because it is ultimately the hardware AND gate, how fast or how slow it is that will determine the delay. So, as a designer I have no idea what the delay will be, how many picoseconds or how many nanoseconds, right, ok.

(Refer Slide Time: 09:45)



So, this optional delay, this is used only for simulation and the logic synthesis tool will ignore these delays.

(Refer Slide Time: 10:29)



So, let us take an example, which shows these delays. So, here we have a structural representation, well ignore the first line for the time being, I will explain this what does this mean. This is an exclusive OR (EXOR) function realization using NAND gates; NAND, AND, NOR gates. So, using 4 gates, I can implement this exclusive here. There are 3 wires t1, t2 and t3. So, we have used this, this M1 is an NAND, M2 is an AND, M3 AND, M4 is an NOR. Here we have all specified delays as 5. As I said, this will be used only for simulation purposes, ok. Now the first, line time scale.

(Refer Slide Time: 11:26)

The `timescale Directive

- Often in a single simulation, delay values in one module need to be specified in terms of some time unit, while those in some other module need to be specified in terms of some other time unit.
- The `timescale compiler directive can be used:
`timescale <reference_time_unit>/<time_precision>
- The <reference_time_unit> specifies the unit of measurement for time.
- The <time_precision> specifies the precision to which the delays are rounded off during simulation.
 - Valid values for specifying time unit and time precision are 1, 10 and 100.

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

Time scale actually tells you what these numbers 5 mean, you see, roughly speaking the first number here ten nanoseconds (10ns), this gives you the basic unit of time. So, when I write 5, it will actually mean 50 nanoseconds and 1 nanosecond will be the precision of simulation. So, these are explained here in some detail. So, when you use the time scale directive, you can specify these two times. Now why you need this? you need this for simulation purposes, that is the first thing. Now with respect to some module, you may have to specify some delay values, but some, but for some other module the delay value may be different.

So, it is always good to have this kind of a declaration, time scale declaration at the beginning of every module. So, if the time scale values are different from one module to the other, you can have different declarations like that. So, the syntax of the time scale command or directive is this, reverse quote time scale ('timescale), you specify a reference

time unit (<reference_time_unit>) slash (/) time precision (<time_precision>), the reference time unit is actually used to specify the unit of measurement of time as I had said. So, whatever here you specify 5, this 10 nanosecond is the unit 5 in to 10, right.

And time precision, the second one here, in case we mentioned 1 nanosecond, it specifies the precision with which we round off the delays during simulation; that means, how accurate the simulation is. This simulation will be accurate up to 1 nanosecond, ok, that is why we have given here 1 nanosecond. And for this times, the valid values you can give only 1,100 and 10, 10 and 100 nothing, you cannot write 2, 3, 5 here, this values can be 1, 10 and 100 only and this unit can be of course, different.

(Refer Slide Time: 13:51)

The screenshot shows a presentation slide with a yellow background. At the top, there is a black bar with a blue triangle pointing right. Below the bar, the slide title is 'Example:'. Underneath the title, there is a code snippet: 'timescale 10ns/1ns'. To the right of the code, there is a bulleted list of four items:

- Reference time unit is 10ns, and simulation precision is 1ns.
- If we specify #5 as delay, it will mean 50ns.
- The time units can be specified in s (second), ms (millisecond), us (microsecond), ps (picosecond), and fs (femtosecond).

At the bottom of the slide, there is a footer bar with three sections: 'IIT KHARAGPUR' (with a logo), 'NPTEL ONLINE CERTIFICATION COURSES' (with a logo), and 'Hardware Modeling Using Verilog'.

This has I said, this was the time scale we give in that example. So, when we write hash 5, it means a delay of 50 nanoseconds. Now instead of nanosecond, I can use other units also, I can use s meaning second ms millisecond, then I can write us microsecond, ps picosecond or fs femtosecond, this can be specified. But again these are all for simulation, this has nothing to do with the actual delay of the circuits, ok, these are just very rough estimates which the designer has given for the purpose of simulation.

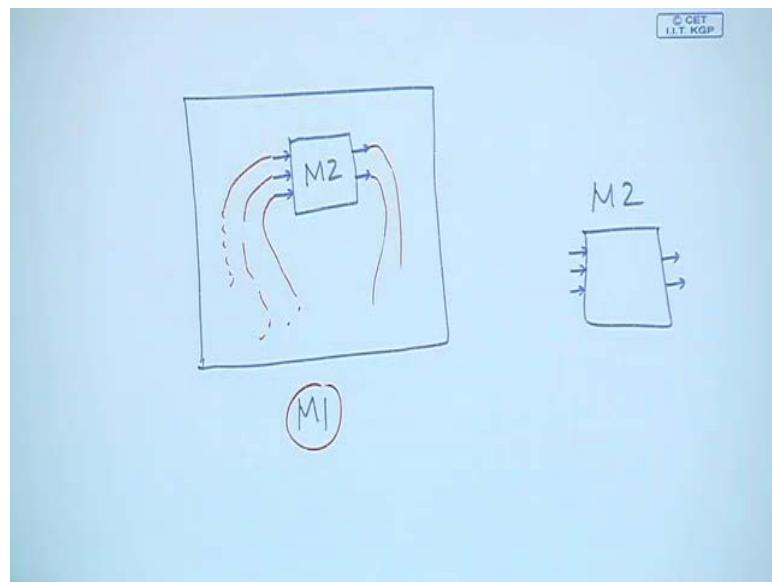
(Refer Slide Time: 14:38)

Specifying Connectivity during Instantiation

- When a module is instantiated within another module, there are two ways to specify the connectivity of the signal lines between the two modules.
 - Positional association
 - The parameters of the module being instantiated are listed in the same order as in the original module description.
 - Explicit association
 - The parameters of the module being instantiated are listed in arbitrary order.
 - Chance of errors is less.

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

(Refer Slide Time: 14:49)



Ok, now, the question comes how to specify connectivity that when you are instantiating a module within another module, like what I mean is that suppose I have a module, this is M1 and I have another module M2 and I am instantiating this module M2 inside here. So, M2 had some input signals, M2 had some output signals. So, here also the same input and output signals will be there. So, I will have to suitably connect this input and output signal from the other means hardware components that are there in M1 so that this instantiation will work in a proper way, right.

So, there are two ways to specify this connectivity, how these input and output lines are connected. The first is called positional association, where the parameters of the modules that is being instantiated are listed in the same order as in the original module description. So, in the original module description, let us say we are defining a full adder. So, if it was sum, carry a, b, c, then when you are instantiating, we have to specify in the same order sum, carry a, b, c.

So, we cannot change the order, ok, but there is an alternate method called explicit association, where we can change the order. We can specify the parameters in an arbitrary order, but we shall see how? I mean arbitrary order the advantage is that here the chance of error is less because suppose, when you wrote the module for the full adder, you give the first two parameters sum and carry, but when you are instantiating by mistake you have written carry and sum. So, in simulation you will see that result is not coming correctly. So, you will have to find out where the error is there, you have to debug it, but if you use this explicit association then this kind of errors, the chance of occurring will be much less, we shall see how we can use this, ok.

(Refer Slide Time: 17:12)

The screenshot shows a presentation slide with a dark blue header and footer. The main content area has a light blue background. On the left, there is a large code block for a testbench module. On the right, there are two smaller code blocks: one for 'Positional Association' and one for 'Explicit Association'.

```

module testbench;
reg X1,X2,X3,X4,X5,X6; wire OUT;
example DUT(X1,X2,X3,X4,X5,X6,OUT);

initial
begin
$monitor ($stime, "X1=%b, X2=%b,
X3=%b, X4=%b, X5=%b, X6=%b,
OUT=%b", X1,X2,X3,X4,X5,X6,OUT);
#5 X1=1;X2=0; X3=0; X4=1; X5=0; X6=0;
#5 X1=0; X3=1;
#5 X1=1; X3=0;
#5 X6=1;
#5 $finish;
end
endmodule

```

Positional Association

```

module example
(A,B,C,D,E,F,Y);
wire t1, t2, t3, Y;
nand #1 G1 (t1,A,B);
and #2 G2 (t2,C,~B,D);
nor #1 G3 (t3,E,F);
nand #1 G4 (Y,t1,t2,t3);
endmodule

```

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

So, we shall be explaining with an example, first positional association. So, we take a simple example, let us say, we have an example, some module name is example and there are seven parameters A, B, C, D, E, F, Y. So, Y is the output and A, B, C, D, E, F are the input.

Now, inside a test bench, test bench is also module. So, we are instantiating this example, we are calling a DUT and these are the variable names you are giving, there are total seven parameters. You are giving X1, X2, X3, X4, X5, X6, OUT, but we are specifying them in the same order. The first six inputs, we are connecting X1, X2, X3, X4, X5, X6; these you are applying and the last one is the output, that is OUT and OUT will be observing, OUT we are printing, monitor, right. This is positional association, this same order in which we have defined, the same order will be using in the instantiation. But in the other one explicit association, what you do is something like this, in the original example it was A, B, C, D, E, F. So, the output was last. Suppose when I instantiate; I want output to be coming first.

So, what I have here is, here with respect to this module, the output first has to be connected to OUT, I write dot OUT (.OUT). Now within bracket, the variable name of the original module description which is Y, which is Y, which means Y is being connecting to OUT (.OUT(Y)). Similarly, I write dot X1 within bracket A (.X1(A)); that means, X1 is connecting to A, then X 2 is connecting to B. You see here although we have to write more, but we can clearly see that which variable is getting connected to which parameter of the module that you are instantiating and this can be put in any order, not necessary A, B, C, D, E, F, I can put F first, then D, then A, then C, ok. So, the order is also not important here, I can put in any order, this is one way to specify the parameters in instantiation.

(Refer Slide Time: 19:42)

Hardware Modeling Issues

- In terms of the hardware realization, the value computed can be assigned to:
 - A "wire"
 - A "flip-flop" (edge-triggered storage cell)
 - A "latch" (level-triggered storage cell)
- A variable in Verilog can be either "*net*" or "*register*".
 - A "*net*" data type always map to a "wire" during synthesis.
 - A "*register*" data type maps either to a "wire" or a "storage cell" depending upon the context under which a value is assigned.

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

Now, some of the hardware modeling issues, well actually when you use the assignment statement, various kind of assignment statements we have seen. We have seen the assign statement, we have seen the equal and the less than equal.

Now, depending on the type of assignment statement you are using, the value that is computed will be assigned to either a net type variable typically a wire or a reg type variable which can be ultimately be a flip-flop or a latch. A flip-flop is nothing, but an edge triggered storage cell, where data will get stored whenever a clock edge comes and latch is level triggered whenever some enable signal is activated. Now a variable in Verilog as I have said can be either net or register. A net during synthesis will always map to a wire, this is a matter of fact. Similarly, a registers which you declare in Verilog, this will map either to a wire or to a storage cell depending on how we have written the Verilog code. So, we shall be seeing some examples later. So, we shall see this distinction, sometimes the register can be mapped to a wire, sometimes it can map to a storage itself.

(Refer Slide Time: 21:09)

The screenshot shows a Verilog module named `reg_maps_to_wire`. The module has three inputs (`A, B, C`) and two outputs (`f1, f2`). Inside the module, there is a `wire` declaration for `A, B, C` and a `reg` declaration for `f1, f2`. An `always` block is used with the condition `@(A or B or C)` to assign values to `f1` and `f2`. The code is as follows:

```
module reg_maps_to_wire (A, B, C, f1, f2);
    input A, B, C;
    output f1, f2;
    wire A, B, C;
    reg f1, f2;
    always @(A or B or C)
    begin
        f1 = ~(A & B);
        f2 = f1 ^ C;
    end
endmodule
```

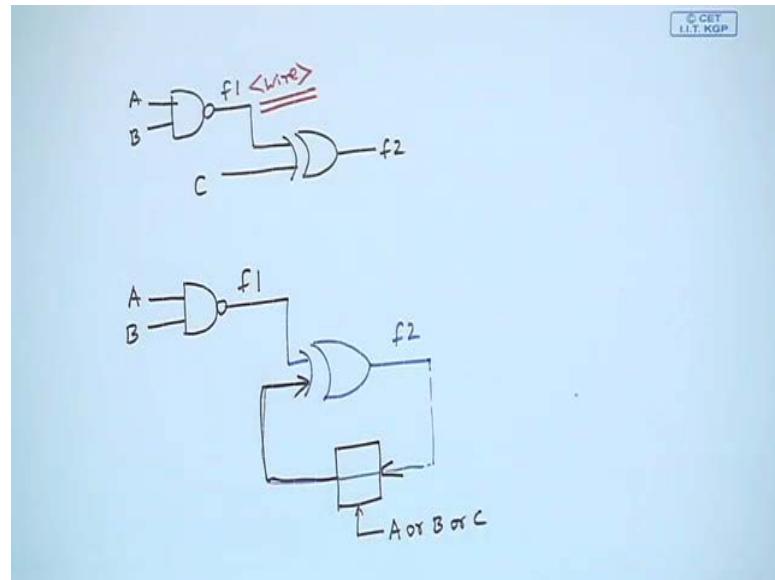
A callout box on the right side of the slide contains the text: "The synthesis system will generate a wire for f1."

The footer of the slide includes the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the title "Hardware Modeling Using Verilog". There is also a small circular portrait of a man.

Let us see some examples, this is a module where there are five ports A, B, C are the inputs f1, f2 are the outputs. Say A, B, C, I am also declared them as wires. This f1, f2 because they are appearing on the left hand side of the always, I am declaring them as a reg. Well here, I am not using a clock, here I am writing always at the rate A or B or C (always `@(A or B or C)`), what this statement means is that you execute the always block whenever either of A or B or C this signals are changing their state. So, whenever one or more of the

signals are changing, execute the always block, the meaning is this, ok. So, here f1 will be doing a NAND and f2 will be doing an XOR with f1 and this. Now here you see, the first one will be generating a NAND; next one will be generating an XOR, it will be like this.

(Refer Slide Time: 22:26)



First gate will be an NAND where the inputs will be A and B, and the output will be f1. Second one would be exclusive OR gate, where one of the input will be f1 and the other input will be C and the output will be f2. So, this will be a combinational circuit which will be generated and for f1, this will also be a wire, this f1 will also be a wire, there is no need for any storage cell here. So, here although we have declared f1, f2 as reg, both f1 and f2 will not require any storage cell either flip-flop or latch. Let us take another example, where we have changed this little bit, f2 equal to f1 XOR f2 and this. So, what does this say, similar declaration, but f1 is fine, for f1 there is no problem, f1 equal to NAND of A, B.

So, let us try to just also show this; what is happening. So, we are saying A, B this is f1, but for the other one; I am writing f2 equal to f1 XOR f2. So, what does this mean? what I am saying is that there is an XOR. So, one of the input is f1, the output is f2 and we are saying that is output is as if, the second input, right. So, you see, this kind of a design is normally not recommended, what will happen here is that. So, if you do this kind of a design, for f2 a storage cell will be generated, but for f1 no need, f1 is a simple wire.

So, what will happen is that for f2 here there will be a latch generated. So, the value of f2 will be stored here and the stored value will be coming here and this latch will be enabled whenever either A or B or C is changing. This will be how the hardware will be implemented, right, this is the meaning. So, here f2 is appearing both on the left hand side, on the right hand side that is why a storage cell will be generated.

(Refer Slide Time: 25:12)

```
// A latch gets inferred here
module simple_latch (data, load, d_out);
    input data, load;
    output d_out;
    wire t;
    always @(load or data)
    begin
        if (!load)
            t = data;
        d_out = ~t;
    end
endmodule
```

The "else" part is missing. So a latch will be generated for "t".

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

So, let us take another example, this is actually the example of a latch, where there are three ports data, load and d_out. So, what you are writing input data and load are the input, d_out is the output and t is a temporary wire t. We are saying always whenever load or data changes, begin if load is 0, load 0 active, then data will go to t and d_out will be NOT of t. See here, the, this thing, I will explain in more detail later just one first look of this; in this if statement, there can also be a else, if then else, we have specified if load is 0, then do this, but we have not mentioned what will happen if load is 1. So, the meaning is if load is 1, the value of t should not change which means the value of t must be stored in a latch, ok.

So, in case of this kind of incomplete, if then else statement, if you have, that will also generate a latch for some of the variables, ok. This one example illustrates; we shall be looking at some more examples later also. So, with this, we come to the end of this lecture where we have looked at again some of the Verilog constructs and we shall be continuing with the discussion, there are a lot of other things to discuss also. We have looked at the

different ways of assigning some values to variables, one is using the assign statement another is equal to, the equal to statement another is less than equal to. So, what are the differences between these? Assign, we have already seen, but what are the other two? So, we shall be seeing this in our next lectures.

Thank you.

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

Lecture - 09
Verilog Operators

So, we continue with our discussion on the various features of the language Verilog. So, in this lecture, we shall be discussing about the various Verilog operators. Like in any programming language, well you can take as an analogy C, C++ or java. There are some operators which are built in the language, which can operate on some data items, some numbers, some characters, some values and they produce some results. So, in the same way in Verilog, there are a set of operators, as if you see some of them are similar to the operators which are available in high level languages like C or C++. But there are a few others which are very specific, and they have some direct connection with the hardware that Verilog is supposed to model, ok.

(Refer Slide Time: 01:27)

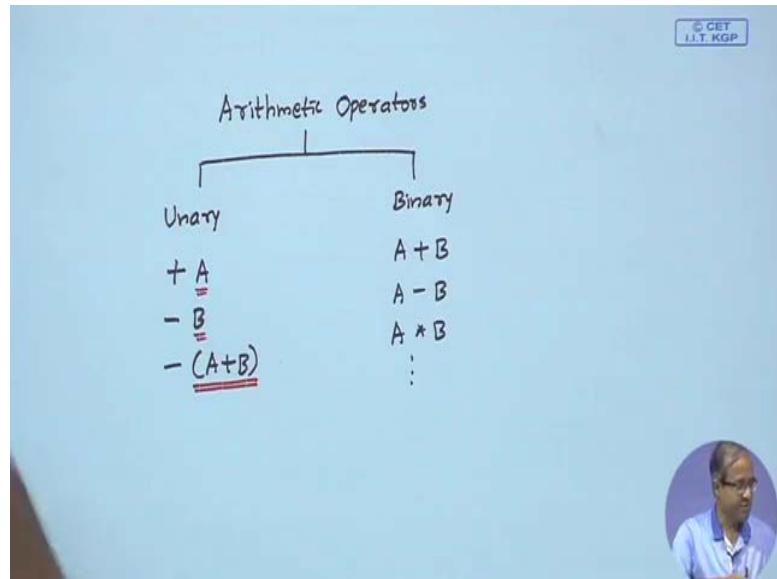
Verilog Operators

Arithmetic Operators:	Examples:
+ unary (sign) plus - unary (sign) minus + binary plus (add) - binary minus (subtract) * multiply / divide % modulus ** exponentiation	- (b + c) (a - b) + (c * d) (a + b) / (a - b) a % b a ** 3

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

So, we start with this lecture on Verilog operators. So, we start with the most basic kind of operators which we see in any language. The operators that allow us to carry out some arithmetic, these are the so-called arithmetic operators.

(Refer Slide Time: 01:50)



Now, broadly speaking if you just think of the Arithmetic Operators, so these operators can be broadly classified into either Unary or Binary. Unary means, this kind of operator will be operating on a single operand or data item. Binary means it will be operating on two data items. For some examples of Unary are you are quite familiar with this kind of notation, you can write plus A ($+A$) or you can write minus B ($-B$), these are something which is quite valid expression, we can write minus within parenthesis, let say A plus B. So, these minus and this plus, these are unary operations, they are working on a single data item.

So, in the first case, this is A, here this is B, and here this is just whole thing, this minus is operating on this whole thing. And in contrast the Binary operators, they operate on two data items like A plus B ($A+B$) when you write. So, this plus is operating on A plus B, A minus B, A multiplied B and so on, there are several others, let see. So, in Verilog the operators that are supported are as follows. So, you have two unary operators one is plus, one is minus. And among the Binary operators, some of them, I have already mentioned, we have addition (+), subtraction (-), multiplication (*), division (/), modulus (%), modulus means you divide by the second opponent and take the remainder, and double star means exponentiations (**). Some of the examples are shown here.

This is an example of Unary operator minus, while this plus is a Binary operator. Here minus star and plus, these are all Binary operator. This minus is working on a and b; star

is working on c and d; and this plus is working on this first number and the second number. Similarly, this slash, mod, exponentiations, this means a to the power three, these are the so-called Verilog operators. And as you see that the Verilog operators are somewhat very similar to what you see in the other programming languages like C with the exception of the exponentiation operator, the others are I means, all most the same as those available in the language C or C++, ok, fine.

(Refer Slide Time: 04:55)

Logical Operators:	Examples:
<ul style="list-style-type: none"> ! logical negation && logical AND logical OR 	<ul style="list-style-type: none"> (done && ack) (a b) ! (a && b) ((a > b) (c ==0)) ((a > b) && !(b > c))

- The value 0 is treated as logical FALSE while any non-zero value is treated as TRUE.
- Logical operators return either 0 (FALSE) or 1 (TRUE).

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

So, let us move on. So, we now look at Logical Operators. Well, when we talk about Arithmetic Operators, we are operating on some numbers, numbers can have some values, which we can add, subtract, multiply, divide, and so on. When you talk about Logic Operators, we are considering logic values, true or false, and we can combine several such logic values into a single result, which is again a logic value, true or false, these are the so-called Logic Operations or Logical Operations. So, in Verilog, there are three kinds of Logical Operators that are supported negation which is NOT (!), logical AND which is denoted by double ampersand (&&) and logical OR which is denoted by double bar (||). These are again very similar to the language C or C++; such things are available there also.

So, these operators are used in some conditionals, while here I am not shown the complete examples, I have only shown the logical expressions, but actually we will be using this in some kind of control constructs like let say in if then else statement. So, you are aware of if then else kind of statements. So, when I specify A condition in an if statement, so I can

use this kind of Logical Operators in that condition expression, ok. So, some examples of condition expressions are shown here done and ack (done && ack). So, here it is assumed that done and ack are both logical expressions or of values true or false, and you are defining the logical AND.

(Refer Slide Time: 06:59)

<u>$\&\&$</u>		<u>$\$</u>	
A	B	A && B	A B
F	F	F	F
F	T	F	FT
T	F	F	FT
T	T	T	TT

<u>$\overline{\overline{A}}$</u>	
A	\overline{A}
F	T
T	F

Now the way this logical AND is defined, it is just follows the AND operator. Like say if you have two operators, two operands A and B and the output which is A AND AND B (A && B) then if both are false, this will be false; if one of them is false then also it will be false; only when both of them are true, then only this expression will be true. So, this is how this AND operation is defined. Similarly, you can have this OR, A OR B (A || B), where A is a logical expression, B is another logical expression which has again values true and false and they are combined with an OR.

So, just to recall the OR operations will be defined as follows. So, when both of them are false, then the OR operation is false, but if at least one of them is true, false true, true false, or true true, then this expression, sorry sorry, this will be true, this expression will be returning true, this is the definition of OR. So, if at least one of the inputs are true, then the expression will be true. And the NOT operation is just the logical negations. So, you have single input and you have NOT of A; if A is false, NOT A is true; if A is true, NOT A is false, right, these are the three Logical Operators which are supported by Verilog.

So, here you can see some other examples are given of logical expressions NOT a AND AND b ($!(a \&& b)$). So, here this a and b will be done first then NOT of that. Here there are two expressions I means a greater than b, this will be either true or false; c equal to 0, this will be a true or false, and you are combining them using this OR. Similarly, here there are two, a greater than b, b greater than c, so you are first doing NOT of this, then you are combining these two using AND, ok.

So, the point to notice that so while you are evaluating such expression like when I have written done AND AND ack (done $\&\&$ ack), the values of this two variable done and ack, if the value is zero, then it will be treated as false, but any nonzero value will be treated as true. And as I said the logical operator will return the value again either true or false, but the values will be 0 for false and 1 for true that is the convention which is followed in this kind of logical operations, fine.

Now, let us come to the Relational Operators, these are again a very standard feature in any programming language. We often have to compare two numbers to check whether they are greater, less than, equal and so on. So, these Relational Operators are used for that purpose. So, they will be typically taking two numbers as input, they will be doing some kind of relational operation; and the result that they will be giving you will be true or false.

(Refer Slide Time: 10:43)

Relational Operators:

- \neq not equal
- \equiv equal
- \geq greater or equal
- \leq less or equal
- $>$ greater
- $<$ less

Examples:

- $(a \neq b)$
- $((a + b) \equiv (c - d))$
- $((a > b) \&\& (c < d))$
- $(\text{count} \leq 0)$

Let us see. So, in Verilog, six such Relational Operators are supported, not equal (\neq), double equal to sign means equal (\equiv), greater than or equal (\geq), less than or equal (\leq),

greater than ($>$), less than ($<$). So, some relational expressions are as follows a not equal to b ($a \neq b$), a plus b equal to c minus d, a greater than b and c less than d, count less than zero. Well, these expressions, you can use in some control constructs, which will be studying later like if then else or while or for these kind of control constructs, ok alright. So, here has I said, so typical Relational Operator, let say a not equal to b ($a \neq b$), they operate on numbers. So, just assume that both a and b are numbers, and the result that is generated, it will say whether this condition is true or false. So, it is a Boolean value, right, these are true for Relational Operators.

(Refer Slide Time: 11:50)

Bitwise Operators:	
\sim	bitwise NOT
$\&$	bitwise AND
$ $	bitwise OR
\wedge	bitwise exclusive-OR
$\sim\wedge$	bitwise exclusive-NOR

Examples:

```
wire a, b, c, d, f1, f2, f3, f4;
assign f1 = ~a | b;
assign f2 = (a & b) | (b & c) | (c & a);
assign f3 = a ^ b ^ c;
assign f4 = (a & ~b) | (b & c & ~d);
```

Next, there are some Bitwise Operator. Again these Bitwise Operators not exactly all of them, some of them at least are available in a language like C. So, Verilog is supposed to model a hardware and at the level of hardware we talk about bits, we talk about the logic operations. So, these bitwise operations are very important in Verilog, because at times when we just model some logic expressions at the bit level, we use this kind of bit level operators, ok. So, the Bitwise Operators that are supported are bitwise NOT is denoted by this tilde symbol (\sim), single ampersand means bitwise AND ($\&$), single bar is bitwise OR ($|$), hat is exclusive OR (\wedge); and tilde hat is exclusive NOR ($\sim\wedge$).

So, we have seen some examples earlier where you use the assign statements to model some combinational functions. So, here again I am showing some examples where some of this operators are used. You see we have defined some variables a, b, c, d, f1, f2, f3, f4

as wires. Let see in a first statement, we have said assigned f1 equal to NOT a OR b. you see these are all Bitwise Operators. First the value of a is inverted, NOT of a then that value is bitwise ORed with the value of b. Similarly, in the second statement, which is actually similar to the evaluation of the carry of a full adder, so you take the AND of a and b, b AND c, c AND A and then OR them together. f3 is take the exclusive OR of three variables a, b, c, a XOR b XOR c; and f 4 says a AND NOT b OR b and c AND NOT d. So, in terms of Boolean expression, this is similar to a $b_{\bar{}}$ OR plus $b \cdot c \cdot d_{\bar{}}$, ok. So, Bitwise Operators as I said, they operates on bits, all the operators are single bit variables, they are not vectors and they return a value f1, f2, f3, f4, those are also of type bit, single bit, ok, these are something to remember.

(Refer Slide Time: 14:34)

The slide has a yellow header bar with the title 'Reduction Operators' in white. Below the header, there is a text box containing the following information:

Reduction operators accepts a single word operand and produce a single bit as output.

- Operates on all the bits within the word.

Reduction Operators:	
&	bitwise AND
	bitwise OR
$\sim\&$	bitwise NAND
$\sim $	bitwise NOR
\wedge	bitwise exclusive-OR
$\wedge\wedge$	bitwise exclusive-NOR

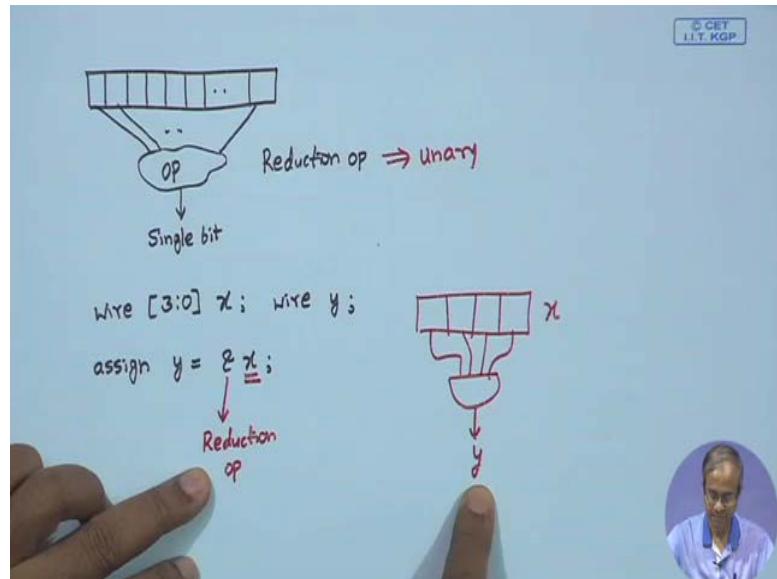
Examples:

```
wire [7:0] a, b, c; wire f1, f2, f3;
assign a = 4'b0111;
assign b = 4'b1100;
assign c = 4'b0100;
assign f1 = ^a; // gives a 1
assign f2 = & (a ^ b); // gives a 0
assign f3 = ^a & ^~b; // gives a 1
```

The footer of the slide includes the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the title 'Hardware Modeling Using Verilog'. There is also a small circular video player showing a person speaking.

So, here we have something which is normally not present in the kind of hardware or the high level languages with the familiar with like C or C++, these are called Reduction Operators. Now, what is a Reduction Operator? Reduction Operator is trying to reduce a set of items into a single item, this is from where the name reduction comes.

(Refer Slide Time: 15:12)



So, let me give you an example. Suppose I have a word, you can treat it as a vector, multi-bit word. So, I just operate on this bits using one of the Reduction Operators, this is a Reduction Operator. So, what the Reduction Operator will do? it will take all the bits of my input word as input, this is that are multiple bits, and it will be generating a single bit result as the output, this is what is the purpose of a Reduction Operator. It converts multiple bits in a variable into a single bit of result, ok.

Let us see. So, this is actually what we have just said, it accepts a single word as input and it will produce a single bit as output. So, it operates on all the bits of the word. So, what are the kind of Reduction Operators available, single AND is bitwise AND ($\&$), this is bitwise OR ($|$), tilde AND is NAND ($\sim\&$), tilde bar is NOR ($\sim|$), this is XOR (\wedge), tilde hat is exclusive NOR ($\sim\wedge$). Well, how we just use this kind of Reduction Operator, it is simple. Suppose, I have a variable let us say I have declared a variable it is of type wire, let say 4-bits, I call it x. So, I can write somewhere let say I have defined another variable a single bit variable called y, I can write y equal to ampersand x ($y = \&x$), this Reduction Operators are all these are Unary Operators, this is something to also remember, they operate on a single data item or operand. So, this ampersand works on a single data item x and this is your Reduction Operator, reduction.

So, what this Reduction Operators is doing? it is taking the bitwise AND of all the 4-bits of x, this x was 4-bits. So, actually logically speaking this will be like AND gate, this is

your x, and the output of the AND gate will be your y, right. So, this Reduction Operator actually models a gate with multiple number of inputs. This ampersand x means this is an AND gate, it will take all the bits of this variable x as input, and it will generate this y as the output, right. So, depending on the kind of operator you have used, so the type of this gate will differ; like as I said we can have so many different types of gates.

Well, some examples are shown here. So, you see, so, here we have defined three words a, b, c, these are vectors of size 8, 7 to 0. and we have also declared three single bit variables of type wires f1, f2, f3. And a, b, c we have just initialized them with some value. Well, here just you can assume for this example at least that this instead of 7, you can consider as 3. So, there is no harm in that, so it will be easier for yours understand. So, let us assume these are 4-bit numbers. So, I am initializing a, b, c with this 4-bit constants 0111, 1100, 0100.

(Refer Slide Time: 19:26)

$a = 0111$ $f1 = 0 \oplus 1 \oplus 1 \oplus 1 = 1$
 $b = 1100$
 $c = 0100$
 $\therefore \begin{array}{r} 0111 \\ 1100 \\ \hline 1011 \end{array}$
 $\wedge: 1011$
 $\ominus: 0 \quad f2=0$
 $\wedge a = 1$
 $\sim \wedge b = 1$
 $\sim a \& \sim b = 1 \oplus 1 = 0$
 $f3=1$

So, this initial values are a is 0111, b is 1100 and c is 0100. Now, the first Reduction Operator I am using here in this example this is a f1 equal to XOR a ($f1 = \Lambda a$), which means it will do a bit by bit XOR of all the bits of a and it will generate f1. So, what will be the f1? So, f1 will actually the bit by bit XOR of these 4-bits. So, it will be 0 XOR 1 XOR 1 XOR 1. So, you know XOR actually counts the number of ones whether it is odd or even. Here since it is odd number of ones, the result of the XOR will be 1. Similarly,

here f2 is actually doing a XOR b and AND of that; that means, this is your Reduction Operator. First a and b, you are taking an XOR.

So, let us see, a is this, b is this. So, a was 0111, b was 1100. You take an XOR first, XOR operation. So, 1 and 0 is 1, 1 and 0 is 1, 1 1 is 0, 0 and 1 is 1. This is your XOR and then you are using a Reduction Operator ampersand (&), you are taking AND of all this 4-bits 1011, because at least one zero is there, the AND operation will give you a result zero, right. So, your f2 will be 0. The third example says, you do a reduction operate on a, then you do a AND on another reduction operate on b. So, this is XOR and this is XNOR. So, XOR on a then XNOR on b, AND of that, this is a Logical Operator, let us see this is Bitwise Logical Operator.

First let us do this, XOR of a. So, XOR of a, what will be the value? So, already you have done it, bit by bit XOR of a, it will be 1, and XNOR of b. So, you do XNOR. So, there is even number of ones in b. So, for even number of ones, XNOR becomes one. So, you take AND of these two, XOR a and XNOR b, so 1 and 1, so result will be 1. So, your f3 will be 1, right. So, this example as we show you how this Reduction Operators work.

(Refer Slide Time: 22:39)

Shift Operators:

- >> shift right
- << shift left
- >>> arithmetic shift right

Examples:

```
wire [15:0] data, target;  
assign target = data >> 3;  
assign target = data >>> 2;
```

Conditional Operator:

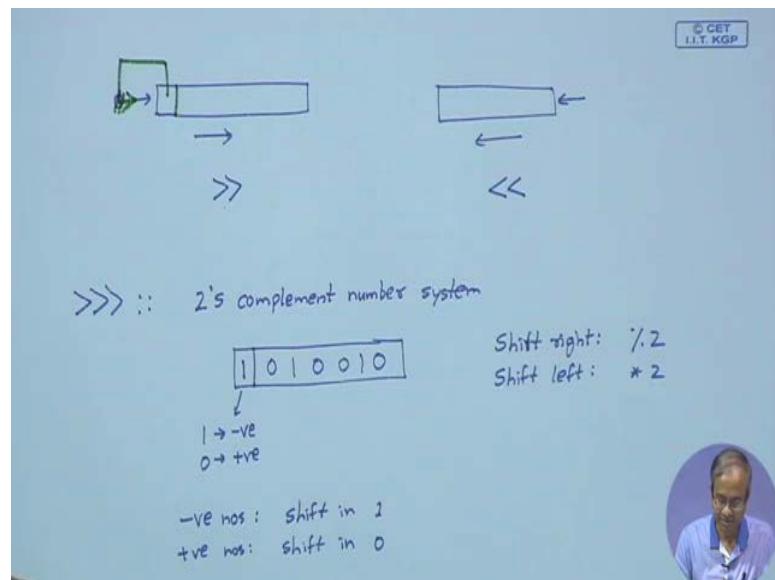
```
cond_expr ? true_expr : false_expr;
```

Examples:

```
wire a, b, c;  
wire [7:0] x, y, z;  
assign a = (b > c) ? b : c;  
assign z = (x == y) ? x+2 : x-2;
```

Next comes Shift and Conditional Operators. Now, the first two kind of Shift Operators are available in a language like C, which simply specifies right shift or left shift. So, here let us see the example I have given, we have defined two variables data and target of vector of size 16. So, if I say data right shift by some number three, it means you shift it right.

(Refer Slide Time: 23:18)



So, when I have a data in a register in a vector, I shift it right by three positions, and when I shift it right, zeros will be inserted on the left, ok. Similarly, when we shift it left, then zeros will be inserted on the right. So, this is the shift right operator and the shift left operator. So, I can specify by how many bits I am shifting, right. So, I can shift it, here in the example I am showing and I am shifting by three positions. But there is a special version of right shift which is specified by three greater than symbols ($>>>$), three greater than means arithmetic right shift.

So, what does arithmetic right shift mean, well in arithmetic right shift, in order to understand its basic purpose, we need to understand the 2's complement number system very clearly. So, just to refresh your memory, so here in the 2's complement number system, whenever you represent a number, the most significant bit will indicate the sign, 1(one) means negative, 0(zero) means positive, right. So, the other bits will be something let us say the other bits will be something. Now, for a 2's compliment number, for any number in fact, if you do a shift right, shift right is actually equivalent to division by 2, shift right by one position; and shift left means multiplied by 2.

Now, for a 2's complement number, if you want implement shift right, the rule is that if the number is negative, for negative numbers, you shift in 1; and for positive numbers, you shift in 0. So, you see to implement such this is called arithmetic shift because we are doing some arithmetic operations here. To implement this, what we have said is there you see

for a normal right shift we said that always 0 will be inserted, but now I am making some changes here we are saying not 0, but whatever was this sign that same sign bit will be inserted. If it was negative, 1 will be inserted; if it was positive, 0 will be inserted right, so this is denoted by three greater than. So, here you have an example data arithmetic shift by two positions.

Well, here exactly this thing is available in a language like C, you have a Conditional Operator. So, we specify a conditional expression then a question mark then an expression, colon, another expression and finally, semi colon. The first expression here will mean that if the condition is true then you take this, if the condition is false then take this. So, some examples are shown here let say wire seven zero x, y, z, these are 8-bits. So, I have written some condition b greater than c. So, I have said if b is greater than c, the true expression is this, then b else c, you assign this to a. What does this mean? this means you assign the greater of b and c to the variable a. If b is greater, than this b will be assigned; if b smaller it will be false, then c will be assigned, ok. Similarly, here another example if this, let us say x and y, if these are equal, this is the condition, then you take x plus 2 else you take x minus 2 and assign it to another variable z, well. This is just example this a, b, c, x, y, z you can have anything, right. So, this kind of operator is also available.

(Refer Slide Time: 28:02)

Concatenation Operator:
 $\{..., \dots, ...\}$

Joins together bits from two or more comma-separated expressions.

Replication Operator:
 $\{n\{m\}\}$

Joins together n copies of an expression m, where n is a constant.

Examples:

```

assign f = {a, b};
assign f = {a, 3'b101, b};
assign f = {x[2], y[0], a};
assign f = {2'b10, 3{2'b01}, x};

```

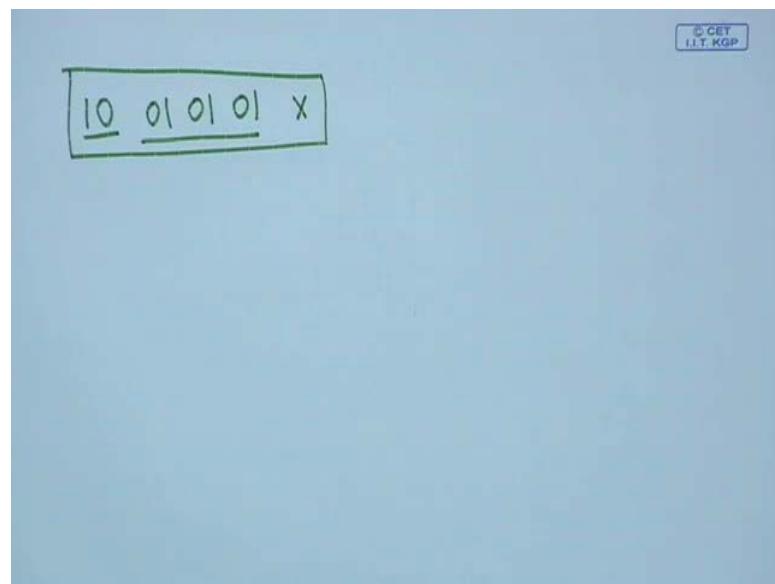
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

A circular portrait of a man with glasses and a blue shirt, likely the instructor.

And you have something called Concatenation Operation, where just I am showing the example within curly brackets, I can specify two or more items separate by commas, like

a comma b, a comma 3-bit 101 comma b. This will mean, all these three items in all these two items will be concatenated together, they will be concatenated together to form a single vector. Suppose a was a 4-bit vector, b was a 3-bit vector, if I write concatenate a comma b, it will become a 7-bit vector, a and b will be joined together, ok. And we can use this Replication Operator along it concatenation, where I can use a constant n followed by something within curly bracket, this will mean repeat n times like the last example shows. This is an example where I am concatenating three things, first is 2-bits 1 0, second is three times 2-bit 0 1.

(Refer Slide Time: 29:19)



So, what would you mean, you start with 1 0, this is a first one, then I am saying three times 0 1, which means 01, 01, 01, and the last item is a undefined x. So, there will be x in the end. So, my final concatenated result will be this 10 01 01 01 x, it will be a 9-bit vector.

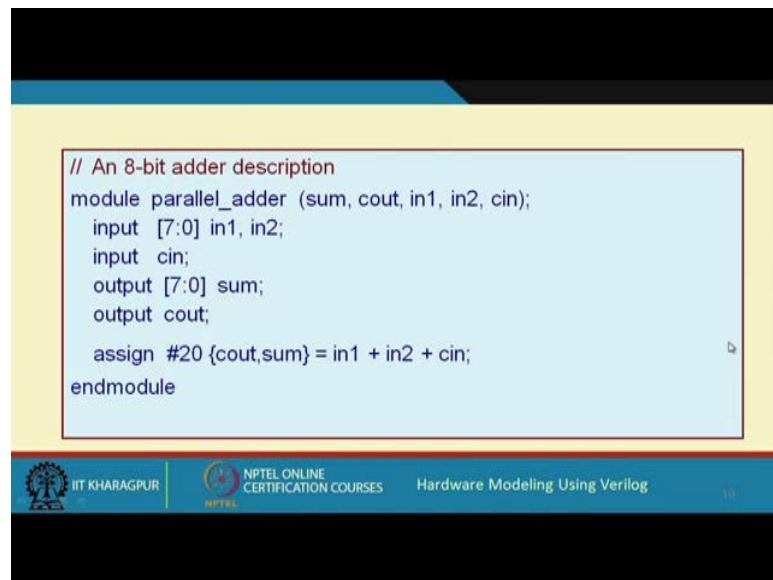
(Refer Slide Time: 29:46)

```
module operator_example (x, y, f1, f2);
    input x, y;
    output f1, f2;
    wire [9:0] x, y; wire [4:0] f1; wire f2;
    assign f1 = x[4:0] & y[4:0];
    assign f2 = x[2] | ~f1[3];
    assign f2 = ~& x;
    assign f1 = f2 ? x[9:5] : x[4:0];
endmodule
```

The screenshot shows a slide from an NPTEL online course titled "Hardware Modeling Using Verilog". The slide content is a Verilog module named "operator_example". The module has four ports: inputs x and y, and outputs f1 and f2. The Verilog code uses several operators: bit range selection (e.g., x[4:0]), binary AND (&), binary OR (|), NOT (~), reduction AND (bitwise AND), and conditional assignment (f2 ? x[9:5] : x[4:0]). The slide also includes the IIT Kharagpur logo and the NPTEL logo.

So, let us see some examples very quickly. So, here there is an example shows some of the operators in use. This is a module, complete module description, x and y are inputs; outputs are f1 and f2, x and y are defined as 10-bit vectors. So, f1 is a 5-bit vector; f2 is a single bit vector. So, here you see, here I have taken cross sections from the vector x and y, from bit number four to zero to make it 5-bits. I have done bit by bit AND, and I have assigned it to f1 because the size of f1 was 5, right. f2 is the single bit. So, I can take any single bit from x, any single bit from some other, let say f1, f1 three (f1[3]), I do a NOT then I do a bit by bit wise OR. This x, I do a reduction operation NAND, I assign it to f2, this is the conditional. Well, I am assigning something to f, so f1 is a 5 bit; if f2 is true then these 5-bit is assign, x bit number 5, 6, 7, 8, 9, this will be assigned. If f2 false then these 5-bit 0, 1, 2, 3, 4, these will be assigned, ok, here are some examples.

(Refer Slide Time: 31:11)



```
// An 8-bit adder description
module parallel_adder (sum, cout, in1, in2, cin);
    input [7:0] in1, in2;
    input cin;
    output [7:0] sum;
    output cout;
    assign #20 {cout,sum} = in1 + in2 + cin;
endmodule
```

The slide also includes logos for IIT Kharagpur and NPTEL, and the title "Hardware Modeling Using Verilog".

So, here we are showing the behavioral description of an 8-bit adder. So, here we are assuming that the number, input numbers are in1 and in2, cin is the carry in, and this is the sum and this is the carry out (cout). So, because it is an 8-bit adder, this in1 and in2 will be 8-bits. This cin is the carry in, sum is 8-bits, and cout is carry out. So, we can express this in behaviorally in a single statement like this, but of course, the addition will be this in1 plus in2 plus carry in. But you see, when you are adding two 8-bit numbers, the result can be 9-bits because there can be 1-bit of carry coming out.

So, if you want to keep everything of the result intact, it will be one bit more. So, how I, how I have specified it here? I have just written cout comma sum concatenation ({cout,sum}), which means the left hand side is a concatenated item which represents 9-bits. So, after this addition, the most significant bit will go in to cout, the remaining 8-bits will go in to sum, that will be the result. And this is just a delay that can be used for simulation.

(Refer Slide Time: 32:31)

Operator Precedence

- Operators on same line have the same precedence.
- All operators associate left to right in an expression, except ?:
- Parentheses can be used to change the precedence.

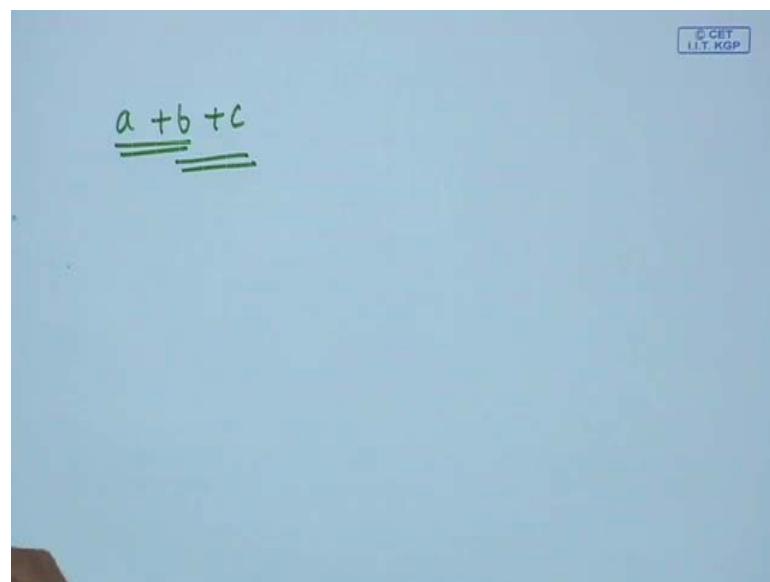
+ - ! ~ (unary)
**
* / %
<< >> >>>
< <= > >=
== != === !==
& ~&
^ ~^
~
&&
? :

↑
Precedence increases

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

And just we talked about so many operators. Just talking about the Operator Precedence, this chart actually shows you just overall picture, how the precedence varies. So, this is lowest priority, and this is highest priority, and operators on the same line have the same precedence. And excepting the last operator, the Conditional Operator, so all other operators, they are left to right associative.

(Refer Slide Time: 33:07)

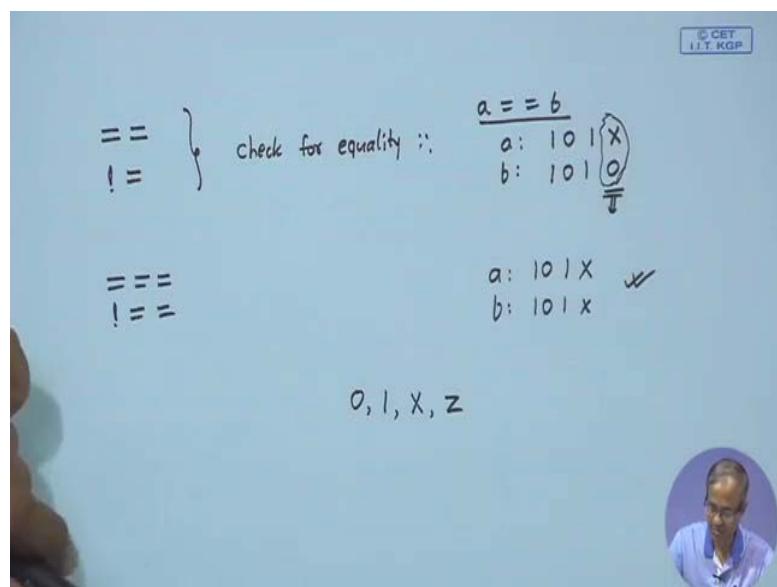


Means suppose I write a plus b plus c, this will mean first this left most addition will be done, then the next one will be done, left to right, ok. Now, you see the highest priority are

the Unary Operators, plus, minus, NOT. NOT, there are two different notations either this or tilde, then are the Arithmetic Operations. So, among this Arithmetic Operations, we have exponentiation in the highest, then comes the other operators multiply, divide, mod. Well here I have not shown, then will come addition, subtraction will be there. So, I have not shown all the operators in fact.

And then the Shift Operators, then the Relational Operators and Relation Operator: less than, greater than are the higher priority and equal to, not equal to checking will be lower priority. Then comes the Bitwise Operators, and this highest priority, then XOR, then OR. This is what the priority values which Verilog assumes, then the Logical Operators, double ampersand, double OR and lastly the Conditional. Now, here you see for equality checking, I have talked about double equal to not equal to; here you see there is another type of equal to triple equal to and not triple equal to.

(Refer Slide Time: 34:45)



So, let us very quickly talk about this thing. So, so you have equal to and not equal to. So, we are saying there is another version, where we can write it like this. Now, when you specify this, this means you check for equality, following a rule like this. Suppose I am checking two numbers a and b, let say a is the current values 101 undefined x, b is 1010. So, if you make the comparison then you see, this x and 0 will be compared, and x and 0, this will be considered to be matched.

But when you are using three equal to, then this means exact equality. So, if a is 101x and B is also 101x, then only this equality will be returning true. You recall that in Verilog, we have a four valued logic system, where the logic value supported at zero, one, don't care or undefined and high impedance z. So, if you write triple equal to, then these values are matched in an exact way, x and x, z and z, 0 and 0, 1 and 1, but if it is double equal, then it is not done that way, ok, this is just the only difference.

(Refer Slide Time: 36:35)

Some Points

- The presence of a 'z' or 'x' in a *reg* or *wire* being used in an arithmetic expression results in the whole expression being unknown ('x').
- The logical operators (!, &&, | |) all evaluate to a 1-bit result (0, 1 or x).
- The relational operators (>, <, <=, >=, ~, ==) also evaluate to a 1-bit result (0 or 1).
- Boolean *false* is equivalent to 1'b0.
Boolean *true* is equivalent to 1'b1.

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

So, these are few things which I have already mentioned most of them. And the first point is a little different that see when you are using an arithmetic expression. Let say we are using a and b, two numbers, we are adding a plus b. Let say a is a vector, b is also a vector. Suppose for one of the number, let say a, one of the bits is having a state x. So, when you do the addition, the entire sum will become x, this is a rule of Arithmetic Operation, ok. So, the presence of either an x or a z in any variable, which is used in an arithmetic expression, will result in the value of the whole expression to become unknown x. So, all the bits will become x and these we have mentioned Logical Operators generate 1-bit result, Relation Operators also generate 1-bit result; false is 0, true is 1, ok.

So, with this we come to the end of this lecture where we have tried to summarize, the various kind of operators which are available in the Verilog language with which we can specify the description or the model of the hardware system that we are trying to design or

build. So, in the next lecture, we shall be looking at some of the examples based on the structures and the constructs which we have seen so far.

Thank you.

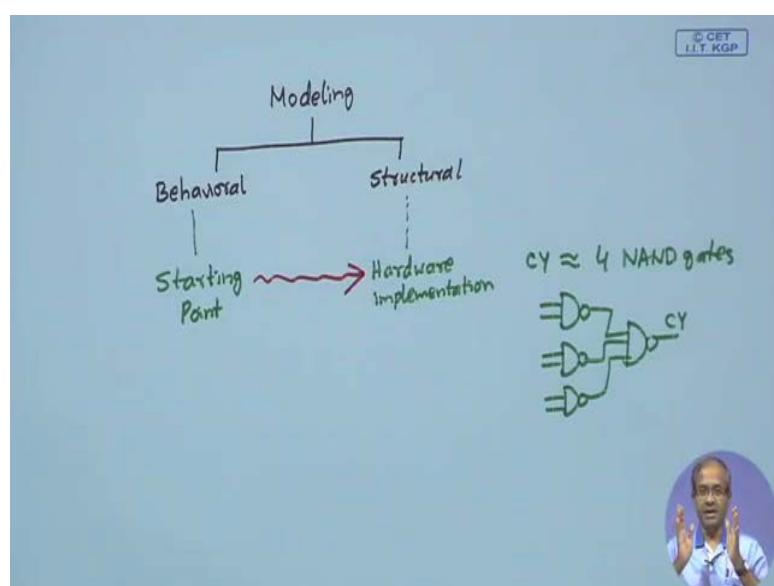
Hardware Modeling using Verilog
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 10
Verilog Modeling Examples

So, if you recall the topics that we have discussed during our last few lectures, we have talked about the various constructs of the Verilog language. We had looked at some of the fundamental constructs, the operators, how we can frame the expressions for evaluation and some of the operations we have defined for assignment. For example, using the assign statement, there are a few more, which will be discussing in the due course of the lectures. But what we are trying to deal with in the present lecture is, we shall take an example. And with the help of an example we shall try to illustrate a few points and concepts which are considered to be good design practice when someone is going about designing a digital system, using a hardware description language.

So, the topic of our lecture is Verilog Modeling Examples. Now actually what we trying to do here let me try to explain this first. You have seen that when we model some system using Verilog, broadly there are two fundamental ways of doing it. So, we call it Behavioral and Structural modeling.

(Refer Slide Time: 01:45)



You can do either using behavioral or using structural. Now in the example that I shall be giving in this lecture, we shall be just elaborating on this further.

Now, just to tell you about when we talk about behavioral modeling, what we are actually doing. We are actually saying that how a system is working, we are not telling or talking about how it is actually constructed or is to be constructed. We are saying that for example, I have an adder, this some output should be a $a \oplus b \oplus c$, they carry output should be $ab \oplus bc \oplus ca$. So, I am not telling that whether you will be using AND gate, OR gate, NOT gate, or NAND gate, or XOR gate, what or how you will be interconnecting them.

So, if I say just the behavior, either in the form of this expression or in the form of a truth table let us say that is a so called behavioral expression which is much easier to do. So, when you are designer, when you have something in your head, in your mind, you are trying to design something. So, the first thing that you can write or start with is a behavioral description of your model, or your thought process, what you are thinking about. Now once you have thought about the behavior, now you can think about how you can implement it, ok.

Let us say, let us take that same example of a full adder. So, once you have thought about what is the functionality of a full adder. Now you think about how you want to implement it. Now you may see that, well, I want to implement it using a technology, where you have only NAND gates. So, let us not use AND, OR, or NOT operation. So, all the AND, OR, NOT, whatever is there we will be translating them using Demorgan's Law, using NAND operations only and we will be implementing it using NAND; so just in this diagram.

So, as I had said behavioral description is often the starting point, because it is easier to write a Verilog module in the behavioral model, but structural description, this is more with respect to hardware implementation. Like here, I can say that well to implement the carry output of the full adder, let us say to implement carry, I need a , I will say that I will need four NAND gates, let us say. And I will also tell how the NAND gates are interconnected. I will say that well you interconnect the NAND gates like this, there will be three 2-input NAND gates and one 3-input NAND gate, this will be ab , bc and ca and this will be your final carry output (cy).

So, this is; what is your structural representation, you specify some modules and you also tell how they are interconnected. Now obviously, this is much more detailed. So, when

you are thinking of the behavior, you have not yet thought about these gates and how they are interconnected, ok. So, there will be a process that you will have to go through that from behavioral to structural, you will have to make a translation, right. So, the normal designed process is that we start with a behavioral description, now in our behavioral description our total description may consist of one or may be multiple modules.

Let say one, we systematically try to convert them into structural descriptions, that can be several steps. I shall take one example and I will show you that how you can use several steps in the translation, and how you can carry out some kind of a hierarchical design process in order to achieve it. At the end of which you will be having a number of modules, all of which are specified in a structural description. So, you have a total design which is a structural description of a system you want to built.

So, from behavioral to structural, this is normally the step which many people follow, ok. So, we shall be taking the example of a 16-to-1 multiplexer and illustrate some of the

(Refer Slide Time: 06:58)

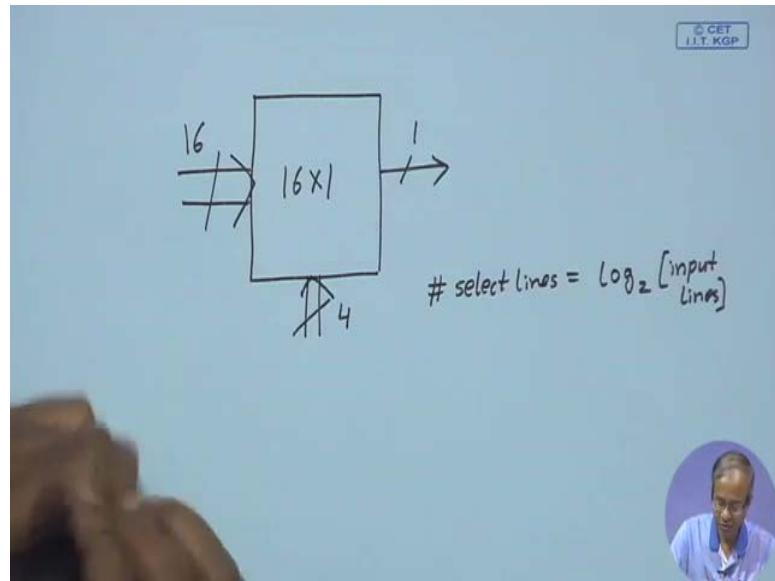
Example 1

- The structural hierarchical description of a 16-to-1 multiplexer.
 - Using pure behavioral modeling.
 - Structural modeling using 4-to-1 multiplexer specified using behavioral model.
 - Make structural modeling of 4-to-1 multiplexer, using behavioral modeling of 2-to-1 multiplexer.
 - Make structural gate-level modeling of 2-to-1 multiplexer, to have a complete structural hierarchical description.

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

points I have just discussed with respect to this. 16 to 1 multiplexer, you know what a 16 to 1 multiplexer is.

(Refer Slide Time: 07:19)



So, a multiplexer has a number of input lines, a single output line and a number of select lines. So, when I say a 16 to 1 multiplexer. So, there will be 16 input lines, there will be one output line, and to select one of the input lines there will be four select lines.

So, the rule is, number of select lines is given by log to the base 2 of the number of input lines. Here input lines is 16. So, log to the base of 16 will be 4, right. So, we are trying to design such a multiplexer. So, to start with, we shall be giving the behavior. So, I am just showing you in this list that how we shall be proceeding with our design. So, in the first step, we shall be using pure behavioral modeling, which we shall see will be rather simple, specifying the behavior of a 16 to 1 multiplexer is very easy.

Then we shall be implementing the 16 to 1 multiplexer using several 4 to 1 multiplexers.

So, we will be using structural description of this 16 to 1 mux, using 4 to 1 multiplexers, but the 4 to 1 multiplexers will still specify, using behavioral model. Then we shall be implementing the 4 to 1 mux using 2 to 1 mux and we will be using still behavioral model of 2 to 1 mux. And finally, we will be specifying these 2 to 1 mux, also in the structural get level form. So, that we have a complete hierarchical structural description. So, let us see. So, as we proceed with this example, these steps will be clear. So, we start with pure behavioral modeling.

(Refer Slide Time: 09:43)

The slide has a yellow header bar with the title "Version 1: Using pure behavioral modeling". Below the title is a code block containing Verilog code for a 16-to-1 multiplexer. The code is as follows:

```
module mux16to1 (in, sel, out);
  input [15:0] in;
  input [3:0] sel;
  output out;
  assign out = in[sel];
endmodule
```

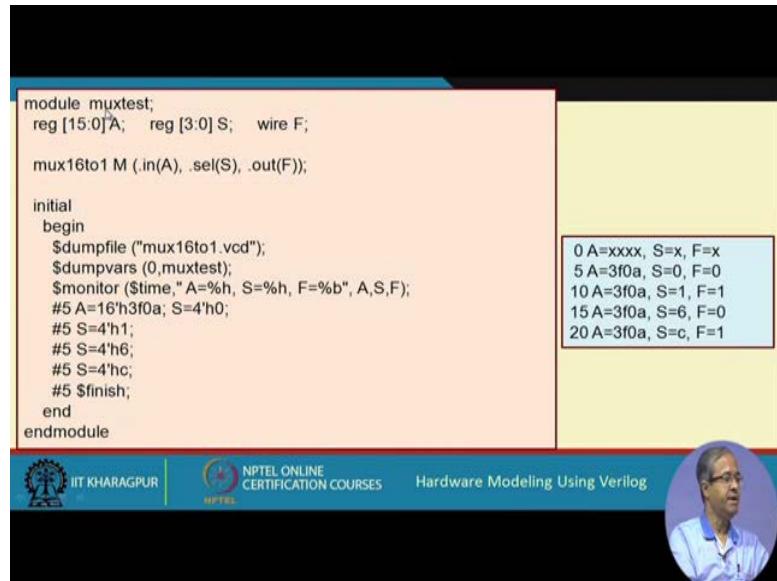
The footer of the slide includes the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the course title "Hardware Modeling Using Verilog". There is also a circular portrait of a man.

Very simple, this is the complete Verilog model for a 16 to 1 mux. You see for a multiplex, as I had said, there are three parameters, this in and sel are the input lines and the select lines. So, in a 16-bits, select lines is 4-bits and out is a single bit output.

So, you can specify the behavior of the multiplexer using the single statement, out equal to in, within square bracket sel (out = in[sel]). You see here as if we are accessing one element of an array, the input lines, there are 16 of them. You imagine the input in as consisting of a 16-bit array, where the index values will be 0, 1, 2, up to 15, and the select lines that you are using, 4-bit select line. So, in 4-bits, what will be the values if the value can range from 0 up to 15. So, if I can use this select line as the index of the array. So, I will be selecting that particular element, and I will be sending it to the output; that is what a multiplexer does, ok. So, you see here we have selected a particular bit of this input vector in, which bit, the bits specified by sel and that particular bit, we are storing in out, this is your multiplexer.

So, let us start with this, we have done a behavioral modeling of this multiplexor, ok. Now how do we test it? whether it is working correctly or not? we, sorry write a test bench. Just go back, we have a module like this, mux 16 to 1, where the three parameters: input, select and output, ok. And now just we want to verify it is operation through simulation, and you have written a test bench for it.

(Refer Slide Time: 12:03)



The screenshot shows a Verilog test bench for a 16-to-1 multiplexer. The code defines a module `muxtest` with inputs `A` (reg [15:0]) and `S` (reg [3:0]), and output `F` (wire). It instantiates a `mux16to1` module with inputs `.in(A)`, `.sel(S)`, and output `.out(F)`. An `initial` block contains a `$dumpfile` command to dump the simulation to a VCD file, followed by a `$monitor` command to monitor the variables `A`, `S`, and `F` at every time step. The monitor output is displayed in a separate window, showing the following sequence of values:

0 A=xxxx, S=x, F=x
5 A=3f0a, S=0, F=0
10 A=3f0a, S=1, F=1
15 A=3f0a, S=6, F=0
20 A=3f0a, S=c, F=1

The slide also includes the IIT Kharagpur logo, the NPTEL logo, and the title "Hardware Modeling Using Verilog". A small portrait of a man is visible in the bottom right corner.

So, what is our test bench looking like? it is here, we have instantiated our multiplexer mux 16 to 1. You see this was mux 16 to 1, the parameter names are in, sel and out. So, we have using this notation to specify the parameters dot in, dot sel, dot out. And in this module, the corresponding variables are A, S and F. And in order to express the data types you specify A and S as reg, because they will be appearing on the left hand side, they have to of type reg, and F is of type wire, these A is 16-bit, S is 4-bit.

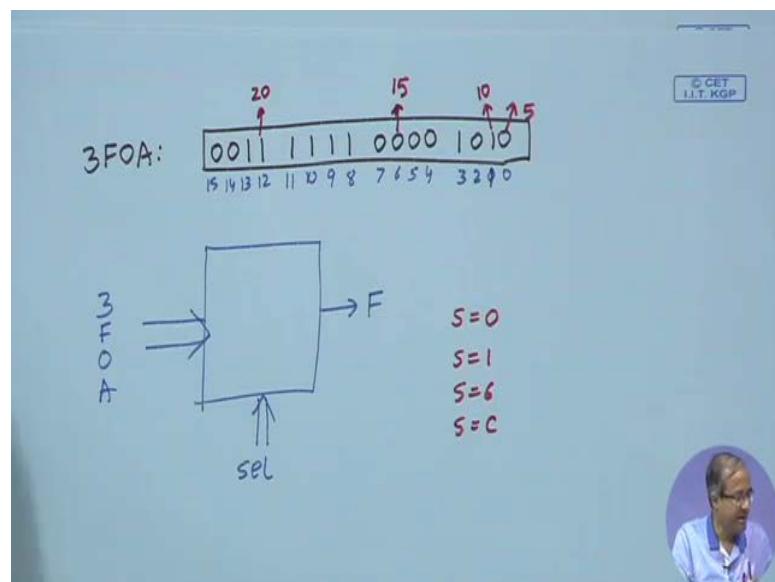
So, the test bench what I have doing, you recall, will be just looking more formal into the constructs of a test bench later. But let us illustrate with examples, this initial block, I am repeating, this means the we execute this begin end block only once; that is a purpose of the test bench. You will be testing your module once, whatever you have specified those values will be applying to the inputs, and you will be seeing what the outputs are coming, ok, that is the purpose of the test bench.

So, just ignore the first two lines once more, we will again come back and explain this once more. Well, this line monitor, monitor, we are monitoring a few of the variables. Dollar time is a system variable that indicates the simulation time, and then in a C like notation, we are saying that we want to print the values of A, S and F, h is a hexadecimal specifier, because A is a 16-bit number. I specify percentage h, S is also a 4-bit number, percentage h and F is a single bit, I give it b, b means binary. Monitor means, whenever any of the variables A, S or F changes, then only you print. You do not print at every time

1 2 3 4, do not continuously go on printing. So, whenever there is a change, then only you print; that is the purpose of this monitor statement.

Now, I specify the inputs. So, what I am saying is, that at time 5, I apply hexadecimal value, 16-bit hexadecimal h 3F0A.

(Refer Slide Time: 14:56)



So, let us write down here also 3F0A. What does 3F0A mean? Let us write it down 3 means 0011, F is 1111, 0 is 0000, A is 1010, these 16-bits. Now if you talk about the index values 3F0A. So, index values will be 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 and 15, these will be the index value. So, if you think of a multiplexer. So, these input, this 3F0A, you are applying here, and you are applying this select line, and you are getting the output F here, right. So, select line means, selecting one of these bits and that bit will come out, right.

Now, at time 5, we are selecting what? 0, we are applying all 0 to S. So, let us see what will happen. So, if we apply select line as 0, which mean this 0 will be selected. So, what will be getting, you will be getting this 0 as the output, right, and this will happen at time 5, right, at time 5. Next in the test bench what we do. Well, we wait for another time 5, after delay of 5 again, we set S to 1, 4 hexadecim value is 1. So, now, S is becoming 1. So, what will be the value? 1. Now the bit is, this 1.

So, this 1 will be now output it, and this will happen at time 10, because we are having another delay of 5, ok. You next look at; you have another delay of 5. Now you apply S

equal to 6. So, what is your 6? S is 6, 6 is here. So, this is 0. So, you will be getting a, this value 0. Now this will happen at time 15 and the last one, you apply S equal to C, C means 12, 1100, right. In hexadecimal S equal to C, C means 12. 12 means, you, here this is 1, this will happen at time 20, right. And at the end you finish after another time 5, you finish this simulation.

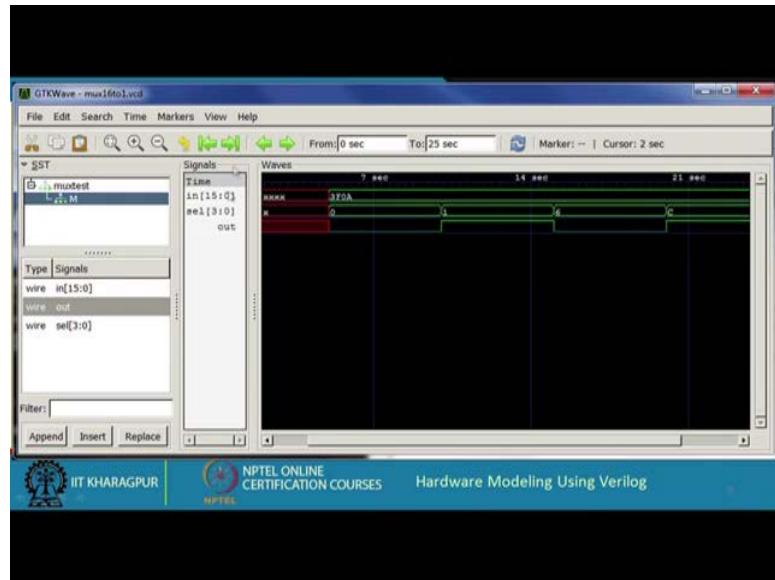
So, I means if I create this two files and I told you earlier that you can use any of the simulators to simulate these, and I had suggested to use the Iverilog one, which is very easy and just freely available. So, here in the examples that I am showing, I have used Iverilog and GTK wave, to show you the simulation outputs, ok. So, if I just simulate it using Iverilog. So, I get an output as is shown in the window here, this blue box, see exactly what I have shown here, this same thing is happening. See at time 5, 0 is coming, at time 10, 1 is coming, at time 15, 0 is coming, at time 20, 1 is coming. You see, the first value that is printed in the monitor is time, at value of 5 output is F0, 10 output is 1, 15 output is 0, 20 output is 1.

So, I have also printed the values of A and S for your reference, A and also S. Now initially at time 0, when this simulation starts; so A is not initialized, S is not initialized, F is not initialized. So, all of them are showing x right, ok. So, you see just from the behavioral specification, you can do a simulation, and you can get the simulation result like this. Now the first two lines in the initial statement, I am repeating this. I have mentioned this earlier also. See these variations with time, this information, you can also dump into a file VCD, ok. So, here I am giving a name mux 16 to 1 VCD, value change dump file, they will all be dumped in the file, and I can specify which variables I want to dump.

Well, 0 comma followed by the name of this module means, that all the variables in this module and all the variables which are instantiated inside those modules, everything will be dumped. Ok, we shall see what are the other options later in dumpvars, but here we are dumping all the variables. Now you can ask that well, I can see all the simulation output like this, why do I need to use this dump file, and dumpvars? the thing is that, well seeing the simulation output in a tabular form or a just on the screen is not good enough in the form of a text, well means often for digital circuits, we want to look at the timing diagrams, how the values are changing at what times and so on.

So, there is a tool, I mentioned about GTK wave. You can use the GTK wave tool to view this file, and since I have dumped it with the name. You can just give a name GTK wave, this particular file name, and it will show you a simulation result in the form of waveform like this.

(Refer Slide Time: 21:12)



So, you see the font size are little small, but you can decode, see on top, the time is mentioned, just going back at time 5, the values are starting to change, right. So, you see this is 7, this is 14 20 1. So, 5 is here, this is our in, select and out. So, in was initially xxxx, select was out, select was x, out was also x. So, it starts getting selected here; 3F0A. this is the value you have applied. Select initially you have applied 0. So, the output, you are getting 0, 0 level, then select, you have given 1, so output has given 1, then select, you have given 6, output has again become 0, then you have given C, output is become 1.

So, exactly what you are seen here in the tabular form, the same thing is being shown here in a graphical form, in the form of a waveform, ok. So, it is sometimes much easier to view it on waveform. If it is a quite a complex circuit, and the signals are means more complex and interacting. Now the thing is that, I have shown a very simple behavioral description of a multiplexes 16 into 1. I have shown how to write a test bench, and I am shown you the simulation results. Now, what I will do? I will proceed with our so-called hierarchical refinement. We shall be modifying our module description, but we shall be keeping our

test bench the same. We shall not be touching our test bench. We shall be making modifications.

We shall be again running the same test bench, and verify whether the simulation results that you are getting is exactly the same as what we got here. That should be so, because it should, the result should confirm to a 16 to 1 mux only, this we should verify at every step, fine.

(Refer Slide Time: 23:28)

**Version 2: Behavioral modeling of 4-to-1 MUX
Structural modeling of 16-to-1 MUX**

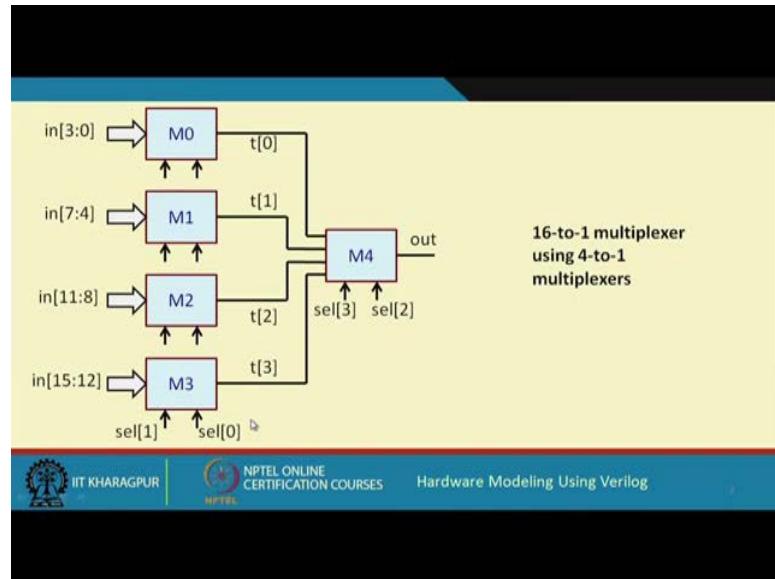
```
module mux4to1 (in, sel, out);
  input [3:0] in;
  input [1:0] sel;
  output out;
  assign out = in[sel];
endmodule
```

```
module mux16to1 (in, sel, out);
  input [15:0] in;
  input [3:0] sel;
  output out;
  wire [3:0] t;
  mux4to1 M0 (in[3:0],sel[1:0],t[0]);
  mux4to1 M1 (in[7:4],sel[1:0],t[1]);
  mux4to1 M2 (in[11:8],sel[1:0],t[2]);
  mux4to1 M3 (in[15:12],sel[1:0],t[3]);
  mux4to1 M4 (t,sel[3:2],out);
endmodule
```

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

So, we come with the version 2 now. Now what you do? see in version 1, we did something like this, we had a mux 16 to 1. Now here what we are doing. This is our mux 16 to 1 description. So, here before explaining this let me show you this diagram.

(Refer Slide Time: 23:51)



We are implementing a 16 to 1 multiplexer by using five 4 to 1 multiplexers. So, in the first level there are four of them. So, the 16 input lines are broken up into four each 0 to 3, 4 to 7, 8 to 11, 12 to 15, and in the second level, the output of these multiplexers are feeding the input of this, and among the four select lines, the least significant two select lines are applied here in this levels sel0 and se 1 and the high order bits are applied here.

So, this is how we built a 16 to 1 mux using 4 to 1 mux. These is a standard design, I am not explaining this. These are available in textbooks. Now actually in the description that I will show, we shall be instantiating the 4 to 1 mux, five times with the interconnections like this. The intermediate lines are t0, t1, t2, t3, ok. You see you have exactly that. So, this is the mux 16 to 1 description input, output and we have defined this wire t, t0, t1, t2, t3.

So, these five multiplexers, we have instantiated. Just check for m0, we have in 3 to 0 and select 1 to 0 output is t0. Let us see for m0, input is in 3 to 0 in, select are sel0 and sel1 and output is t0, ok. Same thing, similarly m1, m2, m3, and finally m4, the input is t, you see input is t, t0, t1, t2, t3, you can simply write t; it is a vector. The high order bits of sel 3 and 2 are the select lines and out. So, this is your structural representation of this 16 to 1 mux in terms of 4 to 1 mux, but this 4 to 1 mux, I am still describing in a behavioral fashion, 3 to 0 same statement 4 to 1 mux.

So, this you can check. I am not showing the waveform. If you modify your design like this, and if you run your simulation, you will be getting the same simulation output, same behavior as you have seen earlier for a full behavioral description of a 16 to 1 mux, ok. So, this is our version 2, which can be checked to work in the same way. Next, we proceed one step further. Here we have used 4 to 1 muxes right. Here we have used 4 to 1 muxes, and our 4 to 1 mux was behavioral.

(Refer Slide Time: 26:47)

**Version 3: Behavioral modeling of 2-to-1 MUX
Structural modeling of 4-to-1 MUX**

```
module mux2to1 (in, sel, out);
  input [1:0] in;
  input sel;
  output out;

  assign out = in[sel];
endmodule
```

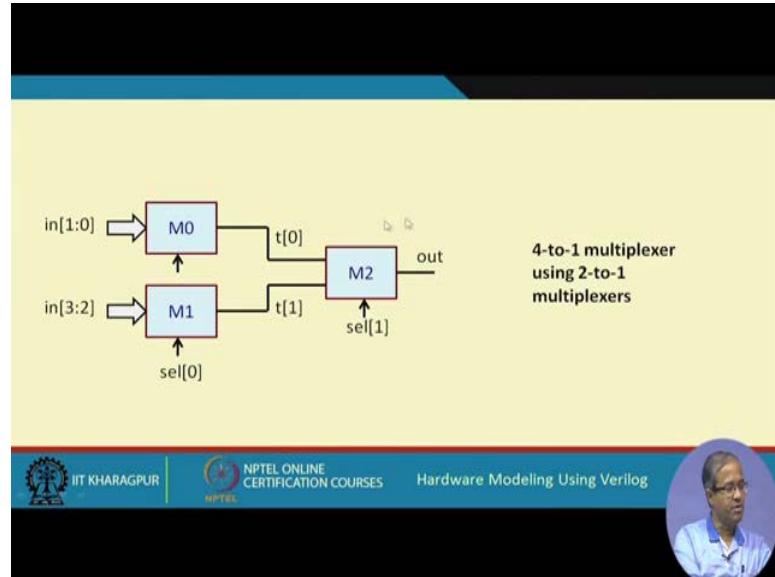
```
module mux4to1 (in, sel, out);
  input [3:0] in;
  input [1:0] sel;
  output out;
  wire [1:0] t;

  mux2to1 M0 (in[1:0],sel[0],t[0]);
  mux2to1 M1 (in[3:2],sel[0],t[1]);
  mux2to1 M2 (t,sel[1],out);
endmodule
```

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

Now we are refining the definition of 4 to 1 mux. We are implementing a 4 to 1 mux using 2 to 1 muxes, like this, using three 2 to 1 multiplexers you can implement a 4 to 1 multiplexer.

(Refer Slide Time: 26:54)



So, two and two inputs get here, the least significant, select line will be here, most significant select line will be here.

So, that way, same way, mux 4 to 1. I can use by instantiating three copies of mux 2 to 1. These correspondences you can check. But again, this mux 2 to 1, I am still defining in a behavioral fashion, 2 to 1 mux. I have just defined it like this, where select is a 1-bit and input is 2-bits, ok. So, I proceed like this. So, at the last step, I have the smallest multiplexer left with me, 2 to 1 this I can straight away implement it using structural fashion.

(Refer Slide Time: 27:49)

Version 4: Structural modeling of 2-to-1 MUX

```
module mux2to1 (in, sel, out);
  input [1:0] in;
  input sel;
  output out;
  wire t1, t2, t3;

  NOT G1 (t1,sel);
  AND G2 (t2,in[0],t1);
  AND G3 (t3,in[1],sel);
  OR G4 (out,t2,t3);
endmodule
```

Point to note:

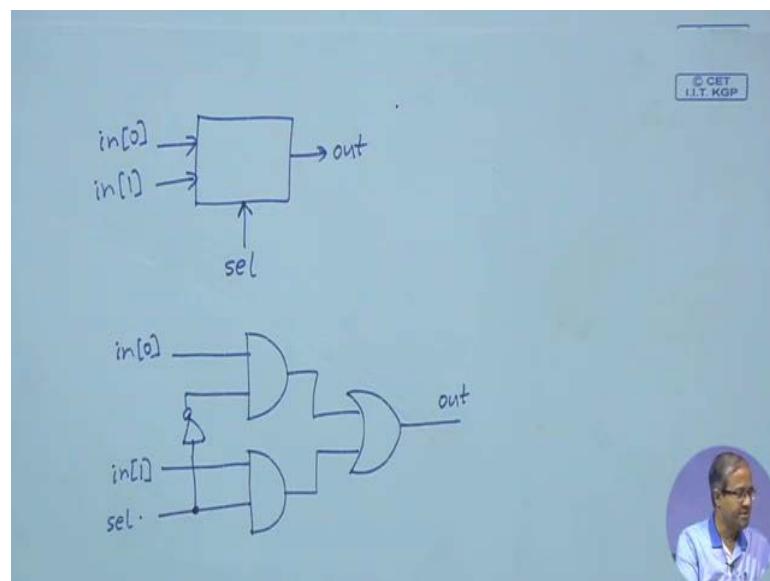
- Same test bench can be used for all the versions.
- The versions illustrate hierarchical refinement of design.

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



So, a multiplexer, how does, means how do we implement a 2 to 1 mux.

(Refer Slide Time: 28:02)



The are 2 inputs, 1 input, and a select line. Let us say, the inputs are $in[0]$ and $in[1]$, this is sel and the output is out . So, the way you can implement it is like this, you can take two AND gates and an OR gate, this $in[0]$ can be connected here, this $in[1]$ can be connected here, and this sel , you can connect directly here, and through an inverter here. So, when select is 0, this will be 1, this $in[0]$ will come out and when sel is 1 this will be 0, and this will be 1,

this in1 will be selected, in1 will be coming out, right. So, these are 4 gates. So, we specify this structural description for the multiplexer, this is what we have described here, a NOT gate, two AND gates and a OR gate.

So, whatever we have shown here is the typical design flow of a designer. So, when a designer goes about designing a complex digital system. Well I have shown a relatively very simple example so that you can appreciate, just a multiplexer, I have shown that we start with using a behavioral description.

Then step by step hierarchically, we break it up into structural representation at various levels, and when you are done. We have all the hierarchical descriptions all described in the structural level, and what we have the overall design is a complete structural design, ok. So, I strongly recommend that you should try out these examples yourself, run them on a simulator, view the waveforms on GTK wave or any other tool that you have accessed to, and you get a feel of this simulation and the feel of the design, ok.

So, we shall be continuing with some more examples in our next lectures again.

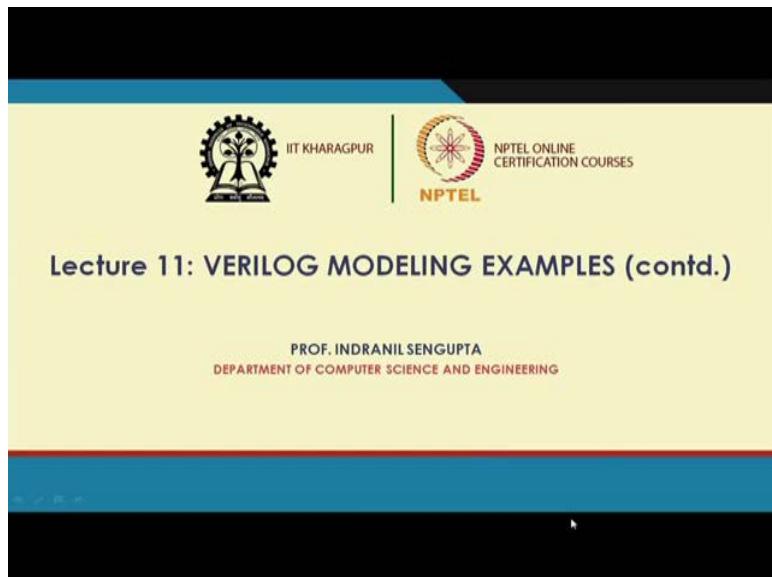
Thank you.

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

Lecture – 11
Verilog Modeling Examples (Contd.)

So, in this lecture, we shall be working out another example. See working with examples is good in two respects, one is that it can give you confidence in actually learning about the language and how you can approach the design of a thing; and secondly, you can actually understand the flow, the design flow how it is actually done in practice. So, the second example that we will be taking is a slightly more complex example than the multiplexer we took last time. This is the example of an adder and means an adder circuit which is part of an arithmetic logic unit - ALU.

(Refer Slide Time: 01:03)



So, this is our lecture topic.

(Refer Slide Time: 01:06)

Example 2

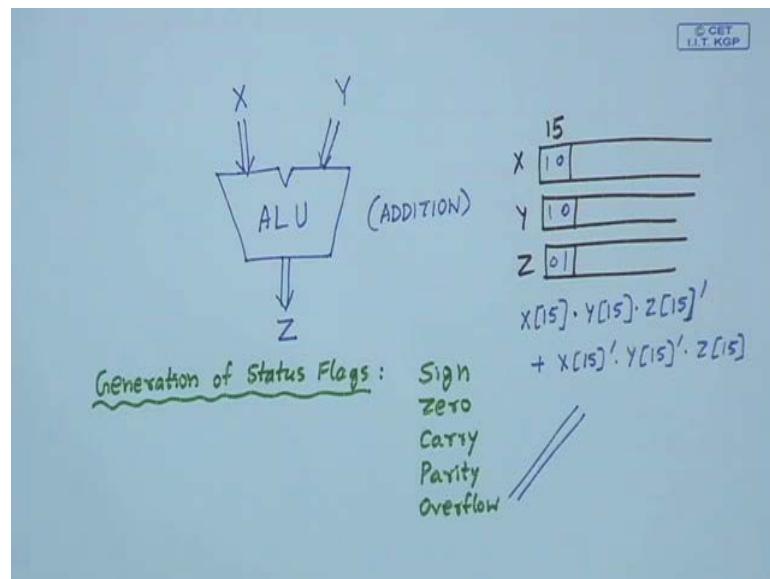
Version 1: Behavioral description of a 16-bit adder.

- Generation of status flags:
 - Sign : whether the sum is negative or positive
 - Zero : whether the sum is zero
 - Carry : whether there is a carry out of the last stage
 - Parity : whether the number of 1's in the sum is even or odd
 - Overflow : whether the sum cannot fit in 16 bits

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

So, we describe an adder; again we start with the behavioral description of an adder. Now, let us see that what we are trying to do here.

(Refer Slide Time: 01:26)



Now, this adder is just symbolic as an example we have taken, but in general let us call it these an arithmetic and logic unit, but in the code that I will show, will only talk about addition. So, what does the ALU do? the ALU will take two data as input, let us call them X and Y and it will be generating a result, let us call them as that Z as output. So, the example that I will be illustrating, I shall be only talking about addition. And when this

structure of course will be elaborating, an addition, we shall be looking at one more thing, what? we shall be looking at the process of generation of the status flags.

See in many processors, I mean, you may be aware of there has some status flags which are automatically set as a result of some arithmetic or logic operations. So, this status flag that we shall be looking are sign, zero, carry, parity and overflow. This status flags actually will give us some idea regarding the addition operation that have been taking place. This sign flag will tell whether the result is negative or positive. This zero flag will tell whether the result is zero or non-zero. The carry flag will tell whether there was a carry out of the addition. The parity flag will tell whether the number of ones in the result is odd or even; and overflow will tell that well the result means after addition, I have done something which is not fitting, there is an overflow, so whether there is an overflow or not. So, we shall be looking at the design of an adder with the generation of this five status flags, ok.

(Refer Slide Time: 04:06)

Example 2

Version 1: Behavioral description of a 16-bit adder.

- Generation of status flags:
 - Sign : whether the sum is negative or positive
 - Zero : whether the sum is zero
 - Carry : whether there is a carry out of the last stage
 - Parity : whether the number of 1's in the sum is even or odd
 - Overflow : whether the sum cannot fit in 16 bits

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

So, we start with the version one of our design which is the behavioral description of the adder. So, this five status flag I have already mentioned, ok, fine.

(Refer Slide Time: 04:19)

The screenshot shows a Verilog code editor with a yellow background. The code defines a module named ALU with parameters X, Y, and Z, and outputs Sign, Zero, Carry, Parity, and Overflow. It uses an assign statement to calculate the sum (Z) and carry (Carry) of X and Y. It also calculates the sign (Sign), zero (Zero), parity (Parity), and overflow (Overflow) flags based on the MSB of the result and the inputs.

```
module ALU (X, Y, Z, Sign, Zero, Carry, Parity, Overflow);
    input [15:0] X, Y;
    output [15:0] Z;
    output Sign, Zero, Carry, Parity, Overflow;

    assign {Carry, Z} = X + Y; // 16-bit addition
    assign Sign = Z[15];
    assign Zero = ~Z;
    assign Parity = ~^Z;
    assign Overflow = (X[15] & Y[15] & ~Z[15]) |
                      (~X[15] & ~Y[15] & Z[15]);
endmodule
```

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the text "Hardware Modeling Using Verilog".

Let us straight away look into the design. This is our behavioral design of our ALU or the adder. You see, what are the parameters the name of the module I have given as ALU, X, Y, Z; X and Y are the inputs. Let us assume they are 16-bit numbers 0 to 15, Z is the output. And these are the five flags – Sign, Zero, Carry, Parity and Overflow, these are all outputs. Now, the addition operation I am doing using a behavioral fashion, using the single statement – assign. Carry and Z using the concatenation operation, because Z will be the result. And the carry out of the addition that will go in to the carry flag, right, that is the Carry. So, I am generating carry right away like this, Carry and Z, this will be 16 plus 1, 17-bits equal to X plus Y.

Now, the other status flags, how are they generated? Well, Sign, well, numbers are represented in two's complement form typically. Now, in 2's complement representation, the most significant bit of the number indicates the Sign; if it is 1, it is negative; if it is 0, it is positive. So, the MSB of the result Z that will go into the sign flag straight away. So, we are straight away assigning Z15 to Sign, the most significant bit. Zero flag, what is zero flag? Zero flag will tell whether the result is zero or not, which means Z the 16-bit sum, it is zero or not, zero means all the bits are zero. If it is zero, this zero flag will be set, it will be 1. So, what kind of operation do you require if all the bits of Z are 0, I have to set Z flag to 1; zero flag to 1. So, I need a NOR operation. I take OR of all the zeros, if the inputs are all zero, output of OR will be 1, then a NOT, it will be the OR will be zero, then NOT, it

will 1, so NOR will be 1, ok. So, I need a Reduction Operator NOR, just I write assign Zero equal to reduction NOR on Z, that will be 0.

And parity is just exclusive NOR, if it is even parity it will be 1; if it is odd parity it is 1, it is 0. So, simply write assign parity equal to again reduction operator exclusive NOR of Z. So, generation of Sign, Zero and Parity are very simple, ok. Now, Overflow, there are various ways to detect Overflow. The way we are implementing Overflow is like this. Let us say we have a number X and Y, this is our bit number 15 or most significant bit, and this is the sum Z. So, here also we have bit number 15. Now, here we are saying that there will be Overflow if the condition that has to be followed is this. I am just writing down, then I am explaining. This means that either both X15 and Y15 were 1 and 1, this is a dot means AND operation, plus means OR operation. X15, Y15, Z15 bar, which means, this is 1, this is 1, and this is 0.

So, the numbers were negative, after addition the number has become positive, which is never possible, this is possible only when Overflow has taken place or the reverse condition. This was 0, this was 0, but in the sum, it has become 1. The numbers are positive, but this sum is showing us negative. So, if we implement this logic that will give you the Overflow straight away, right. So, here we have done exactly that we have used this same expression to generate the Overflow, right. So, this is the behavioral description of the adder.

(Refer Slide Time: 09:37)

The screenshot shows a Verilog testbench script named 'alu.vcd'. The code defines a module 'alutest' with inputs X, Y, and Z, and outputs S, ZR, CY, P, V. It uses an ALU DUT block. The initial block contains a monitor statement to dump variables (\$monitor) and several stimulus cases using the #5 keyword. The monitor statement includes variables \$time, X, Y, Z, S, ZR, CY, P, V, and their respective types (%h, %b). The stimulus cases involve setting X and Y to various hex values (e.g., 16'h8fff, 16'h8000, 16'hffff, 16'h0002, 16'hAAAA, 16'h5555) and then finishing the simulation.

```
module alutest;
reg [15:0] X, Y;
wire [15:0] Z;      wire S, ZR, CY, P, V;
ALU DUT (X, Y, Z, S, ZR, CY, P, V);
initial
begin
$dumpfile ("alu.vcd"); $dumpvars (0,alutest);
$monitor ($time,"X=%h, Y=%h, Z=%h, S=%b, Z=%b, CY=%b, P=%b,
V=%b", X, Y, Z, S, ZR, CY, P, V);
#5 X = 16'h8fff; Y = 16'h8000;
#5 X = 16'hffff; Y = 16'h0002;
#5 X = 16'hAAAA; Y = 16'h5555;
#5 $finish;
end
endmodule
```

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

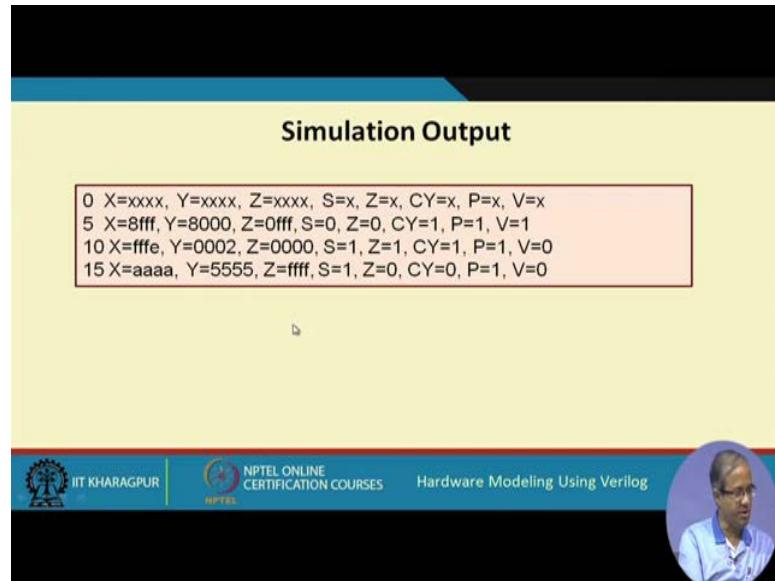
Now, just like in the example we showed earlier, we have also written a test bench here. So, we called it as alutest. So, we have instantiated this module called ALU these the module ALU. So, I have given this name DUT and the parameters are given. So, again in the initial block, we have given dumbfile, dumb variables. We have monitored the time X, Y, Z and all these status flags. So, here we instantiated the status flag names we have given as S, O, ZR CY, P and V in the same order, same order, we have given the variable names, same order. So, this names you can change, same names are not required, ok. And the input data that you applied are you can see with delay of 5, 5, 5, we have given. In the first step, we have given the first number X equal to 16-bit hexadecimal 8FFF and 8000.

(Refer Slide Time: 10:56)

5:	$X = 8FFF$	}	$Z = 0FFF$
	$y = 8000$		
10:	$X = FFFE$	}	$Z = 0000$
	$y = 0002$		
15:	$X = AAAA$	}	$Z = FFFF$
	$y = 5555$		

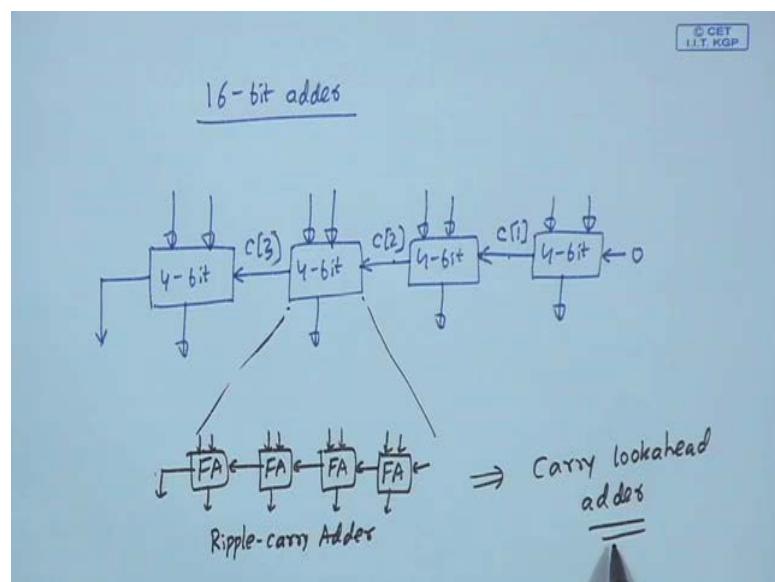
So, let us a just note down this numbers at time 5, we have applied X equal to 8FFF and Y equal to 8000. Then again after a delay of 5, we are applying X equal to FFFE and Y equal to 0002. So, after delay of 5 means, at time 10, we are applying X equal to FFFE, Y equal to 0002. And again after delay of 5, we are applying X is AAAA and Y is 5555. So at time 15, X is AAAA, Y is 555 and 5, right. So, let us see just by normal addition what will be the sum? If I just add 8FFF i.e X and Y equal to 8000, Z will be F and 0 is F, F and 0 is F, F and 0 is F, and 8 and 8 will be 0 and there will be a carry out. And here E and 2, if you add it will be 0, there will be a carry. So, 1 and F will be 0, carry, 1 and F will be 0, carry, 1 and F will be 0, it will be all 0, and there will be carry out again. And here A, if you add A and 5 is F, B, C, D, E, F, so FFFF, right. So, this should be the sum and the of course, the flags will be set.

(Refer Slide Time: 12:57)



Let us see. So, if you just Simulate, the simulation output is showing this, at time 5, 0FFF, 0000, FFFF. So, this is exactly what we got here 0FFF, 0000, FFFF. And in the all cases, you can verify the Sign, 0FFF, the Sign is positive; 00 this is also, so here you can all check the Signs here, Zero flag, Carry flag, Parity flag and the Overflow flag. You can check all of them, they will be consistent, ok. So, this is our simulation output with respect to the description which you have given here, right, fine. So, it works correctly.

(Refer Slide Time: 14:19)



Now, what we are trying to do, here we have just given a behavioral description of a full adder of an adder. Now this you want to refine in to more detailed description. So, the first step what we do? the first step what we do is we had started with the design of a 16-bit adder, right. So, what you are saying will be using 4-bit adders. Let us take four 4-bit adders, so they will be all fed with two 4-bit numbers then we generating a sum. So, the carry in for the first stage will be 0, and this carry out will go into the carry in of this, this carry out will go into the carry in of this, this carry out will go into the carry in and this will be the final carry out.

So, let us do a structural design like this. So, now, we are, this implementing 16-bit adder using four 4-bit adders, and connecting them like this, ripple-carry between the stages, right. So, this is our next step. So, well, this what you got, this with respect to the simulation output, we have also shown it here. Here I think there is a small typographical error, this should be S equal to 0, fine, S equal to 0. See here, S is 0 and it goes 1 here; it goes 1 here, lasted, ok, fine. So, in the waveform also you can see the same thing. So, this is X, this is Y, sum is Z. You can see the value 0FFF, you can see the value 0000, FFFF. You can see the values right, and you can also see the flag value zero or one, whatever it is coming, fine, ok.

(Refer Slide Time: 16:30)

Version 2: Structural description of 16-bit adder using 4-bit adder blocks (with ripple carry between blocks).

```
module ALU (X, Y, Z, Sign, Zero, Carry, Parity, Overflow);
    input [15:0] X, Y;
    output [15:0] Z;
    output Sign, Zero, Carry, Parity, Overflow;
    wire c[3:1];

    assign Sign = Z[15];
    assign Zero = ~|Z;
    assign Parity = ~^Z;
    assign Overflow = (X[15] & Y[15] & ~Z[15]) |
                      (~X[15] & ~Y[15] & Z[15]);

```

.. Contd.

Now, in this second version as a I said, we are using 4-bit adders with ripple-carry between blocks. Now, our let this ALU description, module description, the first part remains the

same, this was exactly identical with the previous one. What you have done? we have just added a wire. So, why you need this wire? we need this wire to just implement this intermediate carries, this we call as c(1), this we call as c(2), this we call as c(3), this intermediate carries. So, we are defining them as wires, wires c[3:1].

(Refer Slide Time: 17:17)

```

adder4 A0 (Z[3:0], c[1], X[3:0], Y[3:0], 1'b0);
adder4 A1 (Z[7:4], c[2], X[7:4], Y[7:4], c[1]);
adder4 A2 (Z[11:8], c[3], X[11:8], Y[11:8], c[2]);
adder4 A3 (Z[15:12], Carry, X[15:12], Y[15:12], c[3]);
endmodule

Behavioral description of a 4-bit adder

module adder4 (S, cout, A, B, cin);
    input [3:0] A, B;    input cin;
    output [3:0] S;    output cout;
    assign {cout,S} = A + B + cin;
endmodule

```

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

And we have just instantiated four adders like that, ok. Just see exactly like that way here for the first adder the inputs are X , 0 to 3 and Y, 0 to 3; for the second adder inputs are X 4 to 7, 4 to 7; next one 8 to 11, and last one 12 to 15, ok. So, for the first adder the carryin is zero, second adder carryin is c1 and for the first adder carryout is c1; for the second adder this c1 is the carryin. Here the carryout is c2, for the next adder c2 is the carryin, c3 is the carryout, for the next one c3 is the carryin, and the final carryout is carry, ok. This is how the carry is generated. Because here we have not shown the carry, carry will be generated like that Sign, Zero, Parity and Overflow are shown, ok. Carries coming here and these are the sum. But the 4-bits adders still will leave it in a behavioral description like this, this adder four is the 4-bit adder, this was still defining like this. So, this design if you simulate, you will see that you are still getting the same simulation result, same simulation output, this you can verify, all right, fine.

(Refer Slide Time: 18:48)

Version 3: Structural Modeling of Ripple Carry Adder

```
module adder4 (S, cout, A, B, cin);
  input [3:0] A, B;  input cin;
  output [3:0] S;   output cout;
  wire c1,c2,c3;

  fulladder FA0 (S[0],c1,A[0],B[0],cin);
  fulladder FA1 (S[1],c2,A[1],B[1],c1);
  fulladder FA2 (S[2],c3,A[2],B[2],c2);
  fulladder FA3 (S[3],cout,A[3],B[3],c3);

endmodule
```

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

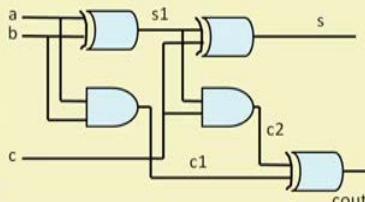


So, the next stage is that we are actually modelling these 4-bit adders as a ripple-carry adder.

(Refer Slide Time: 18:58)

```
module fulladder (s, cout, a, b, c);
  input a, b, c;
  output s, cout;
  wire s1,c1,c2;

  xor G1 (s1,a,b), G2 (s,s1,c),
  G3 (cout,c2,c1);
  and G4 (c1,a,b), G5 (c2,s1,c);
endmodule
```



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



See here in the earlier design, we have just use this 4-bit, just 4-bit adder modules, right. Now, what I am doing is that you see, just look at this diagram. This 4-bit adder modules, so each of this 4-bit adder modules, we are expanding, we are using four full adders. We are using four full adders and using four full adders, we are using a ripple-carry adder. So, the carry output of one will be going in to the carry input of the next, these will be the

inputs, ok and some output, these are standard designs of adder. So, we are actually going for a ripple-carry adder of 4-bits which will be looking like these, four full adders in cascade. So, this is exactly what you have done. This adder4 module which earlier was looking like this behavioral, this we are making it structural now.

We are implementing adder4 by instantiating full adder four times and interconnecting them in a suitable way. Like the first one is getting A0, B0, cin is the carryin, c1, c1 is here, c2, c2 is here, c3, c3 is here; and final carryout is cout, cout, ok. So, this version is also a very standard way. So, you see, if we have this refinement, and if you do a simulation again, you will again see that you will be getting this same simulation result. So, I strongly suggest that you should actually go along with this class not only listening to what I am saying, but also actual implementing and simulating the codes and seeing that whether it is working correctly or not. So, if you do this you will see that still, you will be getting the same result which means that so far our design is correct, so far so good, ok.

Now, these full adders at the end, these full adders also you can implement using structural way because you see, this is one implementation of a full adder which is a compact implementation which requires a total of five gates - three XOR gates, two AND gates. There are many implementations of full adder; this is one possible implementation. So, here this structural implementation that I am showing here of full adder uses this netlist. So, you see this s, cout, a, b, c are the input and output lines; a, b, c is the input, s and cout are the output and the intermediate wires are s1, c1 and c2.

So, there are three XOR gates. So, we are instantiating the gates like this XOR G1(s1, a, b), it means this is G1, s1 is the output, a, b are the inputs. G2(s1, c), this is G2, s, s1, c. Last one G3(cout, c2, c1), cout, G3, cout, c2, c1. And AND, there are two, G4(c1, a, b), this G4, c1, a and b, G5, this is G5, c2, this is s1, this is c. So, you see when you instantiate gates in a structural way, you can group the similar gate types together, like the three XOR gates instead of writing XOR three times. I have written XOR once and just have separated them by commas, this way you can write.

So, you see now, we have arrived at a complete structural description of a 16-bit adder, where the individual 4-bit adders are also ripple-carry. Now, let us also look at an improved version of this design, where this 4-bit adder that I have shown. So, instead of mapping this in to a ripple-carry adder like this, so we can map it as an alternative way into a carry

look-ahead adder. So, carry look-ahead adder is a faster way to implement an adder. See in a ripple-carry adder, the carry is rippling through the stages. So, the worst case delay will be the maximum time it takes for the carry to ripple through, but in the carry look-ahead adder, we are generating all the carries in parallel, so that the addition becomes very fast.

(Refer Slide Time: 24:39)

```

module adder4 (S, cout, A, B, cin);
    input [3:0] A, B;    input cin;
    output [3:0] S;     output cout;
    wire p0, g0, p1, g1, p2, g2, p3, g3;
    wire c1, c2, c3;

    assign p0 = A[0] ^ B[0],   p1 = A[1] ^ B[1],
           p2 = A[2] ^ B[2],   p3 = A[3] ^ B[3];

    assign g0 = A[0] & B[0],   g1 = A[1] & B[1],
           g2 = A[2] & B[2],   g3 = A[3] & B[3];

```

Contd...

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

So, in the next design, here we shall or here we are looking at a design of a 4-bit adder, adder4, using carry look-ahead principle. Now, here let us skip this.

(Refer Slide Time: 24:55)

```

assign c1 = g0 | (p0 & cin),
       c2 = g1 | (p1 & g0) | (p1 & p0 & cin),
       c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & cin),
       cout = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
              (p3 & p2 & p1 & p0 & cin);

assign S[0] = p0 ^ cin,
       S[1] = p1 ^ c1,
       S[2] = p2 ^ c2,
       S[3] = p3 ^ c3;

endmodule

```

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

We shall come back here before the explaining this description.

(Refer Slide Time: 24:57)

How does a Carry Look-ahead Adder work?

- The propagation delay of an n-bit ripple carry order is proportional to n.
 - Due to the rippling effect of carry sequentially from one stage to the next.
- One possible way to speedup the addition.
 - Generate the carry signals for the various stages in parallel.
 - Time complexity reduces from $O(n)$ to $O(1)$.
 - Hardware complexity increases rapidly with n.

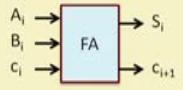
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog



Let us first look at how a carry look-ahead adder works, ok, because this will be required to understand how we have come up with these expressions, because these expressions may look a little complicated, ok. So, you see, for a ripple-carry adder that means, as you have seen, the total propagation delay will be proportional to the number of stages n, because as I had said the carry ripples through from one stage to the other. So, carry look-ahead adder is one way to speed up the addition, where the carry signals are generated in parallel for all the various stages. So, effectively the addition time reduces from order n in ripple-carry adder to order one, which is a constant time in carry look-ahead adder, but the flip side is, the drawback is the hardware complexity also increases very rapidly with a number of bits n.

(Refer Slide Time: 26:06)

- Consider the i-th stage in the addition process.
- We define the *carry generate* and *carry propagate* functions as:
 $g_i = A_i \cdot B_i$
 $p_i = A_i \oplus B_i$
- $g_i = 1$ represents the condition when a carry is generated in stage-i independent of the other stages.
- $p_i = 1$ represents the condition when an input carry C_i will be propagated to the output carry C_{i+1} .


$$C_{i+1} = g_i + p_i \cdot C_i$$



IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

Let us see how. Well, if we look at a single full adder. So, in a ripple-carry adder, you look at the i-th stage full adder, where the inputs are A_i , B_i and C_i , the output is a sum S_i and a carry $S_i + 1$. Now, we are defining two signals or two, you can say two functions called carry generate and carry propagate. Well, carry generate specifies the condition where a carry is generated it in the full adder irrespective of the carryin. The condition is whenever both the data inputs A_i and B_i are 1, then carry will be generated irrespective of C_i . So, that is g_i , g_i is $A_i \cdot B_i$ and propagate, carry propagate represents the condition, it says that when the input carry will be propagating to the output. What is that condition? The condition is means among A and B , at least one of them means, exactly one of them should be one, and the other should be zero. Suppose it is 0 and 1, then if there is a carry coming 0, 1, 1, that will generate a carry, the carry will be propagating. So, this is the condition for carry propagation, exactly one of A_i and B_i should be 1.

So, this carry propagation is defined by the exclusive OR function. This is generate and propagate. So, g_i and p_i , you can implement by NAND gate or an XOR gate, right. Now, with this you can write that the output carry expression, output carry will be given by well, either the carry is generated or the propagate condition is true and there is an input carry. So, C_{i+1} , you can write down as by an expression $g_i \text{ OR } p_i \cdot C_i$, ok. Now, this you can use recursively to find out an expression.

(Refer Slide Time: 28:25)

Unrolling the Recurrence

$$\begin{aligned}c_{i+1} &= g_i + p_i c_i = g_i + p_i(g_{i-1} + p_{i-1}c_{i-1}) = g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \\&= g_i + p_i g_{i-1} + p_i p_{i-1}(g_{i-2} + p_{i-2}c_{i-2}) \\&= g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + p_i p_{i-1} p_{i-2} c_{i-2} = \dots\end{aligned}$$

$$c_{i+1} = g_i + \sum_{k=0}^{i-1} g_k \prod_{j=k+1}^i p_j + c_0 \prod_{j=0}^i p_j$$



IIT KHARAGPUR



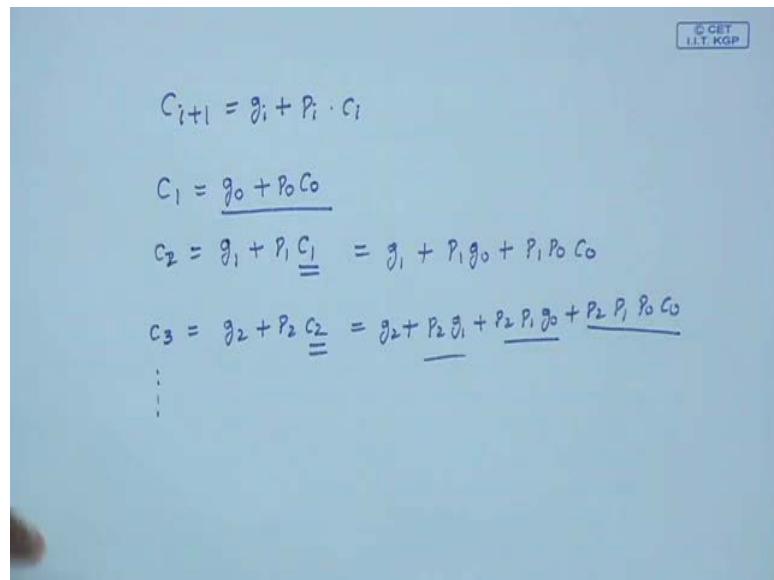
NPTEL ONLINE
CERTIFICATION COURSES

Hardware Modeling Using Verilog



c_{i+1} equal to g_i plus $p_i c_i$. So, you can write this c_i , you can write as g_{i-1} plus $p_{i-1} c_{i-1}$, you can expand it like this. This c_{i-1} again you can write as g_{i-2} plus $p_{i-2} c_{i-2}$, like this you can recursively go on expanding. So, I am not showing you the whole thing. See, basically what it means, I am just writing down some simple expressions.

(Refer Slide Time: 29:00)



© CET
I.I.T. KGP

$$\begin{aligned}C_{i+1} &= g_i + p_i \cdot C_i \\C_1 &= \underline{\underline{g_0 + p_0 c_0}} \\C_2 &= \underline{\underline{g_1 + p_1 C_1}} = \underline{\underline{g_1 + p_1 g_0 + p_1 p_0 c_0}} \\C_3 &= \underline{\underline{g_2 + p_2 C_2}} = \underline{\underline{g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0}} \\&\vdots\end{aligned}$$

See, what you got is C_{i+1} equal to g_i plus $p_i c_i$. So, you can generate the carry for the first stage C_1 as you just substitute i equal to 0, g_0 plus $p_0 c_0$. So, what will be C_1 ? this C_1, C_2, C_3

will be again you put g, put i equal to 2, and i equal to 1, so g_1 plus p_1c_1 , right. Now, c_1 , you have already got this. So, you can substitute c_1 here. So, this becomes g_1 plus p_1 multiplied by this, so p_1g_0 plus $p_1p_0c_0$. You go to c_3 , c_3 will be g_2 plus p_2c_2 . Now, c_2 you have already got this, so this will be g_2 plus p_2 multiplied by this, multiply this p_2g_1 plus $p_2p_1g_0$ plus $p_2p_1p_0c_0$ and so on, like this we will be getting this expression. So, see all the carries you can get just by a two level AND, OR kind of expression. There is no question of carries rippling through, but as you go on this expression become more complex, bigger expressions, more number of gates, ok, this is what I was saying.

(Refer Slide Time: 30:44)

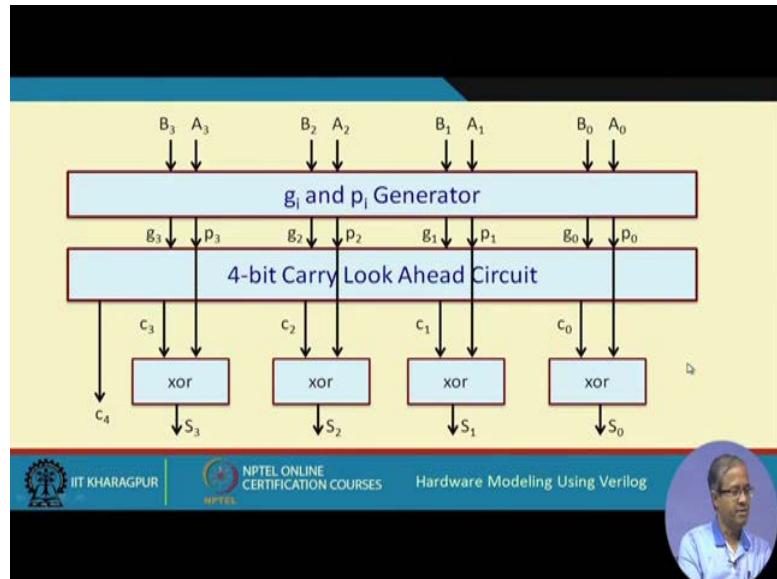
Generation of the Carry and Sum bits

$C_4 = g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 + c_0p_0p_1p_2p_3$ $C_3 = g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2$ $C_2 = g_1 + g_0p_1 + c_0p_0p_1$ $C_1 = g_0 + c_0p_0$	4 AND2 gates 3 AND3 gates 2 AND4 gates 1 AND5 gate 1 OR2, 1 OR3, 1 OR4 and 1 OR5 gate
$S_0 = A_0 \oplus B_0 \oplus c_0 = p_0 \oplus c_0$ $S_1 = p_1 \oplus c_1$ $S_2 = p_2 \oplus c_2$ $S_3 = p_3 \oplus c_3$	4 XOR2 gates

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

So, actually this is what I have just now showed c_1 is this, c_2 is this, c_3 and c_4 . So, you require so many gates. And sum, you can directly generate, you see, p is nothing but A and B's XOR. So, the sum is the XOR of all the three bits. So, you can simply write p_0 XOR c_0 . Similarly, S_1 is p_1 XOR c_1 and so on, you need four XOR gates.

(Refer Slide Time: 31:16)



So, in a carry look-ahead adder, in a yeah, carry look-ahead you have a circuit first which generates all the g and p 's which means XOR and AND gates. Then we have a carry look-ahead circuit which implements these functions c_1, c_2, c_3, c_4 , ok, using AND, OR circuits. So, it generates c_0, c_1, c_2, c_3 and also c_4 . So, now, you can directly generate this sums by using XORs, XOR the carry with the p 's. So, the total time is constant irrespective of the number, there is no ripple of the carry, ok. So, this will be much faster.

So, just remember these expressions and the sum expressions. So, if you remember these expressions, you can just correlate that what you have done is exactly that. Here the carry propagate and generate signals, I have defined as a wires in between and the intermediate carries also. So, p_0 is $A_0 \text{ XOR } B_0$; p_1 is $A_1 \text{ XOR } B_1$ and so on. Similarly, the generate signals AND, $A_0 \text{ AND } B_0, A_1 \text{ AND } B_1$ and so on. So, I generate the propagate signals, I generate the carry generate signals then I generate all the carry signals, c_1 equal to $g_0 \text{ OR } p_0 \text{ cin}$, c_0 is cin ; c_2 is $g_1 + p_1 g_0 + p_1 p_0 \text{ cin}$ exactly we have written down that expression. c_2 equal to $g_1 \text{ OR } p_1 g_0 \text{ OR } p_1 p_0 \text{ cin}$, c_3 and c_4 is finally, cout. So, as you can see as the number of bits increases these expressions are becoming more complicated and finally, sum is the XORs. So, this is a carry look-ahead adder.

So, in the same way as I had said, you can replace a module by a better or an improved module. You replace a module by another module like, for example, we replaced a 4-bit ripple-carry adder by a 4-bit carry look-ahead adder, we have got a faster adder, but we can do simulation and you can find out that functionally our design is correct. It is working correctly. So, this actually completes our discussion for today's lecture. See what I try to illustrate over the last two examples, we discuss is that when we look at non trivial designs slightly bigger designs, we often break the design up in a hierarchical fashion. Whatever something was discussed in a or described in a behavioral fashion at one level, we do a some kind of iterative refinement. We break that behavioral description into structure description, try to be more detailed and we repeat the process until we reach a point where our entire design becomes structural.

So, in a normal digital system design, you will see later that well we do not only have combination circuits, we also have sequential circuits flip-flops, state machines and so on. So, we shall see later that when we have such a system where we have combination circuits, registers, and also lot of finite state machines, there it may not be worthwhile to convert all the modules, all the descriptions into structural. Well, the data path, the circuits where the actual calculations or computations have carried out, those of course, we can convert into structural fashion. But the controller, the circuit which implements a finite set machine, which generates the control signals often that we do not convert into a structural fashion, because it involves too much of a work on the part of the designer. So, we shall just learn these things slowly over the next lectures.

Thank you.

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

Lecture - 12
Verilog Description Styles

In the lectures that we have seen so far, we have looked at a number of examples where we took some modeling instances and showed, how we can create the description using Verilog. Now in this lecture, we shall be starting our discussion on the various ways in which we can model or create the description of a system or a digital block that we are trying to design.

So, the topic of a lecture is Verilog description styles, ok.

(Refer Slide Time: 01:05)

The slide has a yellow background with a black header bar. The title 'Description Styles in Verilog' is centered in bold black font. Below it is a bulleted list:

- Two different styles of description:
 - 1. Data Flow
 - Continuous assignment

Using assignment statements.
 - 2. Behavioral
 - Procedural assignment
 - Blocking
 - Non-blocking

Using procedural statements similar to a program in high-level language.

So, broadly speaking, there are two different styles of description; I have already given you some examples of these. So, you have seen some examples, but broadly speaking the description styles can be categorized into two different types, ok, the first one is the so-called data flow description style, which uses continuous assignments using the assign statement that you have already seen in many examples. So, this uses such kind of assignment statements using assign. Now as an alternative, you can have Behavioral design or description styles as well where you use some kind of procedural statements which are somewhat similar to a program in a high level language that you sometimes

write. You see whenever you write a program in a high level language like C or any such language, you try to specify what exactly you are trying to do, the different steps, ok.

So, this is what you or this is how you specify what you are trying to do. So, in that sense, the behavioral model is somewhat similar. So, you use some kind of procedural statements, which is similar to the construct, which are available in a high level language. These will see and here in place of continuous assignment, we shall be using a different kind of assignment statement called procedural assignment, which again is of two different types; one is called blocking assignment, other is called non blocking assignment.

So, over the next few lectures, we shall be discussing these different alternatives and variations and see that which one to use, when and why, fine.

(Refer Slide Time: 03:08)

Data Flow Style: Continuous Assignment

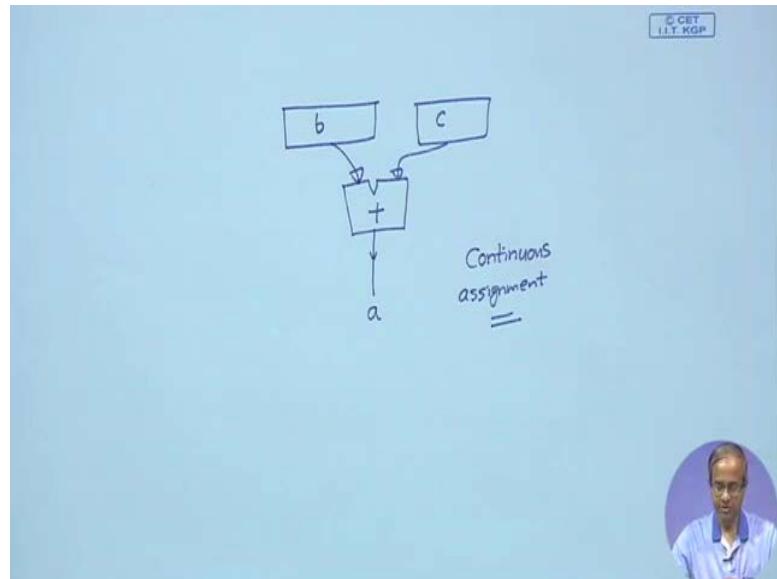
- Identified by the keyword "assign".
- Forms a static binding between:
 - The "net" being assigned on the left-hand side (LHS).
 - The expression on the right-hand side (RHS), which may consist of both "net" and "register" type variables.
- The assignment is continuously active:
 - Almost exclusively used to model combinational circuits.
 - We shall also see some examples of modeling sequential circuit elements.

assign a = b + c;
assign sign = Z[15];

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

So, let us start with continuous assignment or the data flow style which have already seen in sufficient detail, but just for the sake of completeness, let us also look at it once more. So, the data flow design style is identified by the keyword assign. So, whenever you want to make some assignment like, a equal to b plus c or sign equal to Z[15], bit number 15 of a vector Z. We use this keyword assign before that. Now, I mentioned earlier also whenever we use an assign, this forms a static binding, it is not that whenever this statement is executed, b plus is computed and the value is stored in a, not like that whenever you write a equal to b plus c, something like this will be happening.

(Refer Slide Time: 04:02)



Let say b is a register, c is another register. So, there will be an adder which will be generated, which will be adding b and c and the result will be generated or stored in a net which you are calling a. This a is not a register just a net. So, the whenever either b or c will change, this a will change immediately after the delay of this adder, ok. So, there is no concept of any clock or anything such an assignment we said earlier also is referred to as continuous assignment, right, ok.

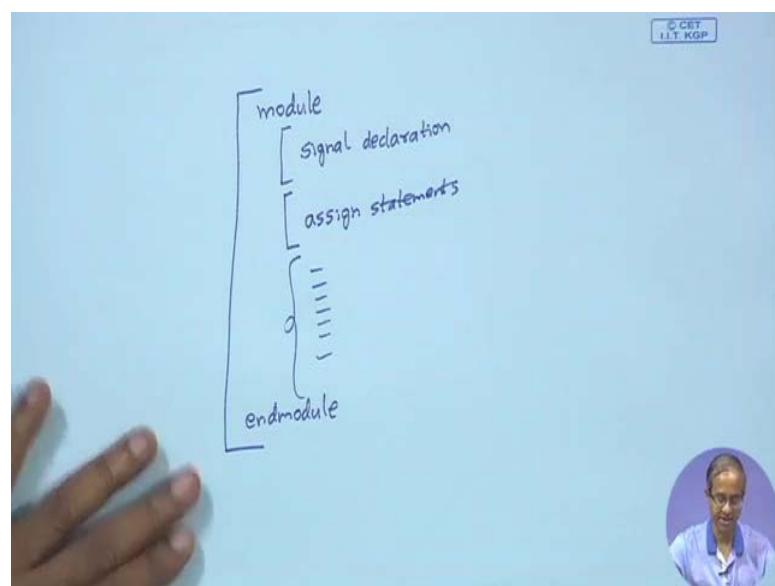
So, there are some constraints in such assignment statements using assign, first thing is that the left hand side must be a net type variable typically a wire, but however, the expression that we are using on the right hand side, here b plus c, here Z[15]. These expressions may consist of variables, which can be either net or register or any combination of the two and this is called continuous assignment because this is continuously active. There is no signal that tells you that when this operation will be carried out and when the output will be stored in that target. So, it is not like that it is some kind of continuous assignment, whenever one of the input changes, the output net value will be changing or be affected right away after the necessary delay of the circuit, ok, fine.

So, these kind of assign statements typically, we use it for modeling combinational circuits, but we shall see that we can also use this for modeling sequential circuit elements which are of course, not very common, but we can also do this. We shall be taking some examples later, right.

(Refer Slide Time: 06:18)

- Some points to note:
 - A Verilog module can contain any number of "assign" statements.
 - Typically, the "assign" statements are followed by procedural descriptions.
 - The "assign" statements are used to model behavioral descriptions.
- We shall illustrate various usages of "assign" statements for modeling combinational and also some sequential logic blocks.

(Refer Slide Time: 06:36)



Ok, now in a Verilog module, we can have any number of assignment statements, assign statements and this is a very typical structure. So, whenever we write a Verilog module, let us say; this is a Verilog module, it starts with the keyword module, it ends with endmodule.

So, in the beginning, we typically start with the signal declarations, all the net and the register type variables are declare there, then after that we usually club all the assign statements together. This is the normal practice and then the other kind of statements will

come, which you shall see later, right. This is how a typical Verilog module structure look like where the assign statements are typically placed in the beginning, right after the signal declarations. And another thing I also mentioned that in the assign statement, we just tell or say what we are trying to do. Suppose, you write a equal to b plus c, we are just saying that we want to add b plus, b and c and we want to store the result back in a, but we are not telling, what kind of adder to use, how to add and so on. So, this is more like a behavioral description, ok, we are specifying the behavior, but not the exact structure of the circuit.

So, now we shall see a number of examples; typical usages and while we are going through the examples, we shall also learn about a number of things there.

(Refer Slide Time: 08:20)

```
module generate_MUX (data, select, out);
    input [15:0] data;
    input [3:0] select;
    output out;
    assign out = data[select];
endmodule
```

Non-constant index in expression on RHS generates a MUX

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

Fine, this is one very simple example; we saw an example of multiplexes earlier. So, here we are generating a MUX, like, whenever we have a module description like this. Now see, now right here, we are not talking exactly about simulation, we are talking about what will happen? When? let us say, we are using a synthesis tool to synthesize our circuit. So, for that purpose, what will happen? how this synthesis tool will be carrying out synthesis and what kind of hardware blocks or modules will be generated.

So, let us see his example. So, in this example, you see that we are using a module called `generate_MUX`, which has three parameters, the input is a 16-bit quantity, it is a 16 to 1 MUX, 4-bit select line and a single bit output. So, we are simply using a single line

description, which says assign out equal to data of select. So, data is like a vector, 16-bit vector, select is a number, 4-bit number, which selects one of the bits and that particular bit goes to out. So, which is what MUX is?

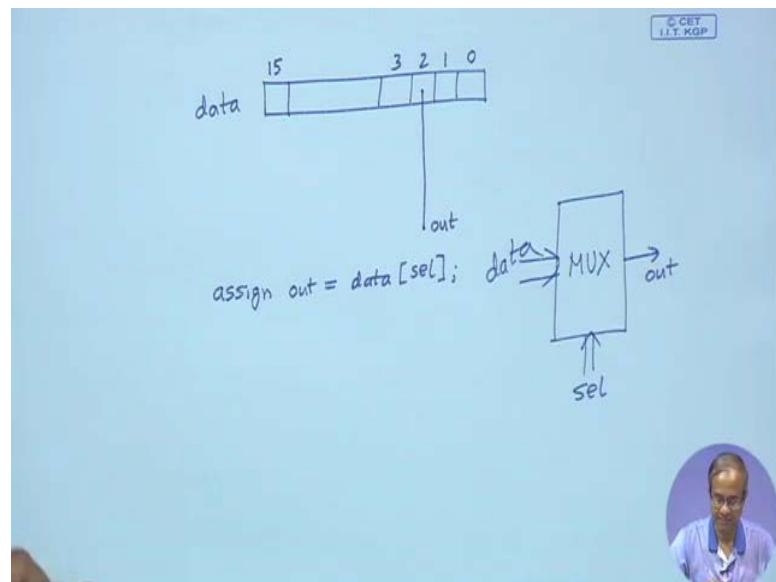
Now, whenever the Verilog synthesizer encounters such a description, a multiplexer will be generated. So, how does this synthesizer know that a multiplexer will be required to be generated? the keyword is a non constant index in the expression on the right hand side. So, whenever there is a vector which I am using with the non constant index, select is a variable, right, it is not a constant. So, the index value is a variable, then we will be generating a multiplexer, ok. So, the data whatever is there that will be the input to the multiplexer and this variable will go to the select line and this out will be the output point.

(Refer Slide Time: 10:40)

- Point to note:
 - Whenever there is an array reference on the RHS with a variable index, a MUX is generated by the synthesis tool.
 - If the index is a constant, just a wire will be generated.
Example: `assign out = data[2];`

Now, the point to note is that like as I had said with example, I have shown that whenever there is an array reference with the variable index, the multiplexer will be generated. But, however, if on the right side, it is not a variable, it is a constant, let us say like this, assign out equal to data[2], then, no multiplex will be generated because you see data is nothing but an vector. Vector means a collection of bits.

(Refer Slide Time: 11:17)



So, it is a 16-bit vector. So, the index is, the index values will go from 0 up to 15.

So, when we say assign out equal to data[2], what will happen is the data[2] is this bit, this bit will straight away be taken out and this will be called as out. So, no circuit will be generated, it will be just wiring. This particular bit will be connected or it will be given a name out, ok. This will be what will happen if I give an assignment like this; no gates or no hardware blocks will be generated for an assignment like this, but if I write like in the previous example, assign out equal to data, but I give a variable SEL here, I do not know which bit I want. So, here you need to synthesize or generate a multiplexer, where the input will be your data, your, this will be your select lines and on the output side you will be getting the output bit out, right, ok.

(Refer Slide Time: 12:38)

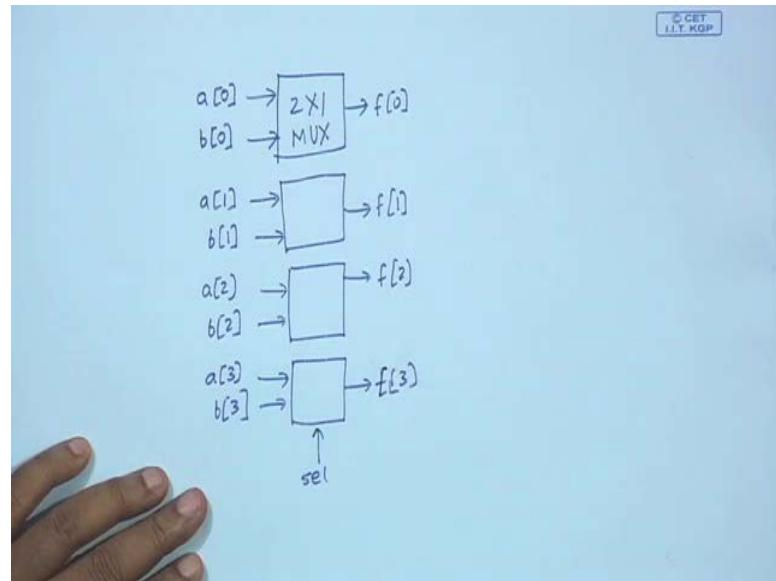
```
module generate_set_of_MUX (a, b, f, sel);
    input [0:3] a, b;
    input sel;
    output [0:3] f;
    assign f = sel ? a : b;
endmodule
```

Conditional operator generates a MUX

So, let us take another example. Here, this is also an example for multiplexer, but we are using a conditional operator here. So, what you are doing? Let us try to understand this a and b are two 4-bit vectors. So, the index goes from 0 to 3 in this order, we have set. Similarly, f is another 4-bit vector, which is the output and SEL is an input. And in this conditional operator, we are saying that if SEL is true, SEL is a 1-bit vector, if this is true means it is 1, then you select a, else you select b, select a, means a will be assigned to f otherwise b will be assigned to f.

So, actually what will happen here? This will be generating a since every conditional operator will be generating a multiplexer because it is f, then else, but here, this a and b are both 4-bit vectors and f is also a 4-bit vector. You see whenever you have a conditional thing, what you are doing? you are checking for a condition, if the condition is true, you are selecting one, if the condition is false, you are selecting the other. So, what you are actually generating is a 2 to 1 MUX.

(Refer Slide Time: 14:00)



You will be generating a 2 to 1 multiplexer, but here what has happened in the example that I have given, both a and b and f, they are all 4-bit quantities. So, actually it will not be one 2 to 1 multiplexer, but four such multiplexers will be generated.

So, each of them will be having 2 inputs and 1 output; they will be having a common select line SEL. This will be connected in parallel to all of them. Now in the inputs will be connecting for the first one a[0] and b[0], the second one will be connecting a[1] and b[1], third one will be connecting a[2] and b[2] and fourth one, we are connecting a[3] and b[3]. And in the output side, this will go to f[0], this will go to f[1], this will go to f[2] and this will go to f[3], f[3]. So, you see, a statement like that using conditional, automatically generates an array of multiplexers, like this and these multiplexers will all be 2 to 1 multiplexers, right, ok.

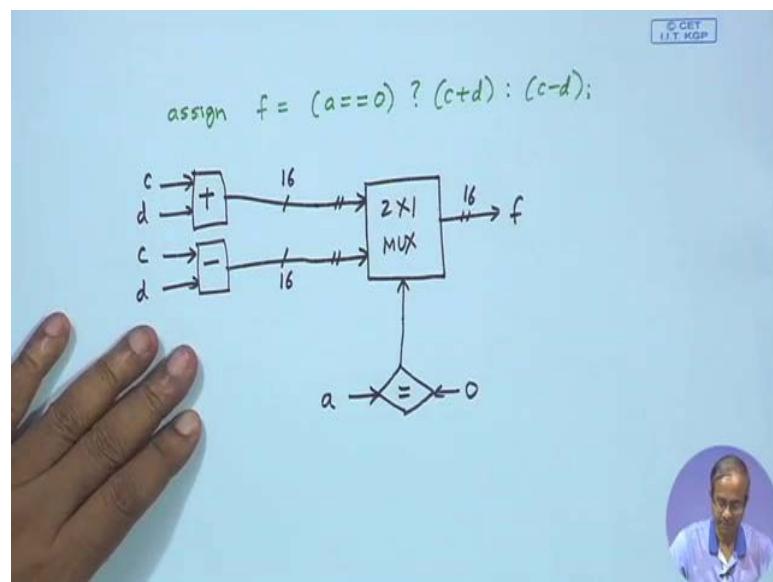
(Refer Slide Time: 15:38)

- Point to note:
 - Whenever a conditional is encountered in the RHS of an expression, a 2-to-1 MUX is generated.
 - In the previous example, since the variables "a", "b" and "f" are vectors, an array of 2-to-1 MUX-es are generated.
 - What hardware will be generated by the following?
`assign f = (a==0) ? (c+d) : (c-d);`

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

So, here the point to observe is that. So, whenever you have a conditional expression like the one we have seen, we have shown in the right hand side; automatically a 2 to 1 multiplexer will be generated. Now in the example which are shown because a, b and f vectors; so, it was not one, but an array of four such multiplexers where generated. Now let us take an example, suppose I have given an assignment statement like this. So, what will be the hardware, this assignment statement will be generating? Let us try to work this out.

(Refer Slide Time: 16:15)



So, our statement that we have seen is assign f equal to this is a conditional. So, the condition checking that we are doing is a equality 0, if it is true, c plus d, if it is false c minus d.

So, this is what we are actually trying to do. Now let us see; so, here what will happen? Here, there will be a multiplexer, which I will be, I am just showing one multiplexer, but actually, it will be an, not one, but several multiplexers. I am using the array notation, 2 to 1 multiplexers using the array or the vector notation. So, it will be having 2 inputs, these are not single lines, but multiple lines. Similarly, it will be having an output not one but multiple.

So, in the select line, we will be comparing a with 0. So, there will be a magnitude comparator. So, I need a magnitude comparator circuit, which will be taking a as input, it will be comparing it with the number 0 and the result of the comparison will be fed to the select line. So, either this will be true or this will be false. And on the other side, I use an adder, where I have fed the numbers c and d, this sum, I am connecting here and I also have a subtractor, where again I have fed the number c and d, this I am connecting here.

So, this will be the kind of circuit that will be generated. Now let us say, if this adders, let us say this is all 16-bit operation, this numbers are all 16-bits, then actually this is a vector kind of a notation, the output is f. So, actually it will be not one multiplexer, but sixteen such multiplexers will be generated, one for every bit of this data. There are 16-bits, so, 16-bits have to be multiplexed. So, I need sixteen 2 to 1 multiplexers, right, ok.

(Refer Slide Time: 18:33)

The screenshot shows a slide from a presentation. At the top, there is a red box containing Verilog code for a decoder module:

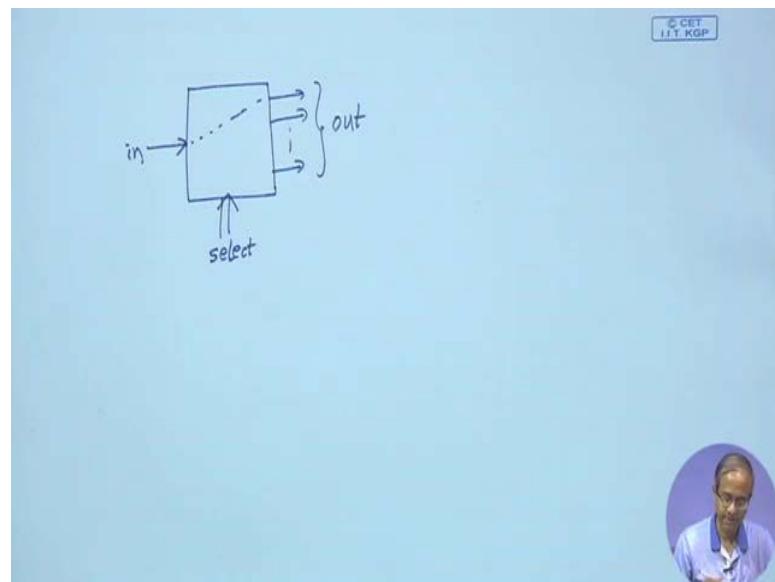
```
module generate_decoder (out, in, select);
    input in;
    input [0:1] select;
    output [0:3] out;
    assign out[select] = in;
endmodule
```

A red arrow points from the text "Non-constant index in expression on LHS generates a decoder" to the line "assign out[select] = in;". Below the slide, there is a footer with logos for IIT Kharagpur and NPTEL, and the text "NPTEL ONLINE CERTIFICATION COURSES" and "Hardware Modeling Using Verilog". To the right of the footer is a circular profile picture of a man.

So, let us take another example. This is the example of a decoder means this is an example which will generating a decoder. So, here you see, what you have given? Here, we have one input, in is a single bit input, select is 2-bit vector and out is a 4-bit vector and what we have written a something like this, out with in index of variable equal to in (out[select] = in;). So, now, the array access with the variable index is appearing on the left hand side. So, whenever such a thing happens, a decoder will be synthesized.

So, if we have on the left hand side, a non-constant index select, then a decoder will be generated. So, what the decoder will do? the input value will be going to one of the outputs depending on select.

(Refer Slide Time: 19:34)



So, now your circuit will be something like this. You have a decoder or a de-multiplexer whatever you call. The input will be in single bit data, but in the output, there will be several lines. This will be your out and this input will be connecting to one of the outputs and which one? that will be determined by the select line select, ok. So, in a assignment statement, if there is, on the left hand side, an expression like this, out with the variable in the index, then a decoder will be generated.

(Refer Slide Time: 20:19)

- Point to note:
 - A constant index in the expression on the LHS will not generate a decoder.
 - Example: `assign out[5] = in;`
This will simply generate a wire connection.
 - As a rule of thumb, whenever the synthesis tool detects a variable index in the LHS, a decoder is generated.

The slide footer includes the IIT Kharagpur logo, NPTEL Online Certification Courses logo, and the title "Hardware Modeling Using Verilog". A circular video player icon in the bottom right corner shows a man speaking.

Now, again if instead of a variable, there is a constant, this will again generate a simple wire connection, this in, whatever is in, it will directly be connected to the fifth bit of the vector out. So, there will be no circuit that will be generated, just a wire. So, as I had said whenever the synthesis tool will find out a variable index in the left hand side, a decoder is generated and if it finds a variable index in the right hand side, a multiplexer will be generated. So, these are some simple rules which the synthesis tool often uses or utilizes, ok, fine.

(Refer Slide Time: 21:02)

The slide contains the following elements:

- Verilog Code:**

```
module level_sensitive_latch (D, Q, En);
    input D, En;
    output Q;
    assign Q = En ? D : Q;
endmodule
```
- Truth Table:**

En	D	Q _n
0	x	Q _{n-1}
1	0	0
1	1	1
- Description:** Generates a D-type latch
- Text Box:** Here is an example to describe a sequential logic element using "assign" statement.
- Logos:** IIT Kharagpur and NPTEL
- Page Footer:** NPTEL ONLINE CERTIFICATION COURSES, Hardware Modeling Using Verilog

Now, here we take an example where we are using an assign statement to describe a sequential logic element, which is actually a level sensitive D-type latch. So, what does this do? Ok, this E will be capital, ok. So, what is this description doing? There are three parameters D and enable (En) at the inputs and Q is the output. So, what is this assignments statement doing? it is saying if enable is true; that means, it is 1, then D, D will be going to Q otherwise if enable is 0, Q will be going to Q, Q will be going to Q means it will be remembering its previous state, there will be no change.

So, this Q will be going to Q, this kind of assignment whenever it is there. So, the synthesizer will immediately deduce that well, I need to generate a latch or a memory element because I have to memorize some data value. So, as this example says that for certain condition of the enable, when enable is 0, I have to remember the value of Q because the output Q will be equal to the old value of Q, ok.

So, this will correspond to the state table like this. So, when enable is 0, this Q will be selected. So, D will be don't care irrespective of D. Whatever was the previous value of Q, I am denoting as Q_{n-1} that will be the new value of Q, I am calling it Q_n . But if enable is 1, then whatever is the value of D that will be copied to Q, ok.

(Refer Slide Time: 23:01)

- Modeling a simple S-R latch:

S	R	Q_n
1	1	Q_{n-1}
0	1	0
1	0	1
0	0	?

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

So, this single assign statement will be generating a D-type latch. Similarly, if you look at a simple R-S or S-R type latch using a cross coupled gates. Here I am showing NAND gates; cross coupled NAND gates. So, a circuit like this which is 2 input R and S and Q and Qbar. So, if you again look at this state table, what will happen if both S and R are 1 and 1, then whatever is Qbar, Q and Qbar. Let us say Q is 1 and Q bar is 0, if this is 1; 1 will come, 1 and 1 will be 0 and 0 and 1 will be 1; so no change, similarly for the reverse.

So, when the inputs are 1 and 1, the previous input value will remain, this is the behavior of this circuit, ok. But if S equal to 0 and R equal to 1, S equal to 0 will force this output to become 1. So, Qbar will be 1 and Q will be 0 because if Qbar is 1, this 1 and R equal to 1 will make the output of NAND gate 0, so, Q will be 0. Similarly if S is 1 and R is 0, the reverse will happen, if R is 0, Q will be 1 and this 1 and 1 will make this 0 and hence this is 1. But 00 is an indeterminate state, if we apply 0 and 0 then you see, these are NAND gates; both output will become 1 and 1, which is inconsistent with Q and Qbar not only that after that if I apply 11; what is Q_{n-1} , that will be confusing to the hardware and also to the simulator, ok, because this is not a valid state, both Q and Qbar is 1.

So, which one will become 0, that will depend on the relative delays of this two gates, which one is faster that will force the output to it to change first that is called race condition, fine. So, this circuit, the exactly the way I have shown here; you can use two assign statement, in one assignment; you are using the NAND of Qbar and R to generated Q and the other one Q and S to generate Qbar.

(Refer Slide Time: 25:23)

```

module sr_latch (Q, Qbar, S, R);
    input S, R;
    output Q, Qbar;
    assign Q = ~(R & Qbar);
    assign Qbar = ~(S & Q);
endmodule

module latchtest;
    reg S, R;    wire Q, Qbar;
    sr_latch LAT (Q, Qbar, S, R);
    initial
        begin
            $monitor ($time, "S=%b R=%b, Q=%b, Qbar=%b",
                     S, R, Q, Qbar);
            S = 1'b0; R = 1'b1;
            #5 S = 1'b1; R = 1'b1;
            #5 S = 1'b1; R = 1'b0;
            #5 S = 1'b1; R = 1'b1;
            #5 S = 1'b0; R = 1'b0;
            #5 S = 1'b1; R = 1'b1;
        end
endmodule

```

The screenshot shows a Verilog simulation environment. On the left, the code for the `sr_latch` module is displayed, defining inputs `S` and `R`, and outputs `Q` and `Qbar`. It contains two assign statements: one for `Q` (NAND of `R` and `Qbar`) and one for `Qbar` (NAND of `S` and `Q`). On the right, the code for the `latchtest` module is shown, which instantiates the `sr_latch` module and monitors the variables `S`, `R`, `Q`, and `Qbar` over time. The monitor block sets initial values for `S` and `R`, then toggles them through various states (1'b0, 1'b1, 1'b0, 1'b1) to test the latch's behavior. The bottom of the slide features the IIT Kharagpur logo, NPTEL Online Certification Courses logo, and the title "Hardware Modeling Using Verilog". A small circular portrait of a man is also present.

So, just like this two, just two assign statements, this is NAND, AND and then NOT. R and Qbar, this is Q and S and Q, this is Qbar, just two assign statement to be generating two NAND functions and this will be specifying the behavior of this latch.

Now, just for the purpose of testing, we also wrote a sample test bench for this, to check whether this latch is actually working like a latch should work. We wrote a test bench like this, hope where this S-R latch was instantiated, we call it LAT. So, this S and R which were the inputs were declared as reg and Q and Qbar the outputs were declared as wire and we monitored all the variables S, R, Q and Qbar. So, what we did initially, we set the input S and R to 0 and 1. So, if the inputs are 0 and 1, the output should be 0, ok, then we set the inputs to 1 and 1, so, that the output should not change. Then we change to the other state 1 and 0, so, the output should become Q, should become 1, then 1 and 1.

Then just for testing we applied the invalid condition 00 followed by 11 to see that what this simulator does here.

(Refer Slide Time: 26:53)

Simulation Output

0 S=0, R=1, Q=0, Qbar=1
5 S=1, R=1, Q=0, Qbar=1
10 S=1, R=0, Q=1, Qbar=0
15 S=1, R=1, Q=1, Qbar=0
20 S=0, R=0, Q=1, Qbar=1

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES Hardware Modeling Using Verilog

So, what this simulator did was something like this; first few lines were absolutely perfect what we expected, $S=0, R=1$ generates $Q=0$ and $Qbar=1$, then we apply both 11. So, the same state remains, no change. Then we apply 1 and 0, so, Q becomes 1 and $Qbar$ becomes 0. Then 11 same state, no change. Then we apply S equal to 0, R equal to 0 which is 1 and 1, but after that we again applied 11, but the problem is that the simulator got confused and the simulator hangs. It continuously tries to deduce that what will happen when S equal to 1 and R equal to 1, but because of the race condition, the output never converges. So, simulator assumes that the two gates have exactly equal delays.

So, when the outputs are 1 and 1 and we apply 11, outputs will become 0 and 0, next state it will again become 1 on 1, next state it will again become 0 and 0. So, both the outputs will oscillate indefinitely, it will never converge and the simulator will go on trying to compute the output value indefinitely. So, this was exactly what was happening here, right. So, the crux is that you should not apply such invalid inputs to an S-R flip-flop in actual circuit operation.

So, with this, we come to the end of this lecture; where we have actually looked at some of the modeling styles. In particular; looked at the assign, again we took some examples of assign; earlier also just work out a number of examples. So, we again saw assign here and we saw that not only combination circuits, we can also model sequential circuit elements. Whatever circuits I can draw using gates, I can also implement that using assign

statements, like you know how to design flip-flops. I just show the example of a cross coupled NAND gates. You can design a J-K flip-flop, S-R flip-flop, T flip-flop, all using gates, NAND gates, NOR gates, then H-triggered flip-flop. So, once you have a gate level circuit diagram, using assign statement or using instantiation of the gates, you can create those designs. So, those are also some ways to create sequential circuit elements in Verilog.

So, in the next lecture, we shall be continuing with some more Verilog constructs and examples.

Thank you.

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

Lecture - 13
Procedural Assignment

So, in the last lecture, we looked at the data flow or the continuous kind of assignment statement in Verilog, and how they can use to model both combination and sequential circuits. Now, today we shall be looking at the other kind of assignment statement which is the so-called Procedural Assignment statement, ok.

And we shall see the different types and how we can use it in the Verilog language.

(Refer Slide Time: 00:46)

Behavioral Style: Procedural Assignment

- Two kinds of procedural blocks are supported in Verilog:
 - The “initial” block
 - Executed once at the beginning of simulation.
 - Used only in test benches; cannot be used in synthesis.
 - The “always” block
 - A continuous loop that never terminates
- The procedural block defines:
 - A region of code containing *sequential* statements.
 - The statements execute in the order they are written.

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

So, the Behavioral style or the Procedural Assignment statement in Verilog comes in two flavors and we have already seen some examples using these. These are sometimes called Procedural blocks, they come in two different types, one is the initial block other is the always block. The main difference between these two are the initial block is executed only once; and this is used exclusively for writing test benches or test harnesses and these are not used in synthesis. When your objective is to synthesize the circuit, you should not use the initial construct. Initial construct is to be used only when you are writing the test benches. It is executed only once at the beginning of the simulation.

And the other type is called always block, well this always block can be used inside the test bench or it can also be used inside your circuit description module. This is like a continuous loop, which never terminates. You see, the always block models a hardware circuit in a more natural way, whenever we design a hardware that is supposed to work as long as power is switched on. So, we cannot say that this circuit hardware circuit will be working for 10 seconds and then it will automatically stop. It will not stop unless you expressly send a single to stop it. So, the always block is like a piece of hardware which is taking some input, generating some output and continuously it is doing it in a repetitive fashion. It never stops, it is more like working in a continuous loop.

Now, these are called Procedural blocks initial or always. Now, inside the Procedural block you have some code which comprises of a set of sequential statements. Sequential statements means they execute in the order they appear or they are written one by one.

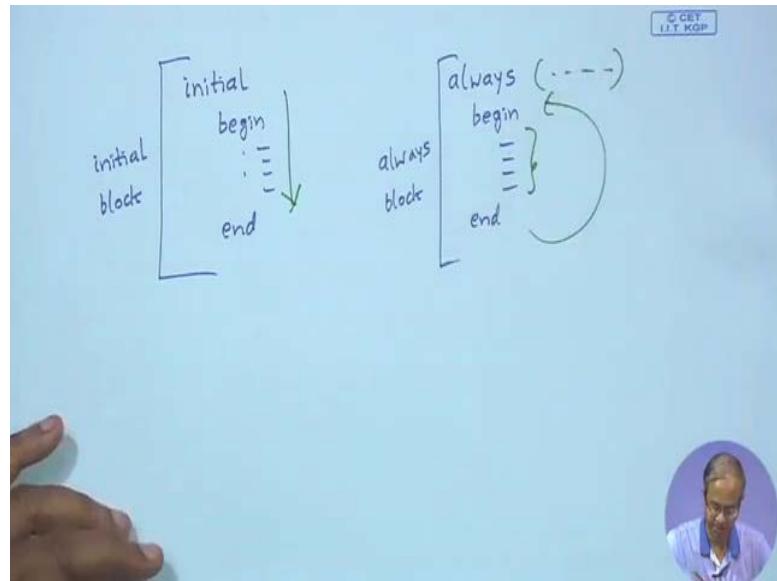
(Refer Slide Time: 03:15)

The “initial” Block

- All statements inside an “initial” statement constitute an “initial block”.
 - Grouped inside a “begin ... end” structure for multiple statements.
 - The statements starts at time 0, and execute only once.
 - If there are multiple “initial” blocks, all the blocks will start to execute concurrently at time 0.
- The “initial” block is typically used to write test benches for simulation:
 - Specifies the stimulus to be applied to the design-under-test (DUT).
 - Specifies how the DUT outputs are to be displayed / handled.
 - Specifies the file where the waveform information is to be dumped.

So, let us see. First the initial block, well we have already seen some examples of the initial block. We have seen some test benches. We shall see some elaborate, some more complex test benches later again, but at least whatever we have seen. So, you can appreciate that this statements that appear inside the initial statement, that is referred to as initial block, and that typically grouped between begin and end. So, inside an initial block it comes like this.

(Refer Slide Time: 03:56)



When you give initial, these are statement. So, inside it this statements are grouped in begin and end and this is what is defined as an initial block right. Now, the rule for execution is that the statements that is there inside this begin-end, they start executing at time zero and they execute only once, this is important. So, if you do not specify any explicit timing by giving that hash(#) command, it is assumed that the first statement inside that block will start executing at times t equal to 0, and initial means this statements will be executing only once, fine. And in a typical test bench there can be multiple initial blocks. And if there are multiple initial blocks, then all the blocks will be executing in parallel concurrently and all of them will start at time t equal to 0, this is something you should remember.

Now, as I have said repeatedly that this initial block is used to write test benches. But sometimes when you are trying to see how some typical Verilog constructs are working you can also use initial block just for testing. You can do some computation, you can give a display or a monitor statement to print it, just to see whether it is working or not, but not for synthesis. When you are designing and you are target is to generate the hardware, you can forget about initial. Initial is something which you should not and cannot use, fine.

So, this initial block is used for writing the test benches. The test benches as you have seen they basically specify this stimulus or the inputs that are to be applied to your design whatever you have designed. And also it is specifies how the output of your design are to

be handled, whether they are to be displayed or they are to be dumped into a file, so that you can display the wave forms later. This test bench specifies all these things and they are typically done inside always blocks, always means initial blocks.

(Refer Slide Time: 06:30)

The slide shows a Verilog testbench example:

```
module testbench_example;
reg a, b, cin, sum, cout;
initial
  cin = 1'b0;
initial
begin
  #5 a = 1'b1; b=1'b1;
  #5 b = 1'b0;
end
initial
#25 $finish;
endmodule
```

A callout box highlights the three initial blocks and their execution characteristics:

- The three "initial" blocks execute concurrently.
- The first block executes at time 0.
- The third block terminates simulation at time 25 units.

The slide footer includes logos for IIT Kharagpur, NPTEL Online Certification Courses, and Hardware Modeling Using Verilog, along with a portrait of a speaker.

So, some simple example, so this is the example of a test bench where I am not instantiating the model because our objective is to just to illustrate the initial blocks. Here you see we have used three initial blocks. And if there are multiple statements inside an initial block, you need this begin and end, but if there is a single statement, you do not need the begin and end, it is optional. So, all the three initial blocks, they start execution at time t equal to 0. You see what is happening at t equal to 0, this is something like a full adder, you see the names are quite familiar a, b, cin (carry in), sum and cout (carry out).

So, I am initializing with the cin (carry in) as 0 and in the other initial block which also starts at time t equal to 0, it encounters a delay of 5. So, it waits for 5 units and then it applies a equal to 1, and b equal to 1. Then it again waits another 5 units and then applies b equal to 0 and that's it, this is a single execution. And the third initial block, this also starts at time t equal to 0, it waits for 25 time units and then calls the system function finish, which means the end of simulation. So, at time t equal to 25, this simulation will stop. So, such multiple means, initial blocks are very typical inside a test bench. Now, inside a test bench you can also have always blocks as you shall see later with some examples. So, as I have set to summarize the three initial blocks execute concurrently; the first block

because there are no delays, this statement executes at time t=0, the third block calls finish. So, it terminates simulation at time t=25, ok.

(Refer Slide Time: 08:37)

Some Short Cuts in Declarations

- “output” and “reg” can be declared together in the same statement.
 output reg [7:0] data;
 instead of output [7:0] data; reg [7:0] data;
- A variable can be initialized when it is declared:
 reg clock = 0;
 instead of reg clock; initial clock = 0;

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

Now, just to note that there are some short cuts in declarations, which are supported by Verilog, means at least the recent version of Verilog, which are available now. See normally whenever you have some parameter in a module which is an output and which is assigned inside a Procedural statement which has to be a reg. You declare it like this output, let us say it is a vector 7 to 0 [7,0] data and again you have to give reg 7 to 0 [7,0] data to indicate that this data is also a registered type. So that it can be assigned inside a Procedural block later on in the code.

But Verilog now permits us to combine this two statements in a single statement like this you can straight away right output reg [7,0] data. So, you are specifying two things together, the data is an output and also it is a registered type variable. Secondly, see earlier you use to write something like this say a clock, you declare a clock as a register type variable then in initial block, you initialize clock to 0 (clock = 0). Suppose, this is the single statement inside the initial block. Now this initialization and declaration can be done together in a single statement now, you can write something like this reg clock equal to 0 (reg clock = 0). Because here also you see, clock equal to 0, initialization will take place at time t equal to 0, because no delay is specified; same thing will happen here whenever you are declaring clock at time t equal to 0, it is also initialized, fine.

(Refer Slide Time: 10:26)

The “always” Block

- All behavioral statements inside an “always” statement constitute an “always block”.
 - Multiple statements are grouped using “begin ... end”.
- An “always” statement starts at time 0 and executes the statements inside the block repeatedly, and never stops.
 - Used to model a block of activity that is repeated indefinitely in a digital circuit.
 - For example, a clock signal that is generated continuously.
 - We can specify delays for simulation; however, for real circuits, the clock generator will be active as long as there is power supply.

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

Now, let us come to the always block the other kind of Procedural block. Now there is a statement, the always statement just like we said that there is an initial statement and it comprises as an initial block in exactly the same way. There is an always statement inside the always statement typically you have begin-end again. So, begin-end can group several statements together and this whole thing is called a always block, right. So, the set of statements which are Behavioral statements inside the always, they constituent the always block. And as I have shown in that example, these block is grouped using begin and end.

Now, just like initial, this always block, this statements inside it also starts at time t equal to 0 and executes this statement inside the block repeatedly and never stops. What I mean is that here inside this always block, I have this begin-end statement. There can be several statements here. Now, these statements are executed repeatedly in an infinite loop kind of a thing and it never stops. Initial executes only once, it executes once and stops, but always will never stop, it will execute the block repeatedly in response to some event condition that I will mention very shortly, this is the difference.

So, the always block is typical used to model some activity which is repeated indefinitely in a digital circuit, which is characteristic of a digital circuit. So, as long as power is switched on it will continue. For example, I have clock generator as long as there is power the clock signal will go up and down up and down indefinitely it will never stop. So, only when I forcibly switch off the clock generator or switch off the power only then the clock

will stop, fine. So, this is the example I have also given here, the clock signal that is generated continuously.

Well for simulation purposes, we can specify delays even within always block, but again when we are actually generating a hardware, the clock generator, this delays does not mean, does not carry any meaning. So, as long as there is the power supply, it will go on, of course in simulation you can specify some delay. We can say that after some delay it will stop, but in an actual hardware there is certain things that we cannot do that will see later on.

Because you see you have looked at so many different constructs in the language something which you can do only for simulation, something you do typically for synthesis. So, we shall see later we will talk specifically about it, there is a subset of the language which of course, is a little subjective it can vary from one software or synthesis tool to another, there is a subset which is called synthesizable subset. There is a set of statements or set of features, which are only allowed by the synthesis tool. If you allow or use any other construct of the language, this synthesis tool will simply ignore them, it will not take them fine.

(Refer Slide Time: 14:28)

```
module generating_clock;
    output reg clk;
    initial
        clk = 1'b0; // initialized to 0 at time 0
    always
        #5 clk = ~clk; // Toggle after time 5 units
    initial
        #500 $finish;
endmodule
```

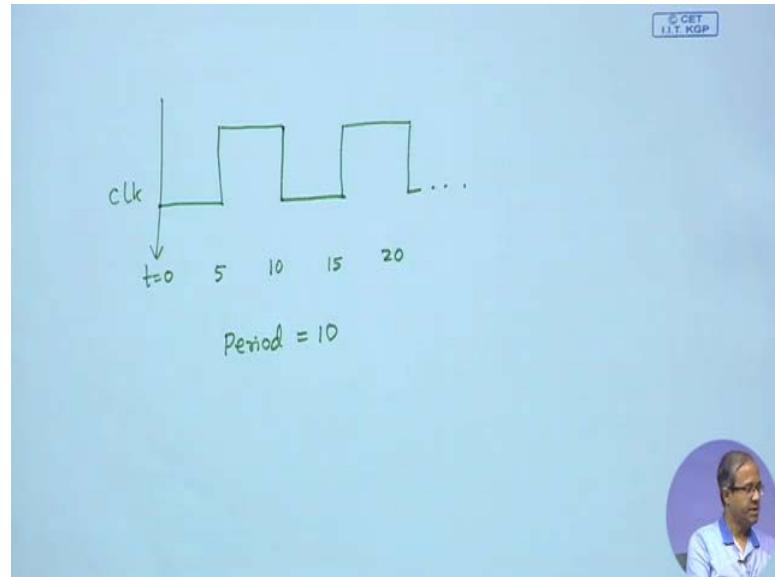
• “initial” and “always” blocks can coexist within the same Verilog module.
• They all execute concurrently; “initial” only once and “always” repeatedly.



So, here is an example, just a clock generator. So, this is just a module, just like a test bench which is generating a clock signal, this an output reg. You see this initial block at time t equal to 0 is initializing the clock signal to 0. So, initialize to 0 at time 0. And in the

test bench I can also give an always block. I have given a always block which says that with the delay of 5, you use clock equal to not clock [$\sim \text{clk}$]. What does this mean?

(Refer Slide Time: 15:13)



Suppose this is my time t equal to 0, and this is my signal clock, my clock was 0 at time t . At time 5, clock equal to not clock [$\text{clk} = \sim \text{clk}$], my clock becomes high. At time 10, after gap of another five again clock equal to not clock, it will become 0. At time 15, it will again become 1. At time 20, it will again become 0 and so on. So, with the gap of five, the clock will continually toggle and the time period will be double the delay, it will be 10; 5 off period and 5 on period. So, this is how we specify that after every five time the clock will toggle from 0 to 1, and 1 to 0 and just for simulation we can specify this that at time at time 500 this simulation should finish. So, the clock will go on toggling up to time 500.

So, inside a Verilog module, initial and always blocks can coexist. You can have a combination of initial and always blocks, but of course, within the test bench only, not within the main Verilog module and they all execute concurrently. The difference I mentioned that initial will execute only once, but the always block will execute indefinitely repeatedly.

(Refer Slide Time: 16:49)

The slide has a yellow background with a black header and footer. In the center, there is a box containing text and code. The text says:

- A module can contain any number of "always" blocks, all of which execute concurrently.
- The @(event_expression) part is required for both combinational and sequential circuit descriptions.

Below this, a box contains the **Basic syntax of "always" block:**

```
always @(event_expression)
begin
    sequential_statement_1;
    sequential_statement_2;
    ...
    sequential_statement_n;
end
```

The footer of the slide includes the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the title "Hardware Modeling Using Verilog". There is also a circular portrait of a man.

So, a module in general can contain any number of always blocks, I mean a module where I am describing the hardware and these blocks are all executing concurrently. Now, I told you just sometime back, this always block will be executing in response to some event. You see this is the general syntax of an always block, after always you give this at the rate symbol and within bracket you give an event expression [always @(event_expression)].

So, whenever this event expression occurs or takes place, this statements inside the begin-end block will execute sequentially. And this event expression, this is required for modelling both Combinational and Sequential circuits as we will see through examples slowly. So, this is the general syntax. There are a number of sequential statements inside the always block and every time this block will get executed whenever this event expression triggers. So, let say this event expression can be a clock. So, every time a clock is coming, it will execute; again next time clock comes again it will execute something like that.

(Refer Slide Time: 18:07)

- Only "reg" type variable can be assigned within an "initial" or 'always' block.
- Basic reason:
 - The sequential "always" block executes only when the event expression triggers.
 - At other times the block is doing nothing.
 - An object being assigned to must therefore remember the last value assigned (not continuously driven).
 - So, only "reg" type variables can be assigned within the "always" block.
 - Of course, any kind of variable may appear in the event expression (reg, wire, etc.).

And there is some restrictions here. So, inside the initial or always block, whenever you are assigning some variables to some values, only reg type variables can appear on the left hand side. You can assign values only to reg type variables. The basic reason is that you see, the sequential always block as I had said it will execute only when the event expression triggers.

(Refer Slide Time: 18:40)

```
always @ (...)  
begin  
    @ = b+c;  
    @ = b+2;  
end
```

reg → hardware registers
reg → combinational circuit

Like what I have said is that always at the rate some event expression I do not care what it is, what I am saying is that inside the begin-end block, let say I am writing some statement

like this $a = b + c$, I write $d = b + 2$, something like this. So, whenever this event expression will appear or will trigger, these two statements will execute, but now think what will happen during the time when this triggering condition is not true. So, what will happen to the statements whatever you have assigned to a and d should remain in a and d that is why we say that the left hand side must naturally be reg type variables. So that if required this reg type variables can be mapped to hardware registers by the synthesizer.

But it is not necessarily true, sometimes you will see that the variable may be reg type variable, but whatever you are mapping, whenever you are mapping it, so it is actually mapping to a combinational circuit. So, no registers or storage elements are required, flip-flops latches are not required. So, a reg can map to either a sequential element or a combinational circuit, but if we use a net type variable, net type variable can never be sequential, it will always be combinational, right. So, this is what I mentioned the object which you are assigning must remember the last value assigned, this is not continuously driven. So, you must have reg type variables in the left hand side, but on the right hand side or in the event expression you have any combination of variables reg, wire etc.

(Refer Slide Time: 20:50)

Sequential Statements in Verilog

- In Verilog, one or more sequential statements can be present inside an “initial” or “always” block.
 - The statements are executed sequentially.
 - Multiple assignment statements inside a “begin ... end” block may either execute sequentially or concurrently depending upon the type of assignment.
 - Two types of assignment statements: blocking ($a = b + c;$) or non-blocking ($a <= b + c;$).
- The sequential statements are explained next.

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

Now, let us come to the sequential statements in Verilog. So, we have said that inside Verilog or in the always block, you can have a number of sequential statements, such sequential statements can also be there inside the initial block. They are supposed to be

executing sequentially one by one. So, what is the idea, there are multiple assignments statements inside a begin-end block may be there, which may execute sequentially or concurrently depending upon the type of assignment, this is something we will be discussing later. What we are saying is that inside a begin-end block, there can be some assignment statements. These assignment statements may execute one by one sequentially or they may be executing concurrently, depending on what kind of assignment we are making, this we shall be seeing later.

There are two type of assignment statement - one is called blocking, which is denoted by the equal [=] to symbol and there is another type called non blocking, which is denoted by less than equal [≤] to or the arrow symbol. Now, the sequential statements, whatever is available in the Verilog language will be explained now one by one.

(Refer Slide Time: 22:20)

(a) begin ... end

```
begin
    sequential_statement_1;
    sequential_statement_2;
    ...
    sequential_statement_n;
end
```

- A number of sequential statements can be grouped together using "begin .. end".
- If n=1, "begin ... end" is not required.

Navigation icons: back, forward, search, etc.

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

The most important and the basic which we have already seen is the begin-end. Begin-end is the basic block, which combines a number of sequential statements in to one composite statement. So, a number of sequential statements can be grouped together. Now, if the number statement is 1, then the begin-end keyword is not required, you can simply write the statement, right.

(Refer Slide Time: 22:51)

(b) if ... else

```
if (<expression>)
sequential_statement;
```

```
if (<expression>
sequential_statement;
else
sequential_statement;
```

```
if (<expression1>)
sequential_statement;
else if (<expression2>)
sequential_statement;
else if (<expression3>)
sequential_statement;
else default_statement;
```

- Each sequential_statement can be a single statement or a group of statements within "begin ... end".

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

Now, if-else, you see these constructs are very similar to the one, which are available in high languages like C. So, I am showing three different types of if-else construct, one is a simple leaf without an else part. If some logical expression, it can be true or false; if it is true then execute this sequential statement. Now, you see one thing, I have written just sequential statement. And just in the previous one I said begin-end is also sequential statement, so that sequential statement itself can be this begin-end block. So, although I have written a single sequential statement, so there can be begin and end inside if and there can be five statements there. Several statements can be here with some begin-end.

So, in the second version there can be an else. If this logical expression is true, then this block or sequential statement will execute it, else this block will be executed. So, you can have a begin-end here, you can again have a begin-end here. In a third version we are showing nested if then else if-else. It says if expression 1 is true, then you execute this sequential statement, else-if expression 2 is true then execute this, else-if this is true then execute this otherwise default statement. So, you see the expressions are checked in a particular order.

The first match whatever you get that particular statement will be executing. So, inside this block exactly one of the statements will execute either this one or this one or this one and if none of them match the default one. And here as I had said that each sequential statement can be either a single statement on its own or they can be grouped using begin-

end. Now, we can have a more concise version of this if then because you can see if then else nested one, it becomes quite complex in terms of the structure.

(Refer Slide Time: 25:18)

The slide has a yellow header bar with the text '(c) case'. Below this is a light blue rectangular area containing Verilog code:

```
case (<expression>)
  expr1: sequential_statement;
  expr2: sequential_statement;
  ...
  exprn: sequential_statement;
  default: default_statement;
endcase
```

To the right of the code is a red-bordered box containing a bulleted list of points:

- Each sequential_statement can be a single statement or a group of statements within "begin ... end".
- Can replace a complex "if ... else" statement for multiway branching.
- The expression is compared to the alternatives (expr1, expr2, etc.) in the order they are written.
- If none of the alternatives matches, the default statement is executed.

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the text 'NPTEL ONLINE CERTIFICATION COURSES', and the course title 'Hardware Modeling Using Verilog'. To the right of the footer is a circular portrait of a man.

So, you can have a concise one well in C language, you have a similar construct called switch case; in Verilog, we have case. Here you say that we give an arithmetic expression inside this case with in bracket [case (<expression>)]. So, it evaluates to a value and that values compared with these expressions you just specified before the colon [:]. So, it compares with this expression sequentially one by one, it first compares with expression one, then expression two, expression three, up to expression n.

And whichever it is matching that corresponding sequential statement will be executing. And if none of them matches then there can be a default, then this default statement will be executing and case will end with endcase. So, each of this sequential statements like the earlier ones, they can be either a single statement or it can be a group of statement inside begin else. So, this becomes much simpler as compared to a complex if-else kind of a construct.

Now, the order in which you list this expressions, they will actually give you the order in which you are checking because you see the same expression the way you are comparing, it can match with more than one condition, there can be some don't care values also; so the first one that matches will be the one that will be taken.

(Refer Slide Time: 26:57)

• Two variations: "casez" and "casex".

- The "casez" statement treats all "z" values in the case alternatives or the case expression as don't cares.
- The "casex" statement treats all "x" and "z" values in the case item as don't cares.

If state is "4'b01zx", the second expression will give match, and next_state will be 1.

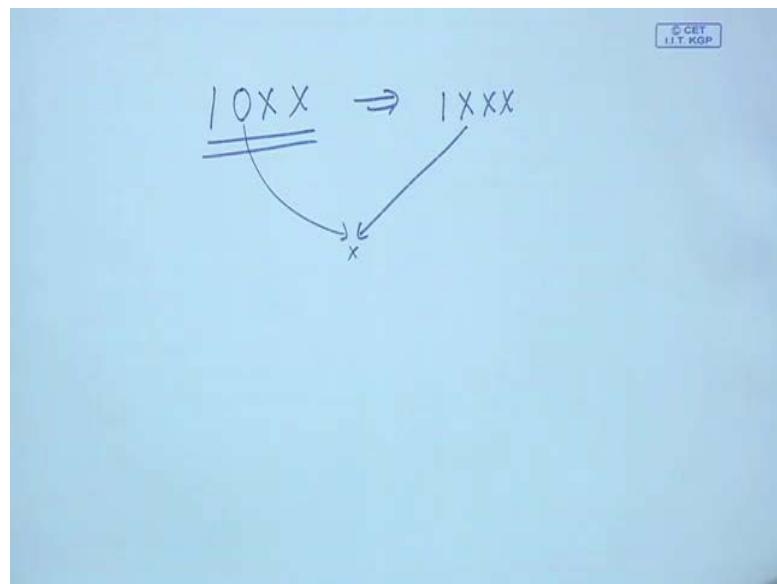
```
reg [3:0] state; integer next_state;
casex (state)
  4'b1xxx : next_state = 0;
  4'bx1xx : next_state = 1;
  4'bxx1x : next_state = 2;
  4'bxx11 : next_state = 3;
  default : next_state = 0;
endcase
```

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

Like there are two variations of case you can have in Verilog, one is called casez, others called casex. In casez statement, here the high impedance or the z values in the inputs and the expressions are treated as don't cares, but in casex not only z also this x, they are treated as don't cares. Let us take a small example. Suppose I have declared a state vector 4bit, next_state is an integer, I am using a casex on the state 4bit. And you see the values of the expressions I am specifying they have don't cares, they are not crisp values, I am saying 1xxx, x1xx, xx1x and here last one is xxx1.

So, depending on which one, here I am saying that if the first bit is 1, then next state is 0, you do this; if the second bit is 1, you do this. But if none of this matches; that means, none of the bits are 1 then default something you can say, here I am saying next state 0.

(Refer Slide Time: 28:31)



You see if your state has a value let say 10xx, this will match with the first condition which is 1xxx because 0 and x matches, x is don't care, and 0 is one possible value of the don't care. But if you give only case then this will not match, but if you give casex then only the first bit is in the state is 1 or not other three can be anything, need not be xxx need be 01 x, 01z, anything. So, this is how you can give. So, there are several other kind of constructs also. So, we have only seen a few begin-end, if then else, case. So, in our this what I am saying, this state is this, this has an example 01zx, this second expression will be matching 01zx because x matches with anything and next state will 1.

So, in the next lecture we shall be continuing with this. We will see that in Verilog, there are some other kind of sequential statements also available because we have so far talked about begin-end, if then else, if-else, nested if-else and case and some versions of case. We have not talked about the different looping constructs for, while and some others. So, in the next lecture, we shall be talking about those and also we shall be seeing some examples.

Thank you.

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

Lecture – 14
Procedural Assignment (Contd.)

So, in the last we were discussing some of the Procedural assignment statements are the procedural sequential statements in Verilog. So, we talked about statements like the begin-end block and if-else, nested if-else, and the multi way branching using case. So, we continue with our discussion today on Procedural assignments.

(Refer Slide Time: 00:49)

(d) "while" loop

```
while (<expression>
    sequential_statement;
```

- The "while" loop executes until the expression is *not true*.
- The sequential_statement can be a single statement or a group of statements within "begin ... end".

Example:

```
integer mycount;
initial
begin
    while (mycount <= 255)
        begin
            $display ("My count:%d", mycount);
            mycount = mycount + 1;
        end
    end
```

IIT Kharagpur

NPTEL



And the first construct that we talk about today is the so called "while" loop. Well, this while statement is variable is available in standard high level languages like C or C++. So, the syntax in Verilog is very similar. So, the general syntax is shown here, while within first bracket some logical expression [while (<expression>)], which can evaluate to true or false. So, while this expression is true, this sequential statement or a block of statements within begin-end will execute. So, this is a repeated repetitive execution construct that is why we call it a loop. So, as I had said this loop comprising of the sequential statement will execute until the expression is not true. So, as long as it is true, it will continue the execution. And the sequential statement as in the other constructs, it

can be either a single statements or it can also be a group of statements within begin-end block.

So, just one simple example using while: this is part of a test bench. So, here we are defining an integer called mycount; and in the initial block we are using a while loop. This is just for the sake of demonstration and illustration. So, while the value of mycount is less than or equal to 255 [while (mycount<=255)], you going to this loop, you display the value of the count and then you increase or increment the value of the count. Now here you may argue that well the first time when the programs start execute it enters the begin block, mycount is undefined. So, the first time when while checks the condition what will be the value true or false, this is not well defined.

So, in order to remove this ambiguity, before this while and after this begin you can introduce one statement actually where you can initialize mycount to 0 [mycount = 0]. Like here, you can add a statement where mycount value can be set to 0. So, if you do this then this ambiguity will be removed. So, the first time when it comes into the while loop the mycount value will be 0, so it is definitely less than equal to 255. So, this loop will continue executing, it will continue till mycount crosses 255. So, it will be displaying the value of count mycount starting from 0, 0, 1, 2, 3, 4 up to 255, this is just an example, fine.

(Refer Slide Time: 03:50)

(e) "for" loop

```
for (expr1; expr2; expr3)
sequential_statement;
```

- The "for" loop executes as long as the expression expr2 is true.
- The sequential_statement can be a single statement or a group of statements within "begin ... end".

- The "for" loop consists of three parts:
 - An initial condition (expr1).
 - A check to see if the terminating condition is true (expr2).
 - A procedural assignment to change the value of the control variable (expr3).
- The "for" loop can be conveniently used to initialize an array or memory.

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

The next looping construct is again very similar to the construct available in the language C or C++, this is the “for” loop. Just like in C, so in the “for” loop the syntax consists of three parts, there are three expressions, you can say among them expression one and expression three, they are actually assignment statements. This expression one basically specifies an initial condition or an initialization, some variables get initialized here; and expression three specifies an assignment, where some control variable that controls how many times the loop will go on, that control variable is updated that is expression three. And expression two is a logical expression which checks if it specifies a termination condition. So, as long as this condition is true the loop will go on.

So, again there is a sequential statement which can be a single statement or it can be a group of statements inside begin-end. So, this “for” loop will start by initializing by executing expression one then it will continue as long as expression two is true; and at the end of every iteration expression three will be executed once to update the control variable. This is just like what is available in standard high-level languages like C.

(Refer Slide Time: 05:29)

```

Example:

integer mycount;
reg [100:1] data;
integer i;

initial
    for (mycount=0; mycount<=255; mycount=mycount+1)
        $display ("My count:%d", mycount);

initial
    for (i=1; i<=100; i=i+1)
        data[i] = 1'b0;

```

Just a simple example using “for”, this again is part of a test bench, where here I am showing two examples one by one. So, let us say we have defined an integer variable called mycount and a vector of bits of type reg. So, it is a vector of size 100, where the index values range from 1 to 100 called data and an integer i. So, in the first example, there can be multiple initial blocks. I told you. Suppose, in the first initial block, we are just

doing exactly what we did earlier using the “while” loop. Using “for” loop we are displaying the count values starting from 0 up to 255. So, this will be the initial condition before starting the loop mycount will be initialized to 0; and at the end of the loop, so you update your control variable mycount equal to mycount+1, execute this, it will be incremented, then you check whether this condition is still true or not. So, as long as this is true, this loop will go on. So, this display will be executing repeatedly.

Now, in the second example this is again another initial block. Here we are using “for” loop to initialize this vector to all zeros. So, here as you can see we are using the “for” loop to run through the index values 1 to 100, starting with 1 and up to 100, and at the end of every iteration we increase i by 1, data i equal to 0 [date[i] = 1b'0]. So, these are some examples for using “for” loops.

(Refer Slide Time: 07:16)

(f) “repeat” loop

```
repeat (<expression>
sequential_statement;
```

- The “repeat” construct executes the loop a fixed number of times.
- It cannot be used to loop on a general logical expression like “while”.

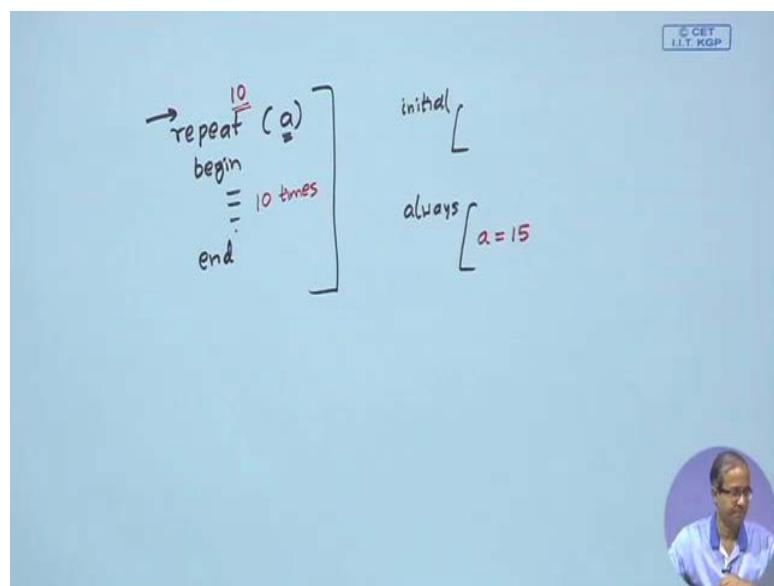
- The expression in the “repeat” construct can be a constant, a variable or a signal value.
- If it is a variable or a signal value, it is evaluated only when the loop starts and not during execution of the loop.
- The sequential_statement can be a single statement or a group of statements within “begin ... end”.

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

Now, the third kinds of loop which is there is “repeat”. Now, see the main difference is like this for “while” or for the “for” loop, we specify a condition. So, we do not say directly that how many times the loop should go on. So, in the “while” loop, we specify a condition and as long as that condition is true, i continue executing the loop. Similarly, for the “for”, there is a logical expression we have specified in expression two you recall, there we specify a condition again, where as long as that condition is true, we continue with the execution. But in contrast in this “repeat” statement, we do not do like that we specify exactly how many times we want to run the loop.

So, in this “repeat” statement, this expression whatever is there, this actually tells how many times this loop has to execute. So, this construct is used to execute the loop some fixed number of times. And it does not use a logical expression which indicates that when to stop and when to continue with the loop. So, some constraints on this expression that you can give here, well this expression can either be a constraint, it can be a variable or it can be some signal value, signal value means which is continuously driven. Now, if it is a constant, it is fine you know how many times to loop, but if it is a variable or a continuously driven signal then the rule is that these value is evaluated only when the loops starts.

(Refer Slide Time: 09:18)



Suppose, at the beginning of the loop you see let say I have given a statement like this repeat some variable a [repeat (a)], then within begin-end, there are some statements. This a might be, this might be updated by some other “initial” or “always” block, that can be an “initial” block or there can be an “always” block, where this variable “a” might get updated. But here the rule says that when this loop starts, you take the value of “a” which was there at that point in time.

Suppose the value of “a” was 10, so this loop will be executing 10 times. But even if during execution of the loop, suppose in this block the value of “a” is changing to 15, but still because the value of “a” was 10 at the beginning of the loop, this loop will be executing 10 times only, right, this is the rule we have to remember, fine that is the rule.

(Refer Slide Time: 10:25)

The slide has a yellow background with a black header bar at the top. In the center, there is a white box containing Verilog code. To the right of the code, there is explanatory text. At the bottom, there is a footer bar with logos for IIT Kharagpur and NPTEL, and the title "Hardware Modeling Using Verilog". On the right side of the footer, there is a circular profile picture of a man.

Example:

```
reg clock;
initial
begin
    clock = 1'b0;
    repeat (100)
        #5 clock = ~clock;
end
```

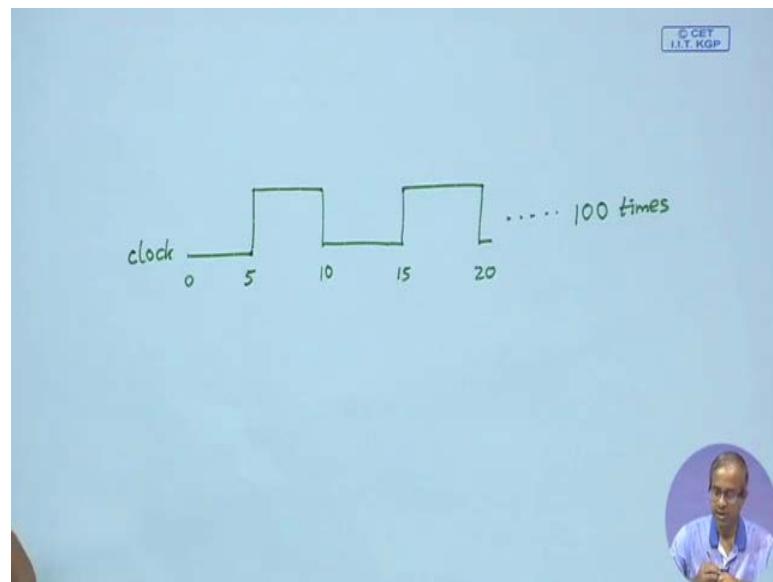
Exactly 100 clock pulses
are generated.

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



So, one simple example here we are actually generating a fix number of clock pulses in a test bench, let say here we are defining a signal clock as of type reg. Well, I am repeating, you will have to define a variable of type reg if it appears on the left hand side inside an “initial” or an “always” block. Here because it is appearing on the left hand side, you have to use reg. So, what we are doing we are initializing the clock to zero then there is a “repeat” loop, repeat 100 times. So, what we are doing repeat 100 times. We are giving a delay of 5 and we are changing the value of clock, the clock was 0, so not clock [~clk] will be 1, 1 will be assigned to clock. So, again after delay of 5, not of 1, 0 will be there, that 0 will be assigned and this will repeat 100 times.

(Refer Slide Time: 11:33)



So, it will be something like this. This variable clock, this will be initialized to 0. And at time 5 this is at time 0, at time 5, it will be toggled, it will become 1. Again at time 10, it will remain at 1; at time 10, it will again come back to 0. At time 15, it will again become 1; at time 20, it will again become 0, and this will continue exactly 100 times, right.

So, if you want in some application to generate a clock which will be free running continuously without any constraint then you do not give this kind of restriction like 100 pulses then you can do something “always” or there is another construct we will see later called “forever” which can implement an infinite loop, a loop which never stops. But for “repeat” you have to specify exactly how many times you are expected to carry on or continue with the loop.

(Refer Slide Time: 12:40)

(g) “forever” loop

```
forever
    sequential_statement;
```

- The “forever” loop is typically used along with timing specifier.
 - If delay is not specified, the simulator would execute this statement indefinitely without advancing \$time.
 - Rest of design will never be executed.
- The “forever” construct does not use any expression and executes forever until \$finish is encountered in the test bench.
 - Equivalent to a “while” loop for which the expression is always true.
- The sequential_statement can be a single statement or a group of statements within “begin … end”.

IT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

Next is with “forever” loop as I had said, this is a version of a loop, where you execute the sequential statement infinite number of times “forever”. In the “forever” construct, you do not use any expression or any count value how many times to loop. So, you will be continuing with this loop “forever”. Now with respect to the test bench if some block executes finish then only this will be terminated; otherwise this will go on continuously. So, you can say that this “forever” construct is like a “while” loop, where the expression is always true. So, in place of the expression, you give something which is always true, like, you can write something as follows.

(Refer Slide Time: 13:35)

©CET
I.I.T. KGP

while ($I < 2$)
begin
 =
end

always true

=====

forever
begin
 =
end

Let say just if you write while 1 less than 2 [while (1<2)], do something, begin, a block of statements, end. Now, clearly this 1 less than 2 is a constant expression this one which is always true. So, this while loop is equivalent to a forever construct, where you may use forever and again the same set of statements inside begin-end. These two will be equivalent. So, this sequential statement again it can be single statement or it can be a group of statement within begin-end.

Now, there is one issue here you have to be careful about this statement “forever” is a kind of a statement which is unconditional, it does not check anything and continuously go on executing. So, it is very important to specify a delay using some hash [#], #5, #10 something like that. What might happen is that if you do not specify the delay then the simulator will start this “forever” loop and it will never execute and because there is no delay, the time will also be not advancing. So, simulator would execute this statement indefinitely infinitely without advancing time.

Now, because time is not advancing, the rest of the design, the following statements of other blocks, where there is a delay specified, it will never be executed. Because you see it may so happen that you have started to execute this forever statement at a time t equal to 5, let say when dollar time is equal to 5. Now, there is another initial block, which says that at time 10, you have to do something, you have to print or you have to assign something, but because this “forever” has started and it has not finished, it is in an infinite loop, so time will never advance. And the other statement which is supposed to be activated at time 10, it will never be activated. So, whenever you use “forever” be sure to use a timing specify there.

(Refer Slide Time: 16:17)

```
// Clock generation using "forever" construct
reg clk;
initial
begin
    clk = 1'b0;
    forever #5 clk = ~clk; // Clock period of 10 units
end
```

The screenshot shows a Verilog code snippet. The code defines a register 'clk' and initializes it to 1'b0. It then enters a 'forever' loop where it toggles the value of 'clk' every 5 time units using the assignment 'clk = ~clk'. A comment indicates that the clock period is 10 units. The code is presented in a light blue box with a dark border.

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the text 'Hardware Modeling Using Verilog'.

Like let us take an example, a simple example that uses a clock signal indefinitely. So, inside the initial block, we again say initialize a signal clk to 0 then we give a forward loop with a timing delay, forever delay of 5 clock equal not clock [forever #5 clk = ~clk]. So, this will generate a clock with a time period of 10 units, 5 time units which will remain 1, 5 time units it remains 0, and this will go on indefinitely because it is a forever loop. But now the time is advancing 5, 10, 15, 20 it is advancing, so you are not holding up any other statements in the other initializer always blocks, ok, fine.

(Refer Slide Time: 17:05)

Other Constructs Available

(time_value)

- Makes a block suspend for "time_value" units of time.
- The time unit can be specified using the `timescale command.

@ (event_expression)

- Makes a block suspend until "event_expression" triggers.
- Various keywords associated with "event_expression" shall be discussed with examples..

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the text 'Hardware Modeling Using Verilog'. On the right side, there is a circular inset showing a man speaking.

So, the other constructs that are available in Verilog one of course, you have used this time specifier hash [#] followed by some time value. So, the meaning of this statement is a block that has this kind of timing delay specified at the beginning of it, the block will be suspended for this much amount of time before it gets executed. And the time unit can be specified as you have mentioned earlier using the time scale command. So, it can be some unit nanosecond, picoseconds, second, millisecond anything, fine.

So, the other important construct which you have already seen through some examples which you, where particular in the always block you use this, this is at the rate [@] with in bracket some event expression. Here what you are doing, here we are making a block suspend its execution until this event expression triggers. So, this event expression can be specified in various ways we will see, there are some keywords using which you can specify this event expression.

(Refer Slide Time: 18:21)

@ (event_expression)

- The event expression specifies the event that is required to resume execution of the procedural block.
- The event can be any one of the following:
 - a) Change of a signal value.
 - b) Positive or negative edge occurring on signal (*posedge* or *negedge*).
 - c) List of above-mentioned events, separated by "or" or comma.

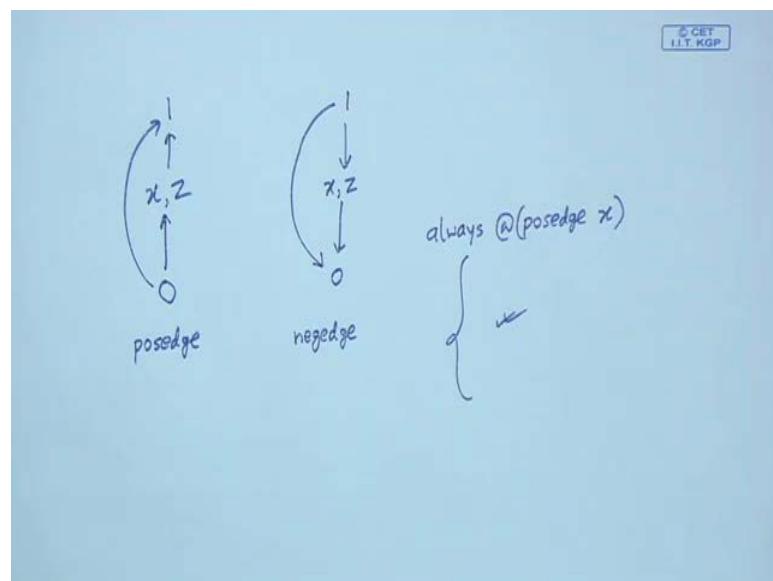
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog | 10

Let us see how. So, just to recall, so in the event expression where actually specifying some event, and when that event occurs whatever is in the procedural block inside that that will be executed or it will be resuming its execution. Because you see we mention that in an “always” block, what you do inside “always”, there is a group of statements within begin-end. Let us say, this “always” block is supposed to execute this block of statement repeatedly, but when it will not do so continuously. Whenever the event expression is true, then it will be executing once; then it will go back and wait till the

event expression gets true again. If it happens it will execute a second time, then it will again go back, again it will check whether the event expression is true. And if it is not true, it will wait till it becomes true, it will be executing it third time in this way it will go on. So, event expression will tell you exactly when to execute the block the next time.

So, the event actually can be in general one of the following. So, either the change of a signal value, so you must specify whenever a particular signal value or any of the values change, then do this or you can specify either a positive or negative edge of a signal. This is particularly used for sequential circuit, synchronous sequential circuit specifications using the keyword posedge or negedge. Or you can have a combination of several of these which is separated by a keyword “or” or you can also use a comma [,]. So, by definition in Verilog, whenever we talk about a positive edge, well a posedge does not necessarily mean a transition from 0 to 1; actually a posedge is defined as any transition where a signal changes from either 0, x or z to 1; or from 0 to z, or x.

(Refer Slide Time: 20:53)



So, the idea is that in terms of the hierarchy, 0 is considered to be lowest, 1 is considered to be highest, x and z are considered to be in between. So, you define a posedge, I mean if either you have a transition from 0 to x, z, from x, z to 1, or from 0 to 1 in this way you define posedge. Similarly, you can define a negative edge as transition like this. So, where again the idea is similar, so you again have a hierarchy where x then you have x, z then you have 0.

So, any transition where you are either moving from 1 to 0, or 1 to x or z, or x or z to 0, this is defined as negedge. So in Verilog whenever the value of a variable is like you specify like this always at the rate posedge of some variable let say x [always @(posedge x)]. So, whenever the value of a variable x changes from a value which is from 0 to 1, or 0 to z, 0 to x, x to 1, x means don't care here, do not confuse this x with this x, then a positive edge will be defined and the block which is inside will get executed.

(Refer Slide Time: 22:38)

- Examples:
 - @ (in) // "in" changes
 - @ (a or b or c) // any of "a", "b", "c" changes
 - @ (a, b, c) // -- do --
 - @ (posedge clk) // positive edge of "clk"
 - @ (posedge clk or negedge reset) // positive edge of "clk" or negative edge of "reset"
 - @ (*) // any variable changes

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

So, let us take some examples of this event specification. So, I can right at the rate with in bracket the name of a variable. This actually means the event, which is defined as a change in the variable. So, whenever the variable in changes, this event is said to have occurred. Similarly, I can write a or b or c, this means whenever any of the variables either a or b or c or more than one changes then this condition is considered to be true and the block will be executed. Now, in place of "or" you can also write a comma, the meaning is the same. So, you can either write the variable separate by "or", or the variables separated by commas. So, I mentioned posedge, so you can specify the posedge on some variable, clk is the variable I am referring to a clock here.

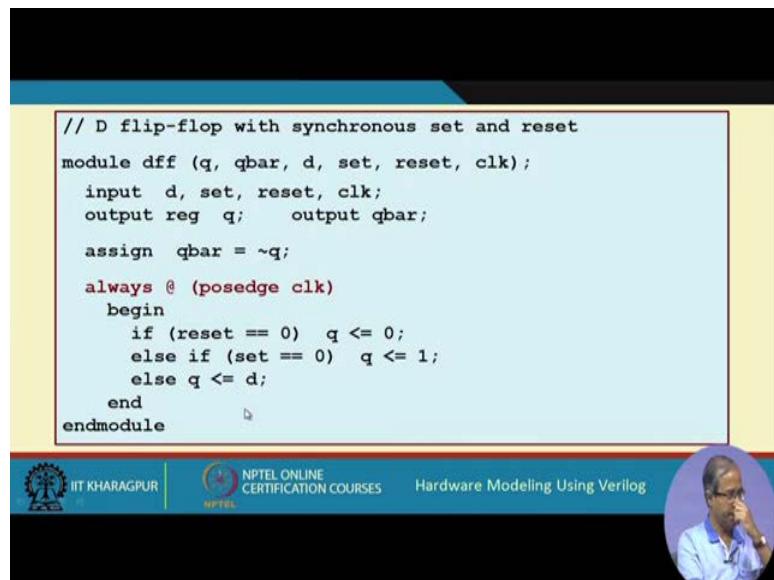
So, whenever there is a positive edge, I have defined what is posedge on this variable, this event is considered to be true. So, you can combine several of the events. You can say that at the rate either posedge clock or negedge of reset [@(posedge clk or negedge reset)]. So, whenever either the positive edge of clock comes or the negative edge of

reset comes, you execute the block. The event you can define a composite event by using “or” and in place of “or” you can also put comma like this, both are allowed.

And the last one I have been telling here I am putting a star [*]. You see here I have given at the rate a or b or c. So, if there are hundred variables in my Verilog code, I can, I may have to write all the hundred variables. Now, if I write a star, star means any, if any of the variable changes execute the block. So, in some designs, in some modules writing a star becomes more convenient because first thing is that you will not have to write so many variable names and secondly you may have forgotten to list one of the variables.

So, you see that your simulation is not coming correctly, may be you have missed one of the variable, d was not there, but if you write star then there is no problem. So, you know that you have included these variable all of them, fine.

(Refer Slide Time: 25:21)

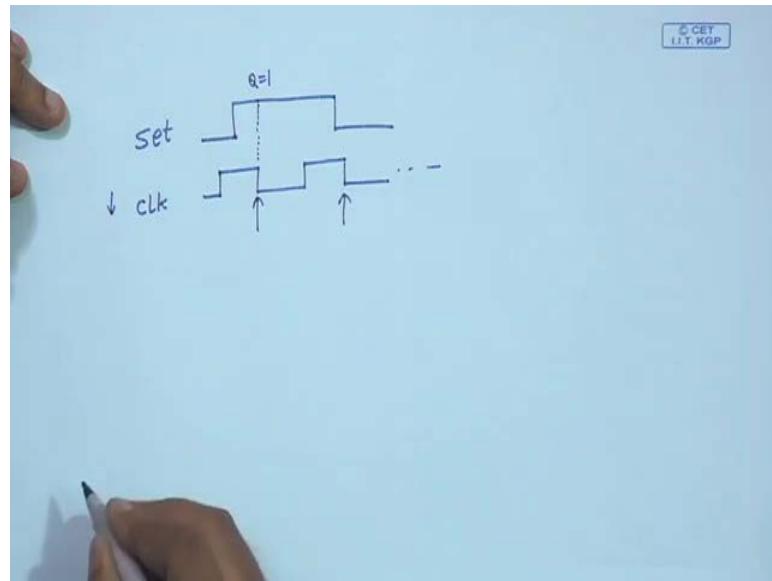


```
// D flip-flop with synchronous set and reset
moduledff (q, qbar, d, set, reset, clk);
    input d, set, reset, clk;
    output reg q;      output qbar;
    assign qbar = ~q;
    always @ (posedge clk)
        begin
            if (reset == 0) q <= 0;
            else if (set == 0) q <= 1;
            else q <= d;
        end
    endmodule
```

The screenshot shows a Verilog code editor with a yellow background. The code is a module named 'dff' with five inputs: 'd', 'set', 'reset', 'clk', and two outputs: 'q' and 'qbar'. It includes an assignment for 'qbar' and an 'always' block that triggers on the rising edge of 'clk'. Inside the 'always' block, it checks the 'reset' and 'set' signals to update the 'q' value accordingly. The code is highlighted in red and blue for keywords and identifiers. At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the NPTEL logo, and the course title 'Hardware Modeling Using Verilog'.

Let us take an example, some illustrations of this “always”, usage of “always”. So, here I am showing the module description of a D type flip-flop with synchronous set and reset. So, when I say a flip-flop has synchronous set and reset. It means whenever the clock, the active clock edge comes, setting and reset of the flip-flop takes place in synchronism with that clock edge, this is meant by synchronous set or reset. Everything setting means, setting the flip-flop to one; resetting means setting it to 0; this happens only when the next clock edge comes and the corresponding set or reset signals are active, like let me give an example.

(Refer Slide Time: 26:17)



Suppose I have a set signal, let say I have set the signal to high and it goes high like this and this is my clock, my clock is going like, suppose my clock is triggered by the falling edge, let say my clock signal is going like this. So, I have one active edge here, I have one active edge here, and this will continue. So, whenever the falling edge of the clock comes, I check whether set is high or low. Suppose set is active high, so I find it high here. So, the flip-flop will be set, the output Q will be set to 1. So, the next clock edge, I find set is already return back to 0, so nothing will happen, no change. Synchronous means, changes take place at the active edge of the clock, right, setting and resetting similarly.

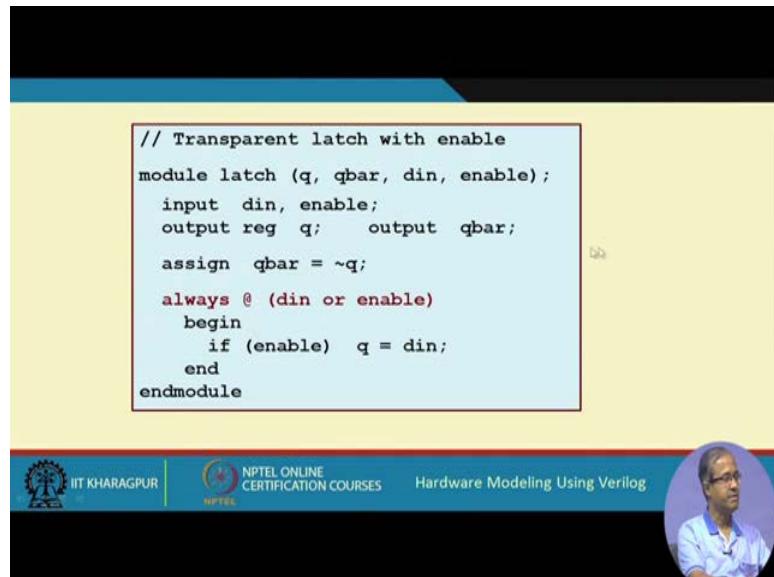
So, let us see the module, we have q, qbar, d is the input and set, reset and the clock. So, d, set, reset, and clock are the inputs; and this q is defined as an output which is also a reg. Well, we could have defined qbar also as reg, but this qbar, we are generating directly from q through an assign statement, qbar equal to not q [qbar = ~q]. Because inside this always block, we are only initializing q and qbar we are generating as the not of q; and inside this always block what we are doing always at the rate posedge clock. So, whenever the clock edge comes, a positive edge these example shows 0 to 1, you do this.

You check if reset is 0 then you set the output q to 0; else if set is 0 then set q to 1; and otherwise if neither set or reset is active. So, set and reset are active low here. Whenever

they are 0, they are active; otherwise you store the input data in q. This is synchronous because you are executing the block only when the active edge of the clock is coming. So, if the reset you have made 0 earlier, so it will not be reset immediately. Now, the meaning of this equal [=] and less than equal [≤] we shall be explaining later, but for the time being you understand the meaning of these statements. This you can easily understand, fine.

Let us take another small modification where we have made it asynchronous set and reset. So, in the earlier example here, we are only triggering this block at the posedge of the clock, but now you are saying at posedge of the clock or negedge of set or negedge of reset. That means, I am not only waiting for the clock to come, but if I have seen that the set signal has gone down to 0, so immediately I can execute this block; or the reset signal has gone down to 0, I can immediately execute this block. So, this is called asynchronous set and reset, where the set and reset operations do not depend on the clock signal.

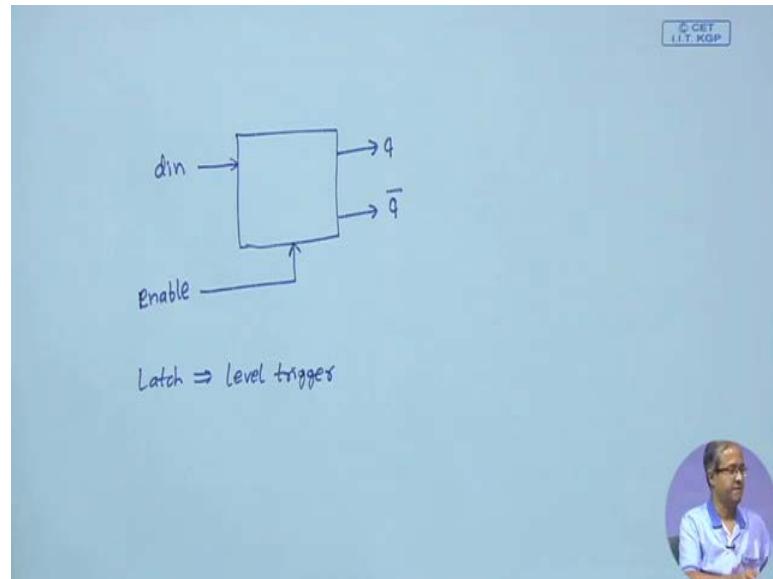
(Refer Slide Time: 29:37)



```
// Transparent latch with enable
module latch (q, qbar, din, enable);
    input din, enable;
    output reg q;      output qbar;
    assign qbar = ~q;
    always @ (din or enable)
        begin
            if (enable) q = din;
        end
endmodule
```

And in this example, we have a transparent latch with enable. Transparent latch mean that there is no clock, there is a latch q, qbar, there is a data in [din] and there is an enable. So, it is a something like this.

(Refer Slide Time: 30:00)



There is a latch where there is an input called din, there are outputs one is q, one is qbar, and there is an enable, this is not a clock. So, whenever this enable is active, if enable is 1 then din will be stored in the latch. So, this is level triggered. You recall a latch means this is level triggered, this is not activated by the edge like in a flip-flop, fine. So, the description is very simple. So, you declare din and enable as input q as reg and again this output qbar is an output, which you can generate using assign not of q. And in this block there is no clock. So, in this always block, you are seeing din or enable. So, whenever either the input changes or the enable changes, you execute this block, it says if enable is true which means enable is 1, 1 means true then you assign d into q, this is the description of a transparent latch.

So, with this we come to the end of this lecture. Now, in the next lecture, we shall be looking at some of the examples that use some of the constructs that we have already seen. And with respect to synthesis, we shall also be discussing a few of the very interesting concepts.

Thank you.

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

Lecture - 15
Procedural Assignment (Examples)

So, over the last couple of lectures we had seen the various kinds of assignments data flow and the procedural kinds of assignments. Now in this lecture we shall be showing you some examples of Verilog module descriptions using mainly the procedural kind of assignment statements. So, by doing that we will be understanding some of the design styles and techniques which are good to use and also some of the implications. If you create the design in certain way, then you are expected to get something otherwise you will be getting something else. We will be explaining a few such issues ok.

(Refer Slide Time: 00:58)

The slide shows a Verilog module definition for a 2-to-1 multiplexer. The code is as follows:

```
// A combinational logic example
module mux21 (in1, in0, s, f);
    input in1, in0, s;
    output reg f;

    always @(in1 or in0 or s)
        if (s)
            f = in1;
        else
            f = in0;
endmodule
```

To the right of the code, there is a callout box containing two bullet points:

- The event expression in the “always” block triggers whenever at least one of “in1”, “in0” or “s” changes.
- The “or” keyword specifies the condition.

The footer of the slide includes the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the text "Hardware Modeling Using Verilog".

So, we start with an example of a simple 2 to 1 multiplexer. See we have seen earlier that we had created the multiplexer design using various behavioral and also structural techniques. We use the instantiation of gates to use or create a multiplexer description. We use the assign statements to define the behavior of a multiplexer. So, we also use, we use that you just recall that vector with a variable index on the right hand side that creates a multiplexer, various ways you have seen. But here we are using a procedural

kind of description that also defines the multiplexer using some kind of an if-else statement, so let us see this.

So, in this example we are trying to create a simple 2 to 1 multiplexer, where the inputs are in0 and in1 and the output is f and the select line is s. So, in1, in0 and s are the inputs. So, f is an output which is also reg, see again I am declaring f as reg because inside this always block this f is appearing on the left hand side. Here in the always block I am giving the condition, the event expression is whenever any of the input changes in1 or in0 or s. So, whenever any of the input changes I update my value of f, what I do, if s is true; that means s is 1 then this in1 will go to f otherwise if s is 0 in0 will go to f.

Now, see whenever you describe something using this “or” notation like this, normally we are expressing or trying to express a combinational circuit behavior, but of course, there can be exceptions we shall be seen later, that even if we specify the event expression by naming variables using “or” there are cases where a sequential circuit might be synthesized. But we will try to explain when it happens, but in the example I have given here it is a simple case of a combinational circuit that will be generated. So, here a simple 2 to 1 multiplexer will be generated where this in0 and in1 will be the inputs, this s will be the select line.

Because you are checking on s and f will be the output. This “or” keyword combines the variables to form the event expression, ok.

(Refer Slide Time: 03:53)

The slide displays a Verilog module definition for a 2-to-1 multiplexer. The code is as follows:

```
// A combinational logic example
module mux21 (in1, in0, s, f);
    input in1, in0, s;
    output reg f;

    always @ (in1 or in0 or s)
        if (s)
            f = in1;
        else
            f = in0;
endmodule
```

To the right of the code, a callout box contains the following notes:

- The event expression in the “always” block triggers whenever at least one of “in1”, “in0” or “s” changes.
- The “or” keyword specifies the condition.

At the bottom of the slide, there is a footer with the IIT Kharagpur logo, the NPTEL logo, and the text "NPTEL ONLINE CERTIFICATION COURSES". To the right of the footer is a circular portrait of a man.

Now, I have a slightly modified version of the same program where in the earlier one I we give “or”, so instead of “or” we can give commas same thing. This is just a alternate way expression which may be a little more compact instead of writing “or” so many times. You can simply separate the variables by commas. Now this notation is supported in the recent versions of Verilog.

So, you can use this, fine, such the third variation.

(Refer Slide Time: 04:25)

```
// A combinational logic example
module mux21 (in1, in0, s, f);
    input in1, in0, s;
    output reg f;

    always @(*)
        if (s)
            f = in1;
        else
            f = in0;
endmodule
```

- An alternate way to specify the event condition by using a "*" instead of naming the variables.
- "*" is activated whenever *any* of the variables change.

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

So, instead of listing the variables I mentioned we can put a star [*]. What does the star mean? So, whenever any of the input variable changes, now here my input variables are in1, in0 and s. So, whenever any of them changes this always block gets activated and the block is executed. So, the body of the block is the same. So, these three versions are equivalent. So, you can see this is perhaps the most compact, and in many design you may use this because it will make a code shorter, fine.

(Refer Slide Time: 05:09)

```
// A sequential logic example
module dff_negedge (D, clock, Q, Qbar);
    input D, clock;
    output reg Q, Qbar;
    always @ (negedge clock)
        begin
            Q = D;
            Qbar = ~D;
        end
endmodule
```

- The keyword “negedge” means at the negative going edge of the specified signal.
- Similarly, we can use “posedge”.
- We can combine various triggering conditions by separating them by commas or “or”.

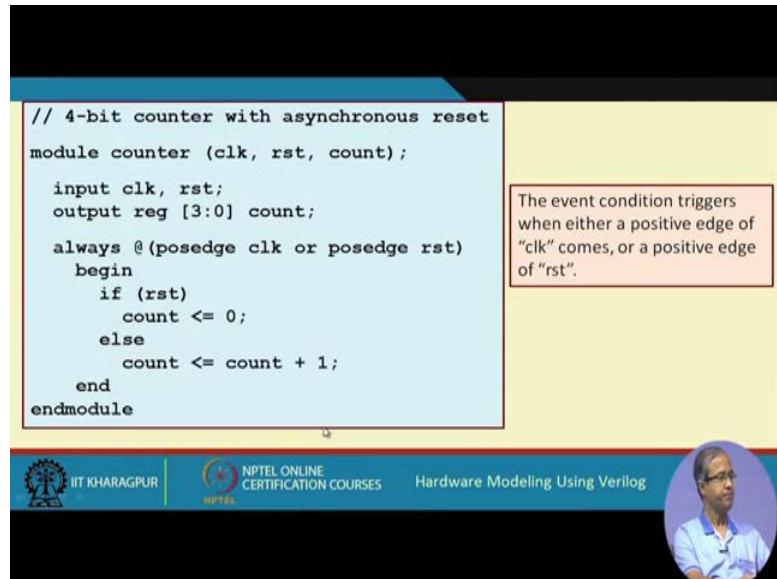
IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES Hardware Modeling Using Verilog

Now, let us come to some sequential logic examples. This is an explicit synchronous D type negative edge triggered flip-flop where you are using a clock with a negedge. So, I mean I in this description we are trying to define a D flip-flop with a D input outputs Q and Qbar with a clock. So, D and clock we are declaring as inputs. And both Q and Qbar we are declaring as output reg. Well of course, we could have done what we saw in the earlier examples, where this Q we can declare is reg and Qbar we can generate by using an assign not of Q. But here we have chosen to just assign Q and Qbar both inside the always block, this is also a correct description. So, what we are doing? Inside this always block at the rate negedge of the clock.

So, whenever there is a negative edge of the clock we are executing two statements, what we are doing? Whatever is on D that goes to Q and whatever is on Dbar that goes to Qbar? This is the logic that is, that defines a D type flip-flop, which is negative edge triggered. So, just again to repeat we mentioned this in detail earlier that what is meant of meaning of negedge. Negedge means whenever this signal is having a negative going edge. Now negative going edge the meaning I told either 0 to 1, or 0 to x or z, or x or z to 0. Similarly, if you want to have it triggered on the positive edge you can use posedge instead of negative edge.

And this condition, several conditions you can separate by commas, “or”, ok.

(Refer Slide Time: 07:11)



```
// 4-bit counter with asynchronous reset
module counter (clk, rst, count);
    input clk, rst;
    output reg [3:0] count;
    always @ (posedge clk or posedge rst)
        begin
            if (rst)
                count <= 0;
            else
                count <= count + 1;
        end
endmodule
```

The event condition triggers when either a positive edge of "clk" comes, or a positive edge of "rst".

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



Let us take another example this is a simple 4 bit counters. So, you know what a 4 bit counter is? So, a counter is a digital circuit, normally if we do not specify anything else we mean it is a binary counter; that means it counts in binary. Suppose it is a 4 bit counter, it can store a value which is 4 bits long 0000 up to 1111, that means 0 to 15. So, whenever a clock comes, the counter will be counting up to 0, 1, 2, 3, 4 and whenever it reaches 15 at the next clock again it will become 0, that is how counter works.

So, here we are trying to design a counter where we can also reset it from outside by activating a signal `rst`. Reset means we can reset it to all 0. And it is asynchronous; that means, it need not be synchronized with the clock, whenever the reset is active immediately I can initialize the counter to 0. So, let us see how the description will be. So, here clock and reset are the inputs, and count here is the example of 4 bit counter. So, the count value is of 4 bits, which is output which is also `reg` again because count is appearing on the left hand side of the assignments.

So, here our event condition says, either positive edge of the clock or positive edge of reset. So, if any one of them is true, this event condition is considered to be true. So, you go inside and check first if reset was active. Positive edge means there are other positive edge and reset has become 1. So, you check whether reset is 1. Then you make the count value 0 else reset was not active just a clock came, so, you will have to increment the count by 1. This is a simple description of a counter with asynchronous reset, but if you

do not need asynchronous reset if you need a synchronous reset then you drop this second part, just only write positive clock, then everything will take place in synchronism with the clock, ok.

(Refer Slide Time: 09:36)

```
// Another sequential logic example
module incomp_state_spec (curr_state, flag);
    input [0:1] curr_state;
    output reg [0:1] flag;

    always @(curr_state)
        case (curr_state)
            0,1 : flag = 2;
            3   : flag = 0;
        endcase
    endmodule
```

The variable "flag" is not assigned a value in all the branches of the "case" statement.
• A latch (2-bit) will be generated for "flag".

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

Here this is a little surprising example. See first thing is that here we have mentioned in the comment that another sequential logic example. But you look at the description does it look like a sequential circuit or sequential circuit description. Let us look at it in some detail. This is a module. This is the name of the module, just forget the name for the time being. There are two parameters current state [curr_state] and flag. Current state is a 2-bit vector 0, 1. Flag is also a 2-bit quantity which is output also reg, because flag is appearing on the left hand side. Now let us look at our procedural block. Well, always whenever the current state changes you do this. So, what is my logic? My logic is "case", there is a case statement.

So, I check the current state. So, if the current state is either 0 or 1. So, in "case" if there are multiple cases which can result in the same expression, you can separate them by commas like in the example shows. If it is 0 or 1, you set flag to 2. If it is 3, you set flag to 0. Now the question is why am I calling it a sequential logic. So, it is clearly like a combinational logic.

(Refer Slide Time: 11:14)

curr-state	flag
0	2
1	2
2	X
3	0

Now, just you try to construct a truth table kind of a thing. Suppose I am listing my current state and the expected value of flag which I am setting. So, according to my description if my current state is 0, I set flag to 2, if my current state is 1, I set flag to 2 again. If my current state is 3, I set flag to 0.

So, it is just like a combinational circuit. But there is a catch here. Here we have not specified what will happen if the current state is 2. So, we have not specified the value of the flag. So, what is the meaning? Now the Verilog simulator or the synthesizer will assume the meaning as follows : If some of the input descriptions are missing in the case statement, like here the value 2 was missing. The interpretation will be, if current state is 2 then flag will not change. Because we have not specified the value of flag. The interpretation will be the value of flag will remain what it was in the previous state which implies that flag will map to a storage element means latch.

So, you will have to store it just for this purpose. So, whenever there is an incomplete specification in order to satisfy the interpretation that the Verilog simulator or the synthesizer assumes. That if it is one of those unspecified condition, the output value will not change, to implement that the output value has to be stored somewhere in a latch. Which implies this becomes a sequential circuit, not a combinational circuit anymore. So, for this example the variable flag is not assigned a value in all the cases of the, all the

conditions of the case statement, 0, 1, 3, we have specified, but for 2 we have not specified because flag is a 2-bit register.

So, a 2-bit latch will be generated for the flag, this is how it will work, right. Now you see, as a designer I know that if I specify a circuit like this, by not specifying all the conditions, case conditions, then the synthesizer will be generating a latch. Well although, I do not want this to be a sequential circuit. So, how can I modify it? I can make a small change in my description. Like I can just add one line here, remaining part is identical just one line here before the “case”.

(Refer Slide Time: 14:49)

```
// A small modification
module incomp_state_spec (curr_state, flag);
    input [0:1] curr_state;
    output reg [0:1] flag;

    always @(curr_state)
    begin
        flag = 0;
        case (curr_state)
            0,1 : flag = 2;
            3    : flag = 0;
        endcase
    end
endmodule
```

Before the “case” I add one line. Now there are two statements that is why I have given a begin-end, flag = 0.

So now what is the difference? Now the synthesizer or the simulator if it does a dataflow analysis, it will find that well for 0 and 1 flag is 2, for 3 flag is 0, but for the condition 2 also flag will be 0 because we have already initialized flag to 0 and I have entered; so if I do not specify a flag will remain at 0. So now, if we look at this truth table we had constructed earlier, now for this case instead of a question mark here this value will also become 0. So, this will be a pure combinational circuit. And there will be no latches that will be generated.

So, as a designer it will be your responsibility to write the specifications in such a way that latch is not generated unnecessarily unless we explicitly want it to be generated, fine. So, here as I said by doing this the variable flag is getting defined for all possible values of current state, which implies that a pure combinational circuit will be generated by the synthesis tool and the latch as in the previous example will be avoided, ok.

(Refer Slide Time: 16:35)

- When a “case” statement is incompletely decoded, the synthesis tool will infer the need for a latch to hold the residual output when the select bits take the unspecified values.
 - It is up to the designer to code the design in such a way that latch can be avoided where possible.

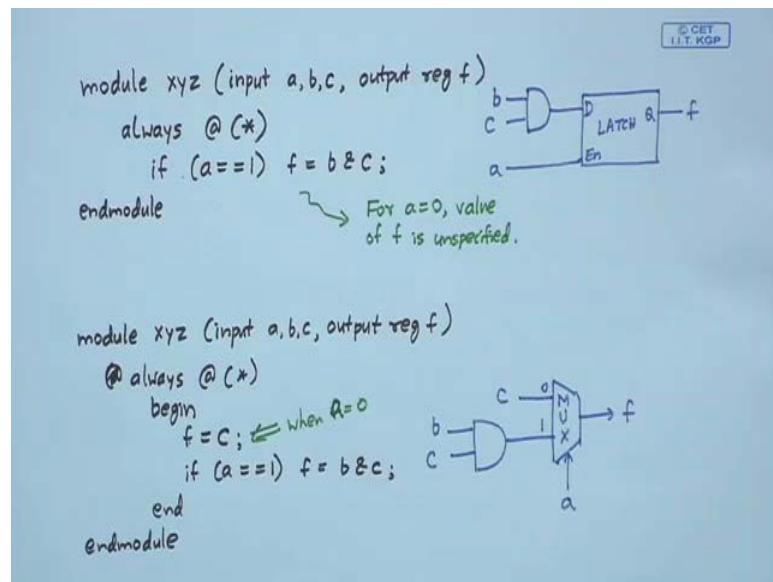
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog



So, to summarize our observation whatever you saw, that when you use a case statement in a Verilog description.

And the case statement is incompletely specified; that means, all conditions are not given. Then the synthesis tool will generate a latch to hold the output values because whenever one of those means unspecified values are coming. So, the synthesis tool will try to keep the output in the previous state, and for that it needs it to store it in a latch. Now as I said it will be up to the designer finally, to code it in such a way that latch can be avoided. Because if you specify your design in such a way that it is incompletely specified, the synthesis tool will assume that as a designer you are wanting to create a latch and a latch will be created.

(Refer Slide Time: 17:47)



Well, let us take a very simple example to just, to basically consulate this observation once more. Let us take a very simple example. Suppose we are creating a module description, let us say the name of the module is xyz. Well, another thing I am specifying here, see the input and output description you can specify here also, the modern versions of Verilog supports this. So, you can specify it like this, module xyz is means earlier you had mentioned abcf and this input and output would mention later, but you can specify in the same line also. When you are declaring, you can say input a, b, c, output reg f, this is also allowed.

Now, my module description goes like this, “always” whenever any variable changes star [*]. So, inside the “always” block there is a single statement it says, if a is equal to 1, f equal to b & c, this is bitwise AND [&], end module. Simple, this is a simple specification. So, here we are saying whenever some input is changing, you check if a is 1. So, if a is 1 you assign b & c to f. Now you see here I have not used a case statement, but there is a conditional if-else kind of thing. So, the situation is very similar. Here also we are not specifying the value to be assigned to f for all possible values of a. We say that if a equal to 1 you do this, but what will happen if a equal to 0? I have not specified. So, if a equal to 0, the interpretation will be the value of f will not change; that means, you will need a latch. So, the inference is that for the condition a equal to 0, value of f is unspecified. So, what will happen for that? So, if you give this description to a synthesis tool. So, possibly your synthesis tool will be generating a circuit like this. It will be

generating an AND gate, whose inputs will be b and c, then it will be generating a latch, where this will go to the D input. And the input a will go to the enable input, and the output Q will be your f. You see this exactly models this behavior what you have given, if a equal to 1.

So, whenever a equal to 1, this enable will be active and b c, b c will be stored in the latch. And that will be available on f, but if a equal to 0, which means you are not enabling the latch. So, your previous value stored in the latch will remain in f, this is what you wanted. Now let us look at a slightly modified version of the description. So, we are using the same example, same parameters, output reg f. Now here there were two statements that is why I am giving a always at the rate star [always @(*)]. Now here there was one statement no begin-end was required, but here I am giving a begin-end.

So, what I do? I add a statement, let us say f equal to, say suppose my descriptions like this f equal to c, and then something like this. If a equal to equal to 1[a==1], f equal to b & c, end and endmodule. So, if my description is like this then what is the interpretation? Then you see my description says, if a is 1 then f will get the value of b & c. And for a equal to 0, I have already initialized f to some value. So, this will be the value when f is 0. So, I have defined now f for both conditions sorry, for a equal to 0, for a equal to 0 this will be condition, for a equal to 1 this will be the condition. So now, I have specified the value of the output f for all values of a.

So now if you give this description to the synthesis tool, the synthesis tool will be possibly generating a circuit like this, to generate b, c, there will be AND gate. Then it will be synthesizing a multiplexer, a 2 to 1 multiplexer. Where the other input of the multiplexer will be c, this will be your input0 input1. And your select line will be a, the output will be f. So, see what is happening here? You are saying when a equal to 0, c should go to f. So, if a equal to 0, c will be selected. And if a is 1, this will be selected b, c will be selected, this will go to f. So, this is a purely combinational circuit you see. So, this is something which is up to the designer.

So, you will have to design it in a proper way. So, that the synthesis tool should not get confused and generate a latch, where I mean, what you actually wanted is a combinational circuit, right. So, it is greatly up to the designer to specify how you are wanting to guide the synthesis tool. So, you should not mislead the synthesis tool in

believing that what you are specifying is actually a sequential circuit when it was not, fine. So, let us take some more examples.

(Refer Slide Time: 24:50)

```
// A simple 4-function ALU
module ALU_4bit (f, a, b, op);
    input [1:0] op;      input [7:0] a, b;
    output reg [7:0] f;
    parameter ADD=2'b00, SUB=2'b01, MUL=2'b10, DIV=2'b11;
    always @(*)
        case (op)
            ADD : f = a + b;
            SUB : f = a - b;
            MUL : f = a * b;
            DIV : f = a / b;
        endcase
endmodule
```

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

So, here we have a very simple 4-bit arithmetic logic unit, where the parameters are f, a, b and op. But op specifies these are 2-bit, it specifies one of 4 operations, and a, b are 8-bit inputs. This f is also 8-bit output, which is of type reg, because f is on the left hand side. Well, using parameter command you can specify some constants, I give some example earlier, like here suppose we are supporting 4 operations addition, subtraction, multiplication and division.

Now to make our program more readable we have specified them as add, sub, mul and div. Here add means this op is 00, sub means 01, mul is 10, div is 11. And here we have an “always” block, where you are saying whenever it is star; that means, any of the input changes depending on the value of op, but instead of writing 00, 01, 10 and 11, I straight way I am writing add, sub, mul, div. So, it becomes much easier to understand the code. So, the operation is given directly. So, this is a very simple way to specify an ALU in a behavioral fashion, right.

(Refer Slide Time: 26:19)

The screenshot shows a Verilog module named "priority_encoder". The module has an input port "in" of type [7:0] and an output port "code" of type reg [2:0]. The logic is implemented using an always block with a @ (in) sensitivity list. Inside the always block, there is a series of if statements that map each input value to a specific output code value based on priority. The code values are: 3'b000 for in[0], 3'b001 for in[1], 3'b010 for in[2], 3'b011 for in[3], 3'b100 for in[4], 3'b101 for in[5], 3'b110 for in[6], and 3'b111 for in[7]. If none of the inputs are active, the output is 3'bxxx. The slide also contains a note about the sequential checking of inputs and the handling of simultaneous active inputs.

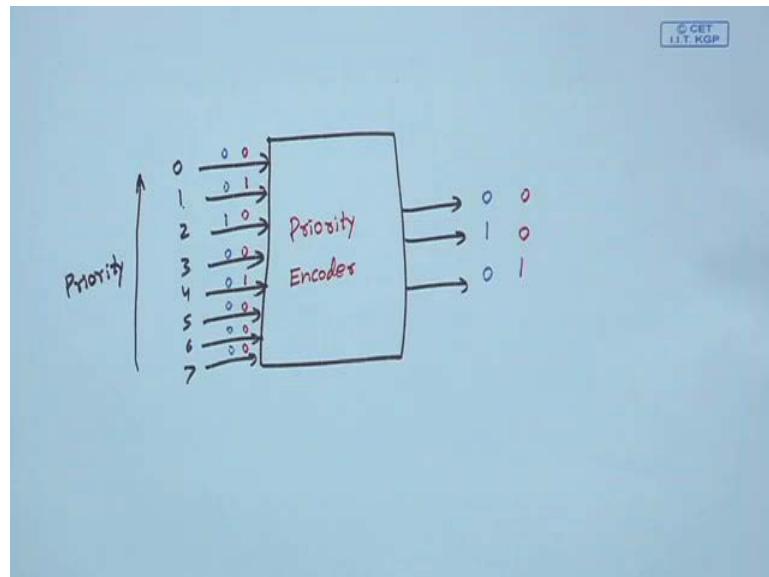
module priority_encoder (in, code);
input [7:0] in;
output reg [2:0] code;
always @ (in)
begin
if (in[0]) code = 3'b000;
else if (in[1]) code = 3'b001;
else if (in[2]) code = 3'b010;
else if (in[3]) code = 3'b011;
else if (in[4]) code = 3'b100;
else if (in[5]) code = 3'b101;
else if (in[6]) code = 3'b110;
else if (in[7]) code = 3'b111;
else code = 3'bxxx;
end
endmodule

• The inputs bits are checked sequentially one by one (in order of priority).
• “in[0]” has the highest priority.
• For simultaneously active inputs, the first active input encountered will be encoded.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES Hardware Modeling Using Verilog

Fine, this is another example of a priority encoder. This priority encoder has 8 inputs and it has 3 outputs.

(Refer Slide Time: 26:34)



So, what is a priority encoder? So, a priority encoder the example that have set, there are 8 inputs, and there are 3 outputs. And inputs are given some index value 0, 1, 2, 3, 4, 5, 6, 7. Now I am assuming that the priority increases in this fashion, my input0 has a higher priority than input7.

Now, the idea is that suppose I set the input number 2 to 1, and the others are all 0s. Then the binary equivalent of 2 which is 010 should appear in the output. Now let us take an example where any 2 of them are 1, let us say this input1 is also 1, input4 is also 1, the others are 0s. So, if more than one of the inputs are 1, then according to the priority you choose the one which has highest priority that is 1. So, binary equivalent of 1 is 001, this should be in the output. So, this is how a priority encoder works. So, let us see the description. So, the input is an 8-bit vector, this is the output. Code is a 3-bit vector of reg because it appears on the left hand side.

So, again in the always block whenever there is a change in “in” there is a begin with a nested if. If in0 means in0 is true, which means in0 is equal to 1. If in0 code equal to 000, else if in1 code is 001 simple. If in7 is 111 and if none of them are active, else I am initializing the code then to xxx. You see, the order in which I have checked the bits that will define the priority. So, if both your in1 and in4 are active, but I will get a match in in1 first. So, code will become 001. This is how priority is determined, the order in which I check the bits, right.

So, as I said the input bits are checked sequentially. So, in0 will be having the higher highest priority. And if more than one inputs are active the first input that is checked will be encoded, ok.

(Refer Slide Time: 29:26)

The slide contains the following elements:

- Verilog Code:**

```
module bcd_to_7seg (bcd, seg);
    input [3:0] bcd;
    output reg [6:0] seg;
    always @ (bcd)
        case
            0: seg = 6'b0000001;
            1: seg = 6'b1001111;
            2: seg = 6'b00010010;
            3: seg = 6'b0000110;
            4: seg = 6'b1001100;
            5: seg = 6'b0100100;
            6: seg = 6'b0100000;
            7: seg = 6'b0001111;
            8: seg = 6'b0000000;
            9: seg = 6'b0000100;
            default : seg = 6'b1111111;
        endcase
    endmodule
```
- 7-Segment Display Diagram:**

A diagram of a 7-segment display. The segments are labeled as follows:

 - a: top horizontal bar
 - b: right vertical bar
 - c: bottom horizontal bar
 - d: left vertical bar
 - e: middle-left segment
 - f: middle-right segment
 - g: middle segment
- Segment bit assignment:**

(a, b, c, d, e, f, g)
- Description:**

A segment glows when the corresponding bit of seg is 0.
- Logos and Footer:**
 - IIT KHARAGPUR logo
 - NPTEL ONLINE CERTIFICATION COURSES logo
 - Hardware Modeling Using Verilog
 - A small circular video player showing a person speaking.

Let us take another example which is also quite common BCD to 7 segment decoder. Now BCD, you know BCD is a way of encoding decimal numbers, decimal digits are 0 to 9. They can be coded in 4-bits 0000 up to 1001 that is 9. Now 7 segment display looks like this, where there means either LED's or LCD's whatever, there are 7 segments a, b, c, d, e, f, g in that order.

And depending on what you want to display, you can glow some of the segments and you can switch off some of the segments; so in this example. So, we are representing the 7-bit as a 7-bit vector in this order. So, a is the most significant bit [MSB], and g is the least significant bit [LSB]. And another assumption we are making that a segment glows whenever the corresponding bit is 0. And when the corresponding bit is 1, this segment will be off. So, the description is very simple. So, input to this circuit will be a 4-bit BCD number a digit 0 to 3, 4-bit. And output will be this 7-bit vector, 0 to 6 you call it seg. So, always whenever this BCD changes, I have a case statement. If it is 0, 0 means I have to glow all except g. So, g should be 1 all other should be 0, see g is 1 all others are 0.

Similarly for 1, b and c will glow. So, you see only b and c are 0, for 2, 2 will be like this. So, c and f will be switched off, c and f are switched off, like this you can check. So, all these digits are glowing. And the default is you are switching off all these segments. If it is, input is something other than 0 to 9, because in 4-bits you can specify any number from 0 to 15. So, if either 10, 11, 12, 13, 14, 15 comes you switch off all the segments, right.

(Refer Slide Time: 31:52)

```
// An n-bit comparator
module compare (A, B, lt, gt, eq);
    parameter word_size = 16;
    input [word_size-1:0] A, B;
    output reg lt, gt, eq;

    always @ (*)
        begin
            gt = 0; lt = 0; eq = 0;
            if (A > B) gt = 1;
            else if (A < B) lt = 1;
            else eq = 1
        end
    endmodule
```

For actual synthesis, it is common to have a structured design representation of the comparator.

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES Hardware Modeling Using Verilog

So, there is another simple example of an n-bit comparator. This is a pure behavioral description where A and B are the numbers, and the outputs are less than [lt], greater than [gt] or equal to [eq]. Here again I am using a parameter, where I am defining variable word size to be 16. So, A and B, I am defining to be 0 up to word size minus 1 [word_size-1]. So, for 16 it will be 15 to 0. And lt, gt, eq are actually this will be output reg, let me just correct. It will be output reg, fine. So, here again in the “always” I put a star, if any of the input changes.

So, I start by initializing gt, lt and eq to 0s. It is very simple. I am making a comparison like this A and B are numbers say for 16, these are 16-bit numbers. I am making a comparison if A is greater than B, I said gt to 1. Else if A is less than B, I said less than, lt to 1. Else they are equal, eq equal to 1, So it is simple. Now in actual synthesis actually we normally do not design a comparator in this way we shall see later. We typically use a structured design using by instantiating some smaller comparator modules.

(Refer Slide Time: 33:27)

```
// A 2-bit comparator
module compare (A1, A0, B1, B0, lt, gt, eq);
    input A1, A0, B1, B0;
    output reg lt, gt, eq;
    always @ (A1, A0, B1, B0)
        begin
            lt = ((A1,A0) < (B1,B0));
            gt = ((A1,A0) > (B1,B0));
            eq = ((A1,A0) == (B1,B0));
        end
endmodule
```

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

Now, another example I am showing here just to show another style of specifying this is a simpler 2-bit comparator.

So, there 2 numbers one is A0, A1 other is B0, B1, and the outputs as usual are less than, greater than, equal to. So, A0, A1, B1, B0 these are all inputs and these 3 are the outputs. Now we check always whenever one of the input changes. So, I am writing like this some assignments, this lt equal to, you see A1 and A0, this 2-bits make the first number. So, I use the concatenation operation to define the first number. So, there will be a curly bracket here let me correct this, there will be a curly bracket yes; so A1, A0 concatenation and B1, B0 concatenation, if A1, A0 is less than this.

So, can I think I have missed some brackets, let me just, good, I have just changed the brackets. If A1, A0 is less than B1, B0, if this condition is true which means true means 1. So, that 1 will be assigned to lt. If this is greater than B1, B0, if this condition is true means 1 that 1 will be assigned to gt and this equality again if it is true that 1 will be assigned to eq. See here we are exploiting the fact that for logical comparisons relational operators true is represented by 1 and false is represented by 0. So, we are directly storing the result of the comparison in the target, variable indicating less than, greater than or equal to, fine.

(Refer Slide Time: 35:23)

The screenshot shows a Verilog code editor with a yellow background. The code is contained within a red-bordered box. At the bottom of the screen, there is a blue footer bar with the IIT Kharagpur logo, the text "NPTEL ONLINE CERTIFICATION COURSES", and the course name "Hardware Modeling Using Verilog".

```
module alu_example (alu_out, A, B, operation, en);
    input [2:0] operation;  input [7:0] A, B;
    input en;
    output [7:0] alu_out;   reg [7:0] alu_reg;

    assign alu_out = (en == 1) ? alu_reg : 4'bz;
    always @ (*)
        case (operation)
            3'b000 : alu_reg = A + B;
            3'b001 : alu_reg = A - B;
            3'b011 : alu_reg = ~ A;
            default : alu_reg = 4'b0;
        endcase
endmodule
```

So, as the last example we take a slightly more complex ALU example, you see here we are concerned an ALU with 2 inputs a, b, which are 7-bits. There is an operation which is 3-bits and enable [en] which is 1-bit. And in the alu_out there is the output of the alu 8-bits and the output will be stored in another register we call it alu_reg. Now this output will be generated in alu_reg and it will be stored in here. So, you see this alu_out, we are just I said doing a continuous assignment, if enable equal to 1 then alu_reg will be transferred to alu_out otherwise it will be tri-state z, four z. And here description is very simple, whenever I mean any of the input changes on case operation, let us say 000 means add, then alu_reg equal to A plus B.

Suppose 001 means subtract A minus B. Suppose 011 means logical NOT, alu_reg equal to bit by NOT, and if it is any one of the others. So, in “case”, I can mention default, then I initialize alu_reg to 0. So, this example illustrates the combination of “assign” and “case” in always block and so on, this also generates a new combinational circuit. So, with this, we come to the end of this lecture. In this lecture we have looked at a number of different examples, various ways of using the procedural style of coding. The always block the different ways of using always block and so on.

And we shall be continuing these discussions later and look at more complex and more elaborate structures using which we can model particularly the sequential circuits and finite state machines. Till then we will have to wait for the next lectures.

Thank you.

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

Lecture - 16
Blocking / Non-Blocking Assignments (Part 1)

So, we have been talking about various kinds of assignment statements in verilog. So, recall we talked about the continuous assignment statement using assign. Then we talked about the procedural assignment statements, we looked at several examples where inside an always block or inside an initial block we can use various kinds of assignments to variables. These are called procedural kind of assignments.

Now broadly speaking these procedural kind of assignments come in two different flavors. So, in this lecture we shall be starting our discussion to clearly distinguish and try to find out when and under what conditions should we use which of these two options or alternatives.

So, the topic of our lecture today's Blocking and Non-Blocking Assignments. These are the two types that I was talking about fine.

(Refer Slide Time: 01:23)

Procedural Assignment

- Procedural assignment statements can be used to update variables of types "reg", "integer", "real" or "time".
- The value assigned to a variable remains unchanged until another procedural assignment statement assigns a new value to the variable.
 - This is different from continuous assignment (using "assign") that results in the expression on the RHS to continuously drive the "net" type variable on the left.
- Two types of procedural assignment statements:
 - a) **Blocking** (denoted by "=")
 - b) **Non-blocking** (denoted by "<=")

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



So, these assignment statements fall under the broad category of procedural assignments. So, what do you mean by procedural assignment? Procedural assignment is an

assignment and expression which assigns some value to a variable, which lies or figures inside a procedural block, in Verilog a procedural block can be either initial or always, ok.

So, these kind of procedural assignment statements can be used to update variables only of these types reg, integer, real or if you have a time variable which keeps track of time. You see you cannot have a net type variable on the left hand side of a procedural assignment. This is the restriction. So, if you are using some variable to assign a value inside a procedural block, they have to be one of these data types. Now you see there is also very clear difference from the continuous assignment type. Say in continuous assignment which we had seen earlier using assign, let us say we write assign a equal to b plus c.

Here whenever the value of b plus c changes, the value of a will be immediately changing, it will be directly driving this net type variable on the left, and this net type variable on the left of this assign statement do not have any facility or capability of storing this value. That is why we call this as continuously driven. As b and c changes, a will be continuously changing, right, but in contrast for a procedural assignment there is an important difference to be remembered. The value that we assigned to a variable this remains unchanged, until you assign some other value to that same variable again. So, there is some notion of storage or memory associated with the variables that you use here.

So, once you assign some value to a variable, the value will remain unchanged until you again assign it to some other value. So, this is the difference from assign which I just now talked about. So, broadly speaking this assignment statements inside procedural blocks can be blocking or non-blocking. And they are denoted by the assignment operator. Blocking is denoted by equal to and non-blocking is denoted by the arrow symbol less than equal to, this symbol, this means arrow, fine.

(Refer Slide Time: 04:21)

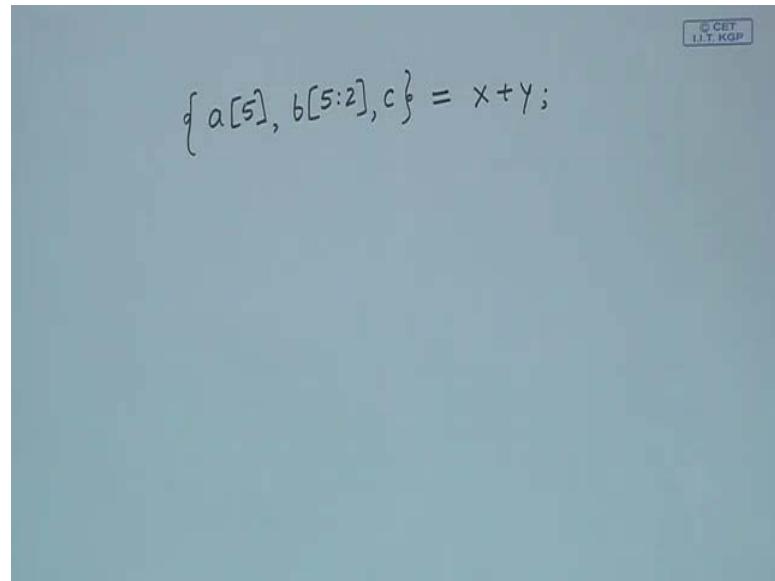
- The left-hand side of a procedural assignment statement can be one of:
 - A register type variable ("reg", "integer", "real", or "time")
 - A bit select of these variables (e.g. sum[15])
 - A part select of these variables (e.g. IR[31:26])
 - A concatenation of any of the above
- The right-hand side can be any expression consisting of "net" and "register" type variables that evaluates to a value.
- Procedural assignment statements can only appear within procedural blocks ("initial" or "always").

So, the rules for a procedural assignment statement are as follows.

So, I already talked about that the variable on the left hand side should be one of reg, integer, real or time, this of course are constrained. Not only this on the left hand side we can also have a bit select of one of the variables of this type like you can write some 15 equal to something, right. This is one particular bit of a data of this type. Or you can also select a part by specifying the index values like IR bit numbers 26 to 31 equal to something. This also you can mention or you can concatenate one or more of this by enclosing them within the curly braces as I mentioned earlier.

So, you can also use the concatenation operation like for example, you can write concatenation.

(Refer Slide Time: 05:20)



Let us say a variable $a[5]$ comma a variable b , you can have a section defined or an entire variable c . So, so you can define a concatenation like this equal to, you can have any expression like x plus y . So, these kinds of expressions are allowed provided a , b , c , these variables are one of these four types reg, integer, real or time. But the right hand side of the expression can be anything, you can have any combination of net type variable and also, register type variables. And these assignments as I mentioned can appear only within an initial block or within an always block.

So, just outside these blocks you cannot use these assignments right.

(Refer Slide Time: 06:17)

(i) Blocking Assignment

- General syntax:
`variable_name = [delay_or_event_control] expression;`
- The “=” operator is used to specify blocking assignment.
- Blocking assignment statements are executed in the order they are specified in a procedural block.
 - The target of an assignments gets updated before the next sequential statement in the procedural block is executed.
 - They do not block execution of statements in other procedural blocks.
- This is the recommended style for modeling combinational logic.

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

So let us start with blocking assignment let us try to understand what it really means. Now the general syntax is on the left hand side, you have the variable name of the types, one of the types, I mentioned, equal to, well you can mention some kind of triggering event like you can mention a delay or you can also mention an event like at some clock edge or something that also you can mention; variable name equal to some expression. So, for the time being let us ignore this. Variable name equal to some expression, this is the general syntax, and this equal to symbol is used to indicate blocking assignment.

So, what is the meaning of blocking assignment? You see, so if inside a block there are several statements, the statements will be executed one by one in the order they are specified. Like suppose inside a procedural block begin-end I have four statements, this statement will be executed one after the other in sequence. Suppose the first statement is a equal to b plus c. So, some value will be assigned to a. Second statement is let us say d equal to a plus 5. So, that value of a which was assigned that value of a will be used in the next expression. So, in that way one by one sequentially the instructions will get executed.

This is the interpretation of blocking assignment; that means, an instruction which is executing or an assignment statement which is executing, it blocks the execution of all the statements which follow. So, unless it is finished the next instruction will not start, this is the meaning of blocking assignment. So, the target of an assignment statement

will get updated before the next statement is executed, right, but if we have several blocks suppose I have two always blocks. Then a blocking type statement in one of the blocks will not block any statements in the other block. It only applied to the statements inside the same block, ok.

So, this point is mentioned, they do not block execution of statements in other procedural blocks. Now this blocking assignment style can be used to model sequential circuits also, but it is recommended that we use this assignment style to model combination logic we shall see a lot of examples later and try to find out why this is so.

(Refer Slide Time: 09:06)

The slide contains the following text and code:

- Blocking assignments can also generate sequential circuit elements during synthesis (e.g. incomplete specification in multi-way branching with "case").
- An example of blocking assignment:

```
integer a, b, c;
initial
begin
    a = 10; b = 20; c = 15;
    a = b + c;
    b = a + 5;
    c = a - b;
end
```

• Initially, a=10, b=20, c=15
• a becomes 35
• b becomes 40
• c becomes -5

This is what I mentioned. So, using blocking assignment you can also generate sequential circuit elements. Since we had seen some examples earlier, you recall those examples where you had a multiway branching using “case”, where if all the values of the case variable were not specified. Suppose I am doing a case on a variable a, which can take on the value 0, 1, 2 and 3, but I specify what will happen if the values are 0, 1 and 3, but for 2, I am not specifying. So, if the value is 2 what will happen? So, all the variables which appear on the left hand side they should not change, which means they should remember or memorize their value. So, the synthesis tool what it will do? It will be generating a latch a storage element for such variables, but if in the case statement you mention all the conditions without an ambiguity, then it will generate a pure

combinational circuit, because earlier we took some examples to explain these things, fine.

So, an example of blocking assignment is shown here, this is not the complete verilog module just a segment of the module. Let us say a, b, c are 3 variables of type integer. So, inside a initial block, I have this begin-end. So, see the statements a=10, b=20, c=15 these are the three statements, one by one they will get executed. So, initially this a, b, c will be getting the values 10 then b=20 then c=15.

Next statement is a equal to b plus c. So, b and c will be added, the value will go to a. So, it will be 35. So, the new value of a will become 35, the next statement is b equal to a plus 5. This new value of a will be used and 5 will be added to it. So, b will become 40. And the last statement is c equal to a minus b. So, the latest value of a and the latest value of b, they are subtracted and the result will be minus 5.

So, this is the interpretation of the blocking assignment, statements execute in the order they appear just like what we see in a high level programming language like c or java, fine.

(Refer Slide Time: 11:36)

```
module blocking_example;
    reg X, Y, Z;
    reg [31:0] A, B;    integer sum;

    initial
    begin
        X = 1'b0;  Y = 1'b0;  Z = 1'b1;    // At time = 0
        sum = 1;    // At time = 0
        A = 31'b0;  B = 31'habababab;    // At time = 0
        #5 A[5] = 1'b1;                // At time = 5
        #10 B[31:29] = (X, Y, Z);     // At time = 15
        sum = sum + 5;                // At time = 15
    end
endmodule
```

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

There is one example here, where various kinds of assignments are demonstrated are shown. Here we are defining three variables x, y, z and two vectors of size 32, a, b. This is not a meaningful code just an illustration. So, you say at the beginning we give some

assignments, say x equal to 0, y equal to 0, z equal to 1. These are all single bit variables and sum equal to 1, sum is an integer. Because we have not mentioned any time delay these assignments all will take place at time t equal to 0.

Similar is the case for these two assignments, a 31-bit all 0 and b 31-bit hexadecimal. We initialize it with the hexadecimal number ab, ab, ab, ab; this all they happen at time t equal to 0. Now let us say I write at time delay 5, #5 a[5] equal to 1. So, the fifth bit of a will be changed to 1 at time 5, then I give another delay 10. So, I define a segment of b these 3-bits 29, 30 and 31, they will be assigned a value which will be the concatenation of x, y and z. And this will happen after delay of 15 which means at time 15 and lastly sum, we are incrementing by 5. So, again because there is no delay here, this will also happen at time 15, ok.

This is just an example showing how the times are kept track off in blocking assignments.

(Refer Slide Time: 13:31)

The slide has a yellow header bar with the title "Simulation of an Example". Below the header, the Verilog code is divided into two parts:

```
module blocking_assgn;
integer a, b, c, d;
always @ (*) begin
repeat (4)
begin
#5 a = b + c;
#5 d = a - 3;
#5 b = d + 10;
#5 c = c + 1;
end
end
initial begin
$monitor ($time, "a=%4d, b=%4d,
c=%4d, d=%4d", a, b, c, d);
a = 30; b = 20; c = 15; d = 5;
#100 $finish;
end
endmodule
```

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the text "NPTEL ONLINE CERTIFICATION COURSES", and a portrait of a man.

Now, here we take an example of a blocking assignment, and we shall show the results through simulation because there are a few things to understand here. This is the example that we take four variables defined a, b, c, d, this is a always block. Always at the rate star means any variable changes, and inside this block we are saying repeat this block 4 times. So, I am saying this begin-end block has to be executed 4 times. And each of them will be having a delay of 5. And you see we have written a test bench which monitors the

values, the time then a, b, c and d. These values will be printed whenever some changes of these variables take place.

So, we start by initializing a=30, b=20, c=15 and d=5 and we continue the simulation, till time 100, at 100 we call finish. So, let us see when you simulate this code with this test bench what is the output, ok.

(Refer Slide Time: 14:44)

The screenshot shows a simulation interface with the following details:

- Simulation Results**: A table showing variable values over time (0 to 80). The values for a, b, and c are highlighted in red, indicating they are changing over time.
- Initially:** The initial values assigned to variables a, b, c, and d.
- Code:** The Verilog code being simulated, which includes assignments for a = b + c, d = a - 3, b = d + 10, and c = c + 1, followed by a repeat loop.
- Logos and Text:** IIT Kharagpur logo, NPTEL Online Certification Courses logo, and the course title "Hardware Modeling Using Verilog".

	a	b	c	d
0	30	20	15	5
5	35	20	15	5
10	35	20	15	32
15	35	42	15	32
20	35	42	16	32
25	58	42	16	32
30	58	42	16	55
35	58	65	16	55
40	58	65	17	55
45	82	65	17	55
50	82	65	17	79
55	82	89	17	79
60	82	89	18	79
65	107	89	18	79
70	107	89	18	104
75	107	114	18	104
80	107	114	19	104

So, I am showing here this code side by sides. So, that you can understand what is happening, you see in the test bench we initialized a=30, b=20, c=15, d=5. So, I am showing the initial values here. And this is the simulation output which you got. And on color I have shown the values which are changing ok.

So, initially at time t equal to 0, a=30, b=20, c=15, d=5 was printed. Then at time 5, this $a = b + c$ happened. So, b and c was added and a become 35 others did not change. Then again after time 5, $d = a - 3$. So, a was 32, d become 32, 35 minus 3, 32. Then again 5, $b = d + 10$, d + 10, b becomes 42. Then lastly $c = c + 1$; C was 15 it becomes 16. Now this will repeat 4 times. So, again $a = b + c$, b is 42, c is 16. So, a becomes 58, $d = a - 3$, d becomes 55, $b = d + 10$ it becomes 65 and $c = c + 1$, 17.

So, this thing repeats 4 time, third time and a 4th time. So, at the end you get results like this. So, just by simulating this code you get results like this for the variables change like

this. So, the point to note, that every time this loop is executed, you execute them statement by statement. One statement is finished then only the next statements starts.

(Refer Slide Time: 16:37)

The slide has a yellow header bar with the title '(ii) Non-Blocking Assignment'. Below the title is a bulleted list of points:

- General syntax:
`variable_name <= [delay_or_event_control] expression;`
- The "<=" operator is used to specify non-blocking assignment.
- Non-blocking assignment statements allow scheduling of assignments without blocking execution of statements that follow within the procedural block.
 - The assignment to the target gets scheduled for the end of the simulation cycle (at the end of the procedural block).
 - Statements subsequent to the instruction under consideration are not blocked by the assignment.
 - Allows concurrent procedural assignment, suitable for sequential logic.

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the text 'NPTEL ONLINE CERTIFICATION COURSES', and a circular portrait of a man.

Fine, now let us look at the second type of assignment statement namely the non-blocking assignment statement. Now non-blocking assignment is different from blocking in the way they work, and they are more suitable to be used for specifying sequential circuit behavior as you will see through many examples later.

So, first the general syntax; syntax is very similar variable name and expression, but instead of equal to we use this arrow assignment less than equal to this symbol. This operator specifies non-blocking assignment. Now as the name implies, a non-blocking assignment statement, does not block the execution of the statements which follow inside that block. The statements inside the block they can execute concurrently, this is a concept here. It is not that after a statement finishes only then next one will start, not that, it is not blocking the other statements. So, all these statements can execute together, ok.

So, the idea is this. So, if you have a non-blocking assignment, then the assignment to the target which takes place that will happen at the end of the simulation cycle of that block; that means, at the end of the block. So, if you see here the assignment to the target will get scheduled for the end of the simulation cycle means that one run of the loop, at the end of the block. So, here we will explain with an example.

So, because it is non-blocking statements, after the instruction currently we are looking at executing will not be blocked, they will all be executing together which will allow something called concurrent procedural assignment, which is very suitable for modeling sequential logic, we shall see these things.

(Refer Slide Time: 18:46)

- This is the recommended style for modeling sequential logic.
 - Several "reg" type variables can be assigned synchronously, under the control of a common clock.

```

integer a, b, c;
initial
begin
  a = 10; b = 20; c = 15;
end
initial
begin
  a <= #5 b + c;
  b <= #5 a + 5;
  c <= #5 a - b;
end

```

- Initially, a=10, b=20, c=15
- a becomes 35 at time = 5
- b becomes 15 at time = 5
- c becomes -10 at time = 5

All the right hand side expressions are evaluated together based on the previous values of "a", "b" and "c". They are assigned together at time 5.

IIT Kharagpur
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog

So, let us take an example. As I had mentioned this is the recommended style for modeling sequential logic, where a number of reg type variables can be assigned synchronously under the control of a common clock. But here it is just a simple example I am taking, here this is just a test bench I am writing, there is no clock. There were 3 variables a, b and c. So, in one initial block I am initializing them to 10, 20 and 15. And in the second initial initialize block I am using non-blocking statement with some delay identifier. So, the idea is that when there are non-blocking statement inside a procedural block.

Then all these 3 statements are executing concurrently, meaning the right hand side expressions, they will be evaluating together using the current values of a, b and c. And after the delay which is specified, they will all be concurrently assigned to the variables on the left. So, in this example, let us say a is 10, b is 20, c is 15. So, on the right hand side $b + c$, $a + 5$ and $a - b$ will be evaluated first, what is $b + c$? $b + c$ is 35. What is $a + 5$? It is 15, what is $a - b$? It is -10. So, they will be assigned all together at time 5. So, a will become 35, b will become 15, c will become -10.

So, you see the final value that a, b, c gets here is quite different from what it got when you used a blocking assignment statement. So, when you use blocking and when you use non-blocking the results are different. So, you should be careful when you are using it to model some behavior of a system or a circuit. So, as I said that all the right hand side expressions are evaluated together based on the previous values of a, b, c, not these latest values. And they are assigned all together concurrently at time 5, right.

(Refer Slide Time: 21:06)

```

always @ (posedge clk)
begin
    a <= b & c;
    b <= a ^ d;
    c <= a | b;
end

```

All assignments take place synchronously at the rising edge of the clock.

Recommended style for modeling synchronous circuits, where assignments take place in synchronism with clock.

The slide footer includes logos for IIT Kharagpur and NPTEL, and the text 'NPTEL ONLINE CERTIFICATION COURSES' and 'Hardware Modeling Using Verilog'.

So this is a more common way where we use this kind of non-blocking assignment with a clock, this statement means whenever there is a positive edge of a clock, you compute these expressions.

And assign them to these variables concurrently. So, all assignments take place synchronously at the rising edge of a clock. You see, this is quite natural from hardware point of view because in any hardware register, there is some kind of a clock signal. So, whenever there is a clock and there is a load, the input value gets loaded. So, here also something similar can take place, the right hand side can be evaluated and the values can be made available to the input of the registers, As soon as the clock comes, all those input values will get loaded in the registers that will happen exactly when the clock comes, ok.

So, that feature can be modeled using this posedge clock facility. And as I said this is the recommended style for modeling synchronous sequential circuits where there is a clock

and you are wanting assignments to take place in synchronism with the clock, right. This is the recommended style.

(Refer Slide Time: 22:20)

Swapping values of two variables "a" and "b"

<pre>always @ (posedge clk) a = b; always @ (posedge clk) b = a;</pre>	<pre>always @ (posedge clk) a <= b; always @ (posedge clk) b <= a;</pre>
<ul style="list-style-type: none">Either $a=b$ will execute before $b=a$, or vice versa, depending on simulator implementation.Both registers will get the same value (either "a" or "b").Race condition.	<ul style="list-style-type: none">Here the variables are correctly swapped.All RHS variables are read first, and assigned to LHS variables at the positive clock edge.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog | 12

Now let us take on one example, suppose we want to swap the value of 2 variables a and b. Swapping means, if a is 10 and b is 20. So, we want to make a=20 and b=10, interchange their values. Let us look at various styles of doing it, and let us see that whether swapping actually takes place or not, ok.

This is the first example I am taking using non-blocking assignment with 2 always blocks. Both are activated at the posedge of clock, there is no delay. So, at time $t = 0$, just immediately after the clock edge comes, this will be executed. So, you see because these are concurrent blocks $a = b$, and $b = a$, they are actually executing concurrently. But you see when you are simulating this, depending on the simulator, simulator will be either doing this assignment first and then this or the reverse. So, if $a = b$ is done first, then b will be going to a and that value of b will again be copied to b. So, both a and b will be getting the value of b, that was there. But if this statement executes first, then the value of a is get copied to b and that same value remains in a.

So, both the registers will finally get this same value, but it can be either a or b, you do not know. It depends on the simulator how this simulator actually schedules these 2 operations executed, execution. And this is called a race condition. Depending on the relative order of execution the final result can be either a or it can be b. Let us look

another example. Same one instead of blocking I am using non-blocking assignments. Say here there is no problem, because these assignments will take place only when positive edge of clock comes and they will take place together. So, the right hand side will be evaluated.

So, right hand side b, right hand side a, that values will be taken, and they will be assigned to a and b at the posedge of the clock. So, there is no question of indeterminism like in blocking assignment that was happening here. Because here there is nothing that says that whenever the clock comes only then to be assign, blocking says it will be done one by one. It can be either this or this or this first this or this depending on the order of execution. So, if you use non-blocking statement then the variables will be correctly swapped, because the right hand side variables are read first, let us say a = 10, b = 20, they will be read first. And whenever the positive clock edge comes, 20 will be assigned to a and 10 will be assigned to b. So, swapping will actually take place.

(Refer Slide Time: 25:30)

Trying to swap using blocking assignment

```
always @ (posedge clk)
begin
    a = b;
    b = a;
end
```

• Both "a" and "b" will be getting the value previously stored in "b".

```
always @ (posedge clk)
begin
    ta = a;
    tb = b;
    a = tb;
    b = ta;
end
```

• Correct swapping will occur, but we need two temporary variables "ta" and "tb"

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

Let us take another example. Suppose we are using blocking statement to try to swap. Suppose I write $a = b$, $b = a$, what will happen? Inside a same block, just try to see what will happen here. First this statement will be executing, b will be copied to a and then this will be executing, a will be copied to b. So, both a and b will be finally getting the value of b. Because this is the first statement to execute, the value of b if it is 20, 20 will be copied to a.

So, a will also become 20. So, 20 will also be copied to b again. So, both a and b will be 20, the previous value of b. So, using blocking statement if you want to really do a swap, you can do it like this, by using two additional temporary variables ta and tb. You first assign a to the temporary variable ta, then b to tb then tb to a and ta to b. In this way swapping will correctly take place, but of course, you see, you have to require, you use two additional variables. So, this example shows you that using non-blocking assignment for certain application can be very convenient, as this swapping example is one of them it shows you, fine.

So, let us similarly simulate one example using non-blocking assignment; same example that is showed for blocking, but here I use non-blocking assignment. So, my test bench is also similar, but since here we use a clock. So, I am applying a clock here. So, what you are doing?

(Refer Slide Time: 27:26)

Simulation of an Example

```

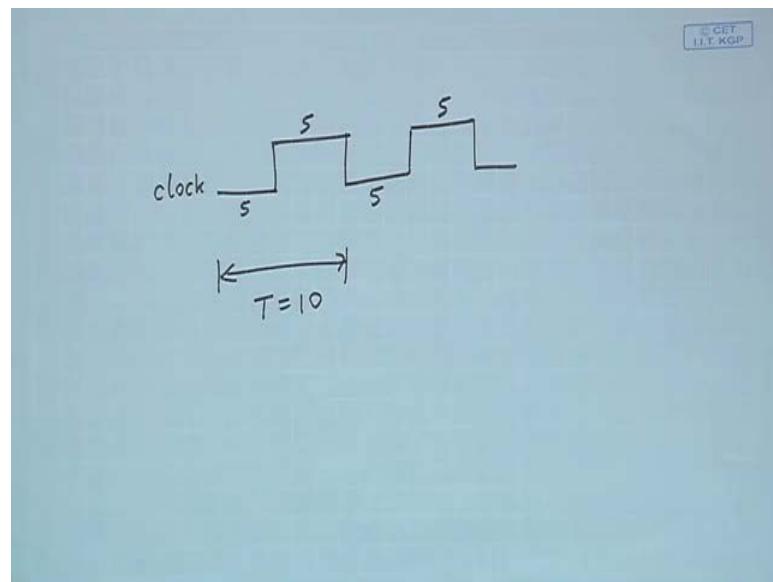
module nonblocking_assgn;
integer a, b, c, d;
reg clock;
always @ (posedge clock)
begin
    a <= b + c;
    d <= a - 3;
    b <= d + 10;
    c <= c + 1;
end
initial
begin
    $monitor ($time, "a=%d, b=%d, c=%d, d=%d", a, b, c, d);
    a = 30; b = 20; c = 15; d = 5;
    clock = 0;
    forever #5 clock = ~clock;
end
initial
#100 $finish;
endmodule

```

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

We are using the monitor to just mention what are the variables to print. Then we initialize the variables, clock also is initialized to 0. And in a forever loop we are generating the clock signals. Forever is a loop which goes on indefinitely, we say that after a delay of 5, you do $\text{clock} = \sim\text{clock}$. So, the clock signal will be generating like this.

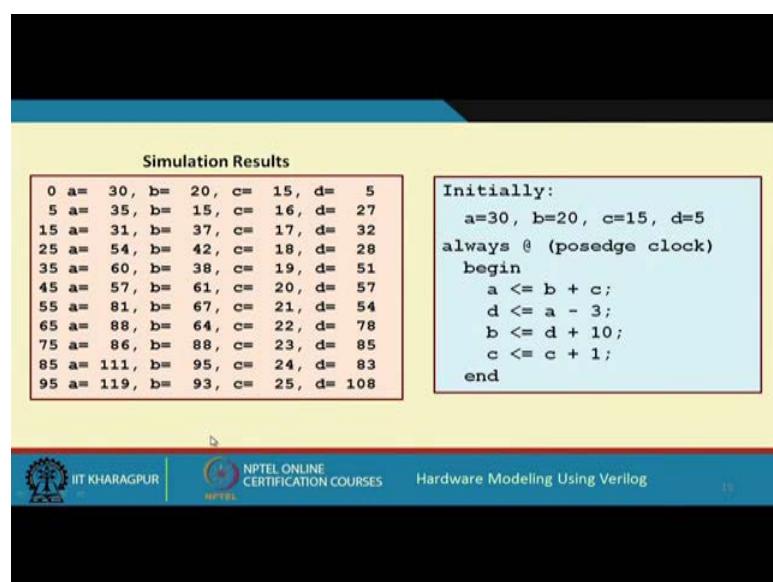
(Refer Slide Time: 27:53)



Clock was initially 0.

So, after a delay of 5, it becomes 1, after delay of 5 it becomes 0. After delay of 5 it again becomes 1 like this. So, a clock will be generated with a time period of 10, right. So, this forever statement generates this clock and we continue simulation till 100, there is another initial block for you say at 100, you finish. Let us see the simulation output for this example.

(Refer Slide Time: 28:31)



So, I have shown this side by side for convenience, initially $a=30$, $b=20$, $c=15$, $d=5$ that is the first value which is printed at time $t = 0$. Then just this is the statement, positive edge of the clock is coming after time 5, clock was 0. So, after time 5, the first clock edge comes, right.

So, at time 5, the first change will take place, the positive edge comes at time 5. So, $b + c$ will be assigned to a , $a - 3$ will be assigned to d , $d + 10$ to b and $c + 1$ to c , what is $b + c$? $b + c$ is 35, $a - 3$ is 27, $d + 10$ is 15, $c + 1$ is 16. So, all the values are assigned together, you see 35, 15, 16, 27. Now next clock positive edge will be coming after the time period 10. So, next one change will happen at 15.

So, again you do this with these new values, $b + c$, $15 + 16 = 31$; $a - 3 = 32$; $d + 10 = 37$; $c + 1 = 17$. You see all these values are assigned parallelly 31, 37, 17, 32 and these repeats up to the time 100. So, 95 is the clock edge that you get d for 100, last change will take place at 95. So, this is how this simulation result shows.

(Refer Slide Time: 30:10)

Some Rules to be Followed

- It is recommended that blocking and non-blocking assignments *are not mixed* in the same “always” block.
 - Simulator may allow, but this is not good design practice.
- Verilog synthesizer ignores the delays specified in a procedural assignment statement (blocking or non-blocking).
 - May lead to functional mismatch between the design model and the synthesized netlist.
- A variable cannot appear as the target of both a blocking and a non-blocking assignment.
 - $x = x + 5;$
 - This is not permissible → $x = x + 5;$
 $x <= y;$

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

So, there are a few rules which you should remember when you are using this kind of blocking and non-blocking statements in a procedural block.

And these are summarized here you see, the first thing I told you is that blocking statement is the recommended style for combinational circuits, for specifying combinational logic. And non-blocking statement is a recommended style for specifying

synchronous sequential circuit. So, the first thing to remember is that you should not mix blocking and non-blocking assignments together in the same block. Well, it is not that it is not allowed, I shall show you some examples later where you will show that what will happen if you mix them. But it can make the things complicated for you. Simulator or the synthesizer may allow, but this is certainly not a good design practice, you should avoid this. You use either blocking or non-blocking, inside a particular procedural block.

The second thing you remember is that this delays that you show or you specify, these are only for simulation. So, when you are doing synthesis. So, all these delays will be ignored. So, the design which is working perfectly in simulation, may not work exactly, exactly as you just want them to work in the final synthesized hardware. So, that is why you should keep these things in mind. Secondly, say a particular variable, you cannot have it as the target of a blocking as well as non-blocking assignment together in two blocks in the same module. If you do it the simulator or the synthesizer will give you a warning on error message. So, with this we come to the end of this lecture, where we had looked at the mainly the differences between the blocking and the non-blocking assignment styles.

So, we shall be continuing our discussion on these two styles over the next few lectures, because this is very important in modeling. And unless you understand clearly the differences between the two and what will happen if various types of combinations you use. It will be difficult for you to understand the interpretation of the semantics of different structures. So, we shall be continuing with this in the next lecture as well.

Thank you.

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

Lecture - 17
Blocking / Non-Blocking Assignments (Part 2)

So, in this lecture, we shall be looking at some of the examples where we will be using both Blocking and the Non-Blocking styles. And we shall see some of the features and some of the issues that might happen or that might arise.

So, this is our part 2 of this lecture, fine.

(Refer Slide Time: 00:40)

Introduction

- We shall be looking at some examples of modeling using blocking and non-blocking assignments.
- Objective is to get a feel of the type of assignment statement to use for some particular scenario.
- Avoid some of the “not-so-good” design practices in modeling.

↳

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

So, as I said, we shall be looking at some actually small examples. Now why we are doing? that our objective is to try to give you a feel that for what kind of design, which kind of assignment statements would be better suited and there are a few practices which might lead to some errors which are best avoided. These are or this may be regarded as not so good design practices.

So, while we discuss the various styles and we look at the different examples, we shall see that some of the design styles may lead to some constructs which are very easily confused by the designer that the designer may very easily insert some error, means in advertently in the design. It is very easy to do, so, if you are not extremely careful. So,

there are some constructs which are best avoided like one, I have already told you in the last lecture that you should avoid using both blocking and non-blocking assignment within the same always or the same initial block.

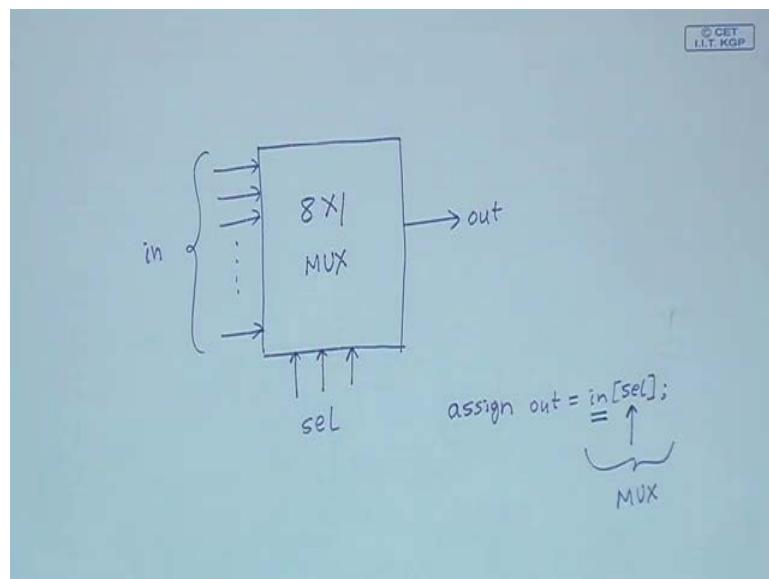
(Refer Slide Time: 02:04)

```
// 8-to-1 multiplexer: behavioral description
module mux_8to1 (in, sel, out);
    input [7:0] in;    input [2:0] sel;
    output reg out;
    always @(*)
        begin
            case (sel)
                3'b000: out = in[0];
                3'b001: out = in[1];
                3'b010: out = in[2];
                3'b011: out = in[3];
                3'b100: out = in[4];
                3'b101: out = in[5];
                3'b110: out = in[6];
                3'b111: out = in[7];
                default: out = 1'bx;
            endcase
        end
endmodule
```

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

So, the first example, we take is that of a multiplexer, this is a 8 line to 1 line multiplexer, this is the behavioral description of it. Now just to tell you by 8 line to 1 line multiplexer, what is this schematic we are looking at?

(Refer Slide Time: 02:27)



Yes, we are looking at a 8 to 1 multiplexer whose output is “out”, there are 3 select lines which we call is “sel” and there are 8 input lines, the input lines are called “in”. So, depending on the select line; one of the input will be selected. Now earlier we had seen that we can very easily model a multiplexer using an assign statement like we can write assign out = in[sel].

So, if we use a vector on the right hand side with a variable as the index, this will generate a multiplexer, this is what we mentioned. But here we are going a little bit into the behavior of it, but instead of using assign statement, we are trying to use a procedural block to model a multiplexer. So, how we do it, we are using an always block. So, let us see here the parameters are in, sel and out, in is the input, there are 8 lines, select line. There are 3-bits and output, see this output we are declaring as reg because inside this always block we are assigning some value to this out.

So, I mentioned inside the procedural block, the left hand side has to be either reg or an integer or a real or a time variable. So, here we have defined it as type reg. Now here we are saying that always @(*), whenever some in values change, input values change either in or sel, you do this; it is a simple case statement on sel, here we are just enumerating all 8 binary combinations 000 up to 111. If it is 000, then in0 is selected, it will be assign to out, if 001, then in1 will be assign to out and so on.

If it is 111, in 7 will be assign to out. Now you see at the end, we have given a default case which says that we are initializing some undefined value to out. Now you may ask that we have already defined all possible 8 combinations here in the case. So, why do we need to specify default? So, you remember that in Verilog, we mentioned when we talked about variables and their values that Verilog is actually a 4 valued logic modeling system.

So, every variable can assume a value not only 0 and 1, but also x and z, due to some problem in the test bench or the circuit, which is driving maybe you have not initialized some variable in a proper way, this select line may be coming here as an x value. So, what will happen. So, it will not match with any one of 000. 001 up to 111. So, it is a different value x. So, for those cases it will go to the default option. So, it will get matched to the default and the output will also be said to undefined x, in that case, right.

(Refer Slide Time: 06:15)

The screenshot shows a Verilog code editor with a yellow background. The code is a Verilog module for a synchronous up-down counter:

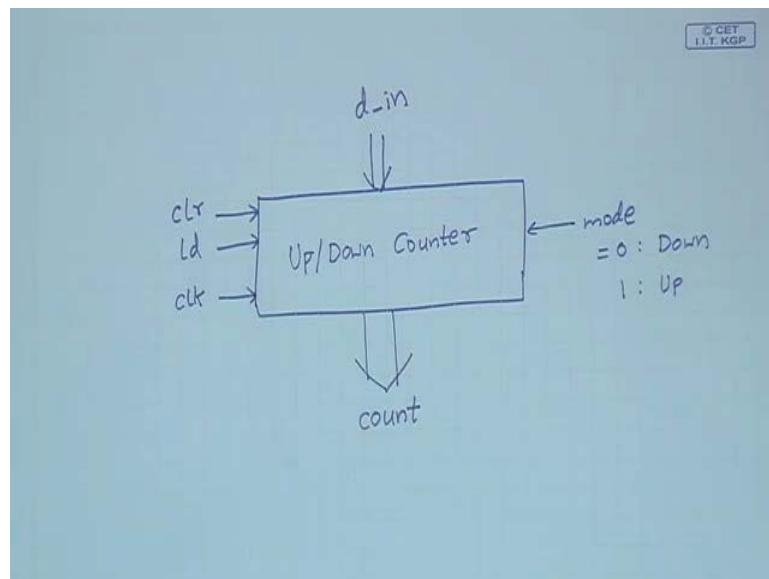
```
// Up-down counter (synchronous clear)
module counter (mode, clr, ld, d_in, clk, count);
    input mode, clr, ld, clk;
    input [0:7] d_in;
    output reg [0:7] count;

    always @ (posedge clk)
        if (ld)          count <= d_in;
        else if (clr)    count <= 0;
        else if (mode)   count <= count + 1;
        else             count <= count - 1;
endmodule
```

Below the code, there is a navigation bar with the IIT Kharagpur logo, NPTEL Online Certification Courses logo, and the title "Hardware Modeling Using Verilog". To the right of the title is a circular portrait of a man.

So, this is a multiplexer, the next example we take is a synchronous up down counter.
Now let us again look at what kind of a circuit we are talking about.

(Refer Slide Time: 06:30)



Here we are talking about a circuit which is a up-down counter which can count up or which can also count down. Here the count value which is the output, this we are calling as count. Now we can initialize the value of the counter. There is another input called din, there is an input called clr, I can clear the counter to 0. I can load the counter with this input value din, if I want and of course, there is a clock, these are my signal values in

this counter. So, I can clear it to all 0s. So, I can load it with any value I want and if I apply a clock it will count.

And there of course, there is another input called mode; mode will tell whether I am going to count up or count down. If mode equal to 0, it means I want to count down, if mode equal to 1, it means I want to count up, right. This is the circuit which you want to design or model in Verilog. Let us see how we have done it. So, the first thing we have done it is that we have defined all the parameters mode, clear, load, data in, clock and count. This four are the input signals, din also input, but din is a vector 8-bit counter, I am assuming 8-bits and count is also an output, which also the reg because we are assigning count, we have defined as reg also 8-bits.

And here we are doing everything in a synchronous way load, clear, everything is synchronous, this will happen at the positive edge of the clock. So, whenever there is a positive edge of the clock, we first check whether the load input is 1 or not if the load input is 1, then the value is loaded from din. Next we check if clear is 1; if clear is 1, then count value is initialized to 0 or else we check if mode is 1 or 0. If mode is 1, we increment the count by 1, up count, if mode is 0, we decrement it by 1, down count. So, this shows you the behavior of the up-down counter. So, you see using this kind of non-blocking statement, we can model this counter in a very convenient way, well.

Well of course, here the statements are not executing concurrently because of the if then else, exactly one of them will be executing, but there could have been more statements also.

(Refer Slide Time: 09:32)

The slide has a yellow header bar with the text "Parameterized design:: an N-bit counter". Below this is a white text area containing Verilog code for a counter. To the left of the code, there is explanatory text and a bulleted list. At the bottom, there are logos for IIT Kharagpur and NPTEL, and the title "Hardware Modeling Using Verilog" next to a portrait of a man.

Make a design general for any number of bits.

Using the keyword "parameter".

- Parameter values are substituted before simulation or synthesis.

```
// Parameterized design:: an N-bit counter

module counter (clear, clock, count);
    parameter N = 7;
    input clear, clock;
    output reg [0:N] count;

    always @(negedge clock)
        if (clear)
            count <= 0;
        else
            count <= count + 1;
endmodule
```

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

Next, let us look at a parameterized design, parameterized design we mentioned earlier we can define some kind of a constant call a parameter like here we define, say here we are trying to design an N-bit counter, but N can be anything. We are specifying the value of N by this parameter $N = 7$, but inside my program I am using N everywhere, well here just in one place. So, the count register I am declaring of size N. So, if $N = 7$, means I am actually declaring a 8-bit register.

One more than this; 0 to 7 because N will be 1 less than that. So, actually, it will be $N + 1$, not exactly N, 1 more than that. So, the declaration is very simple. So, always just assuming that the counter will be count at the negative edge of the clock. At the negative edge of the clock, there is a clear, if clear, you clear the count or otherwise you increase the count by 1, right. So, using this parameter, you can create a general design where you can only change this one single line and your entire design can become a instead of an 8 bit counter, it can become a 16-bit counter, right.

(Refer Slide Time: 11:00)

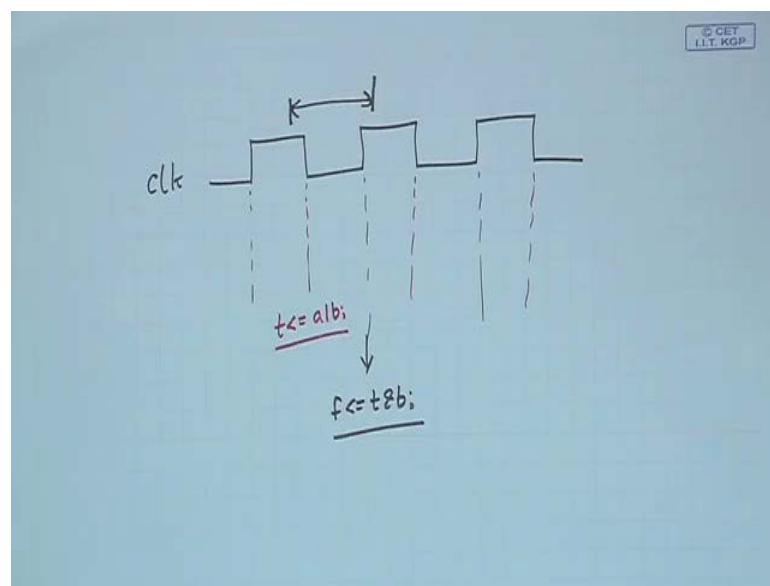
The screenshot shows a slide from an NPTEL online course titled "Hardware Modeling Using Verilog". The slide contains a Verilog code example:

```
// Using multiple edges of the same clock
module multi_edge_clk (a, b, f, clk);
    input a, b, clk;
    output reg f; reg t;
    always @ (posedge clk)
        f <= t & b;
    always @ (negedge clk)
        t <= a | b;
endmodule
```

The slide also features the IIT Kharagpur logo and the NPTEL logo.

This example shows that you can have a module where you can use more than 1 clocks; there can be very complex designs, I means in such complex designs you may be using not only 1 clock.

(Refer Slide Time: 13:29)



But more than 1 clock. So, 1 clock signal may be used to control one part of the circuit the other clock signal may be used to control some other part of the circuit and those two parts may be executing concurrently under the control of the two clocks, the two clocks may be of different frequencies also, right. So, in Verilog even inside a single module,

you can do this kind of modeling very easily. Here I am assuming that the two clocks are the clk1 and clk2 and a, b, c, these are inputs and f1, f2 are two outputs, declared as reg. So, I can use and this is just a very simple example for illustration, I can use two concurrent always block, let us say one of them is triggering at the positive edge of clk1 other one is triggering at the negative edge of clk2.

So, these are working concurrently; one of them is assigning the value a and b to f1 and the other one is assigning b xor c; the exclusive OR to f2 and they are happening concurrently in synchronism with two different clocks, the clocks may be of different phases, different frequencies. So, there is no restriction there. So, you can have a very general design like this where multiple signals can be used to synchronize the operation of your design, this is possible. Not only that you can also use the two edges of the same clock, multiple edges of the same clock, you can use to carry out some operations.

Like in this example, I am using a single clock signal clk, a, b are the inputs, this output reg, this f is an output and t is another temporary, I am assuming. Just see what you are doing here, here let us say I have a clock signal, a clock signal is coming like this let us say, these are the positive edges and the negative edges, I am showing with different color; these are the negative edges. Now let us look at this specification, this specification says that always at the positive edge of the clock you do $f \leq t \& b$. So, at the positive edge, let us say here you are doing $f \leq t \& b$, this statement is executing and at the negative edge you are doing $t \leq a | b$; let us say here at the negative edge you are doing $t \leq a | b$.

So, you see what is happening, here in this, in this f statement, $f \leq t \& b$; you are using the value of t and t is getting assigned by this statement which is happening at this clock. So, you can say that I am doing some computation which is starting here and it is continuing till here, I am using two edges. So, at the first edge; I am computing a value t and at the second edge, I am using that value t to compute some other final value f and this will go on repeating; right. So, this is one way or one technique using which we can actually carry out two operations within a single clock period, we can do something in the rising edge of the clock, something else in the falling edge of the clock.

(Refer Slide Time: 15:34)

```
// Another example
module multi_edge_clk (a, b, c, d, f, clk);
    input clk;
    input [7:0] a,b,c,d;
    output reg [7:0] f;
    always @ (posedge clk)
        c <= a + b;
    always @ (negedge clk)
        f <= c - d;
endmodule
```

- Two operations are carried out every clock cycle.
 - "c" is assigned at the rising edge.
 - "f" is assigned at the falling edge.
- It is assumed that addition or subtraction can be completed in half a clock cycle.

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

And one of the values may be fed to the input of the other computation like in the example I showed just now, right. So, this is quite possible. So, let us take another example this is an example we shows some addition and subtraction operations, similar with this multiple edge clock that same kind of a thing. Here we are using let us say this $a + b$, $a - b$, here of course, the input description is not completed, let us just make it complete. Let us say input a , b , clk , let us make another declaration, let us say we define input, let us say we have 8-bits and we define a , b , c and d all of them. Let us declare like this a , b , c , d , all are 8-bit inputs; let us say and this f is a reg, t is also reg.

t is not required here of course, c , let us make it t now finds, fine. This t is not required. So, t you can forget, fine. So, here you see at the positive edge you are doing $a + b$ is assigning to c . So, whatever value was there in c that gets modified and at the negative edge of the clock you are using that value of c subtracting with d and you are storing it into another value f , right. So, in this f will also be a vector, sorry, this will also be a vector like this, fine. So, here at the rising edge you are doing some computation, at the falling edge you are do some computation.

So, you can do two computation in the same clock cycle; one addition and one subtraction, if your clock cycle time is large enough. So, basically; what I am saying is that you are carrying out two operations in every clock cycle in these always loops, in the first positive edge of the clock c is assigned a value, the sum of a and b and at the other

one at the falling edge of the clock, f is assigned a value. This is the subtraction of c and d. So, here we assume that our clock cycle time is large enough. So, that this addition and subtraction can be completed within half a clock cycle. For this kind of a scenario we can use these multiple edges of the same clock to activate two operations, all right.

(Refer Slide Time: 18:24)

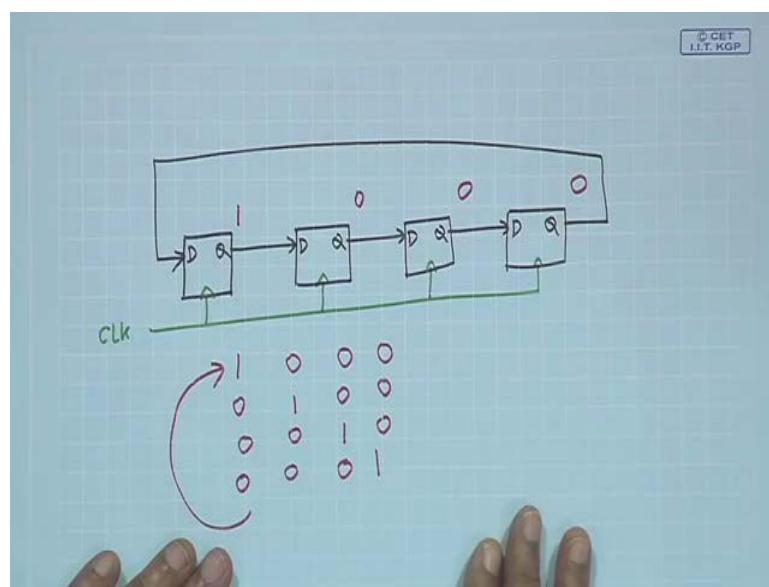
```
// A ring counter
module ring_counter (clk, init, count);
    input clk, init;
    output reg [7:0] count;
    always @ (posedge clk)
        begin
            if (init) count = 8'b10000000;
            else begin
                count = count << 1;
                count[0] = count[7];
            end
        end
endmodule
```

This solution is wrong.
 • count[7] will get overwritten in the first statement.
 • Rotation of the bits will not happen.

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

So, let us now look at the example of a ring counter. Now I mean what is a ring encounter, just to recall ring counter is nothing, but a shift register.

(Refer Slide Time: 18:32)

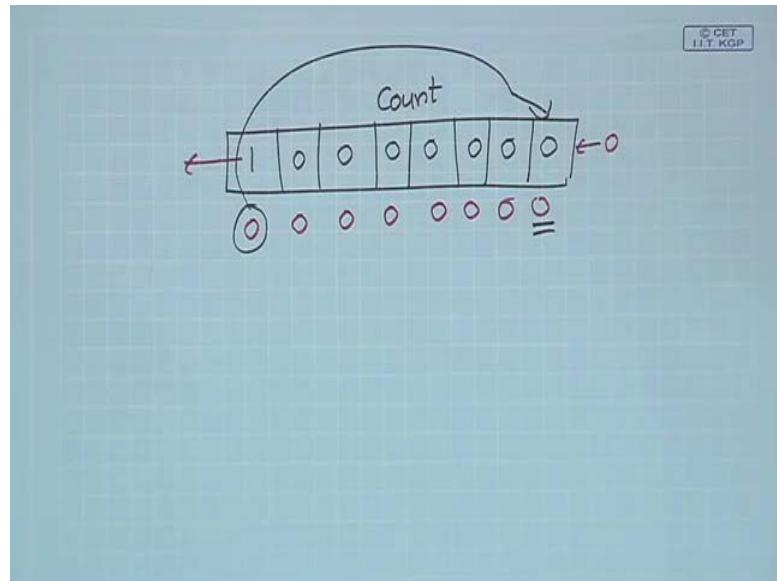


So, I am showing a 4-bit ring counter, it is just a shift register comprising of several d flip-flops. So, the output of one flip flop is connected to the input of the other, ring counter means they are connected as a ring, as a chain and this counter is initialized typically to the state 1000 and the clock signal is applied to all the flip-flops together, this is the clk. So, if you initialize this shift register with 1000 and if you go on applying the clk. So, what will happen initially it was 1000 after 1 clk, it will be rotating right by 1 place.

So, this 1 will be shifted here. So, this 0 will come back, 0 will come here, next clock this 1 will come here, this 0 will again come back, 0010, next clock 0001, then again it will go back to 1000. So, this will repeat, right, this is the function of the ring counter. So, here in this design we are showing you an 8bit ring counter. So, we have a clk, we have a init, init signal which will be initializing the ring counter to a single 1 and all 0s and the output of the counter we just count. So, clk and init are the inputs and count is the output which is reg.

So,. So, every time when there is a clock edge; posedge, we are checking that if init is active, init is 1, well if init is 1, we are initializing count to this 1000; you see here, we are using blocking assignment statement, else begin count = count << 1, then count[0] = count[7]. Now tell me whether this is a correct description of a ring encounter. So, these two statements, do they actually do the shifting correctly; just see count is an 8-bit variable; right.

(Refer Slide Time: 21:25)



So, let me just show you; count is an 8-bit register; there are 8 flip-flops. So, I have 1 here and there are 7 0s. This is my count, now in this code what we are doing, in this begin-end we are first shifting count left by 1.

So, if you shift it left by 1; this 1 will go out and 1 will disappear and shift left by default will feed as 0 on the right. So, the counter will become all. So, after the shift like this 1 will disappear and then you are saying $\text{count}[0] = \text{count}[7]$ to make the rotator, but this $\text{count}[7]$ has already become 0. So, even if you store this 0, here this value will still remain a 0, this 1 will not come back. So, means after a rotate operation after this begin-end block, the count value will become all 0 and after that it will remain all 0.

So, this is not a correct description of a ring counter. So, as I said, this solution is wrong, this $\text{count}[7]$ will get overwritten in the first statement itself, it will become 0 and the rotation will not happen.

(Refer Slide Time: 22:56)

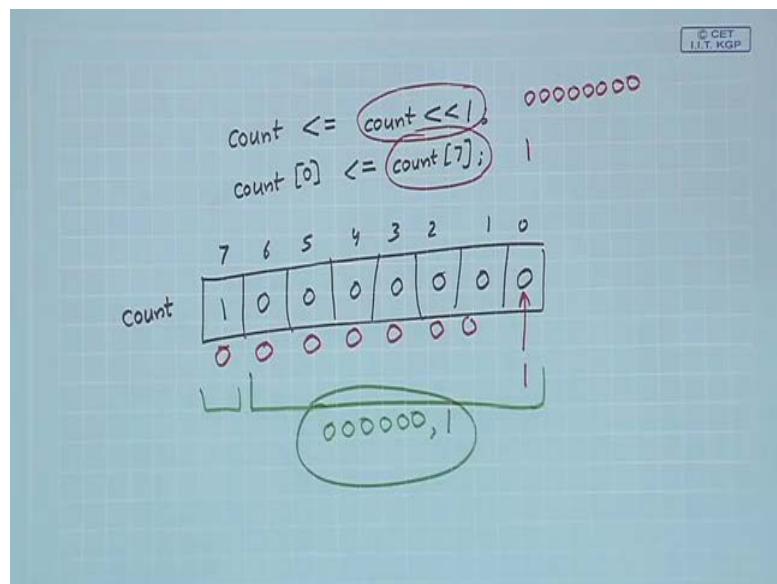
```
// A ring counter (Modified version 1)
module ring_counter (clk, init, count);
    input clk, init;
    output reg [7:0] count;
    always @ (posedge clk)
        begin
            if (init) count = 8'b10000000;
            else begin
                count <= count << 1;
                count[0] <= count[7];
            end
        end
endmodule
```

• This is the correct version.
• Since non-blocking assignments are used, rotation will take place correctly.

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

So, how I can rectify this error; it is very easy, I change the blocking to non-blocking see here what will happen, here we are doing the same thing, but we are using a non-blocking assignment.

(Refer Slide Time: 23:17)



Let us see; what will happen. So, now, our statements are the first statement says $\text{count} \leq \text{count} \ll 1$ and the second statement says $\text{count}[0] \leq \text{count}[7]$. Now again let us look at the count, there is an 8-bit register what will happen let us see. Suppose initially my register contains this, this is my count and these are the index values 0, 1, 2, 3, 4, 5, 6

and 7. Now according to the rule of the non-blocking assignments, the right hand sides will be evaluated first concurrently.

So, if you do a count less than less than 1; what will happen? if you do a count less than less than 1? This was the count. So, it will become all 0, but count[0]; this count[7]; count[7] is 1. Now you are assigning them together. So, when you are just assigning them, these counts[7], 1; this is also bit getting stored. This will be assigned to count 0. So, this 0 will become 1 and the remaining bits will become 0. This is how it is interpreted. So, actually the rotation will take place correctly here, fine. So, here this is the correct version rotation will actually take place.

(Refer Slide Time: 25:15)

```
// A ring counter (Modified version 2)
module ring_counter (clk, init, count);
    input clk, init;
    output reg [7:0] count;
    always @ (posedge clk)
        begin
            if (init) count = 8'b10000000;
            else
                count = {count[6:0], count[7]};
        end
endmodule
```

This is a correct way of modeling using blocking assignment.

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

Now in fact, even without using this non-blocking we can have a correct version like this. Suppose I write my blocking assignment like this, I write count = {count [6:0] and count[7]}, what does this mean count[6:0] means you see this part count[6:0], bit number 6 to bit number 0, you take this first these are all 0s; 6 0s concatenate with count[7]; count[7] is this, it is 1, you concatenate this with 1, take this whole thing together. So, 1 has already come here, you assign this to count 0, 0, 0, 0, 0, 0, 1 which is what it should be rotate right, rotate left, fine.

So, this will also work correctly. So, you see that using either blocking or non-blocking assignment, if you know what it means, what this statements do and how they execute, you will be able to find out that whether or not your model is correct or not because you

may see that you have written something, but well simulation you see that your result of the output is not coming correctly. So, you will have to interpret why it is so, that; what is the problem or what is the error in your code; you will have to understand. What is the meaning of the blocking assignment; how they execute; how they change the values, then only well be able to debug and come up with a correct version of the code. So, with this we come to the end of this lecture.

Now, in the next lecture, we shall be continuing with some other aspects of blocking and non-blocking assignments, till then you can refresh your memory, I mean, I strongly suggest; whatever I am covering, I am discussing, you try to work them up yourself, you try to run them at least on the simulation platform and get a feel of, if you make some changes; what is the impact that will get on the output only, then you will be able to learn the language and will be able to get a confidence on the language constructs that are available.

Thank you.

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

Lecture - 18
Blocking / Non-Blocking Assignments (Part 3)

So, we continue with our discussion on the Blocking and Non-Blocking Assignments. So, in this lecture, we shall be looking at some more differences in the way they behave and the way they synthesis or the simulation tool will interpret or handle the description that you have given or provided in either blocking or non-blocking kind of assignments. So, we shall be illustrating these points through a number of examples that will be the best way to just explain.

(Refer Slide Time: 00:56)

Blocking vrs. Non-blocking Assignments

- We shall now illustrate some examples that show how the modeling style influences the simulator or synthesizer to capture the behavior of the modeled circuit.
 - Very important concept required to be clearly understood by the designer.
 - Even a slight error in modeling can result in a drastically different circuit.
- Highly recommended:
 - For any confusion, write a Verilog code, simulate it and analyze the output(s).

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

So, actually through the examples, we shall be showing you some modeling styles; some modeling styles using both blocking assignment and also using non-blocking assignments.

And there we will try to understand by looking at this semantics. So, the meaning of the statements, how this simulator or the synthesis tool might be influenced to capture the behavior of a model in a particular way; that whether it is actually capturing the behavior in the way you are intending or it is doing something else. So, as these are something which is extremely important from the point of view of the designer because you have to

understand this blocking and non-blocking statements very clearly because if you do not model the behavior of a circuit or system that you want in the proper way.

Then even a very small error in the model which might easily be overlooked that might result in a drastically different output or a drastically different circuit depending on whether you are simulating or synthesizing. So, we shall be trying to see some examples. Now there is another thing, I just mean, I mentioned repeatedly that when you are studying some codes, when you are learning some constructs of a language. So, it is always recommended that you actually run the code say for Verilog, you simulate it and see how the outputs are coming, change the code and see what are the changes that are happening.

(Refer Slide Time: 02:50)

Example 1

```
begin
  a = #5 b;
  c = #5 a;
end
```

- The value of "b" will be assigned to "c" 10 time units after the "begin ... end" block starts.

```
begin
  a <= #5 b;
  c <= #5 a;
end
```

- "a" is scheduled to get the value of "b" 5 time units into the future.
- "c" is also scheduled to get the value of "a" 5 time units into the future.

Value of "c" will be different for the two cases

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

So, the first example that I take is a simple timed assignment using blocking and non-blocking, a very simple segment of code. So, inside a block begin-end block, this can be inside either always or initial whatever some procedural block, I write blocking assignment after delay of 5 b, $c = \#5 a$ and here similar, but non-blocking. So, let us try to understand what will happen here, blocking statements will be executing one after the other, first statement will say that after a time delay of 5, b will be assigned to a, second statement says after this is finished, only then, it will come to the second statement. So, after another time delay of 5 that value of a will be copied to c. So, the value of b which was there it first goes to a and then it goes to c.

So, the value b will be finally assigned to c after a delay of 10 time units, right, this will happen here. Now if you use non-blocking assignment statement, what will this mean? This will mean here the time delay does not mean that you first finish this after time 5, then again after time 5 do this, no, it will mean that you evaluate the right hand sides, see a scheduled to get the value of b, 5 time units into the future, right hand sides are evaluated together when you say hash 5 and hash 5. It does not mean like this at first this 5 and then this 5, this is a 10, not this, both of these are waiting for time 5 and they are assigning, they are executed concurrently, right. So, here b will be assigned after time 5 to a; a will be also assigned after time 5 to c.

So, you see in the first case here the value of b was assigned to c, but here finally, the value of a will gets assigned to c, right. So, the final value of c will be different for these 2 cases. This is a very simple example, but you will have to understand the difference very clearly what this hash 5 here and what this hash 5 here means. So, when you are using a blocking assignment, I am repeating if you are using those delays because in a blocking assignment statement, this statements will be executed one after the other, their delays will get accumulated, if you give 5, 5, 5, it will 5, 10, 15, 20 that way the delays will get accumulated, but in a non-blocking case, it is not.

So, there the delay means you evaluate all the right hand sides together and after how much time you will be assigning it to the left hand side, if you give all then hash 5, hash 5, hash 5, means after delay of 5 all of them will be assigned together, it is not that the delay will be adding up, not like that. So, because of that for this 2 cases, the final value of c was coming to be different.

(Refer Slide Time: 06:35)

Example 2

```
always @ (posedge clock)
begin
    q1 = a;
    q2 = q1;
end
```

Parallel Flip-flops

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

Let us take another example. This is procedural block with a clock, always @(posedge clock), here I am using a blocking assignment. So, what does this block mean, this block means that whenever there is a positive edge of the clock, you activate or start executing this, what is this saying, that you first assign a to q1, then assign q1 to q2, right.

So, in terms of the hardware, if you give it to the synthesis tool; what synthesis tool it will generate? You see a is assigned to q1, just think a flip-flop, a will be assigned to q1. So, whenever the positive edge of the clock comes, a will be assigned to q1 and it will be stored in the d flip-flop. The first statement goes and after this is done, this q1 will be assigned to q2, because you see this is not that q1 will be connected to this, because I mean you will have to understand what is the meaning of non-blocking. The synthesis tool and also the simulation tool they know what is the meaning of the blocking assignments, blocking assignments means these statements are supposed to be executed one after the other, after the first statement executes that value will be used to execute the second statement, one by one sequentially.

So, if you keep that sequential thing in mind then you see the first statement says q1 = a, this is fine, it will be mapped to this flip-flop, q1 will be a, but after this is completed only then the second one will start. So, this q1 already has received the value of a. So, that a is supposed to go to q2. So, the synthesis tool will understand that meaning of this blocking assignment. So, instead of connecting q1 and it will connect a directly to this

second flip-flop also to generate q. So, it will be 2 parallel flip-flops after the clock finally, both q1 and q2 will be getting the same value a; a will be stored here as well as a will be stored here.

As this meaning shows, this a will go to q1; q1 will go to q2, same a will come, right. So, in terms of the hardware it will be two parallel flip-flops. Let us look at a slightly different, you see this was my description, I simply interchange the order.

(Refer Slide Time: 09:30)

Example 3

```

always @ (posedge clock)
begin
    q2 = q1;
    q1 = a;
end

always @ (posedge clock)
begin
    q1 <= a;
    q2 <= q1;
end

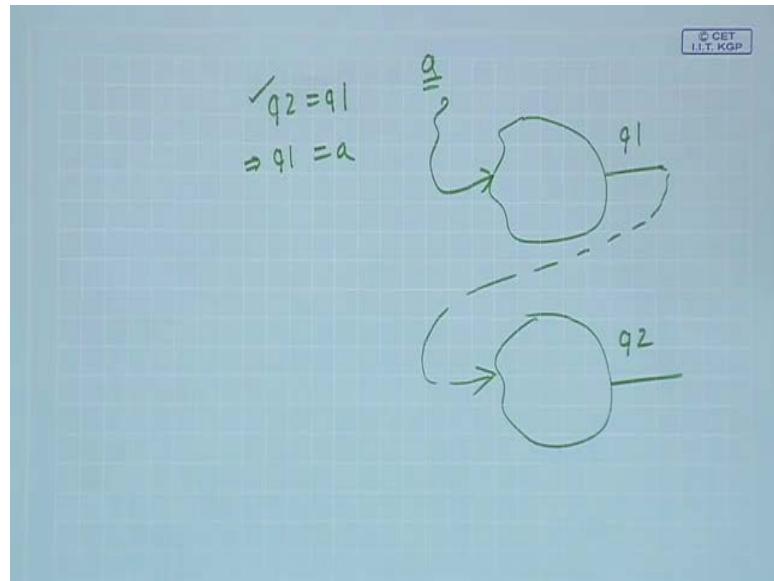
```

Shift Register

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

I just write it like this and below I write a description, same description using non-blocking, non-blocking assignment. Here let us try to understand what will happen; here we have written $q2 = q1$ and $q1 = a$; what does this mean?

(Refer Slide Time: 09:57)



You see, you can imagine, there were two flip-flops, q_1 will be the output of one of the flip-flops, q_2 will be the output of the other flip-flop. First one you are saying that q_1 will be going to q_2 which means this q_1 has to be connected to the input of this, q_1 will have to go to q_2 .

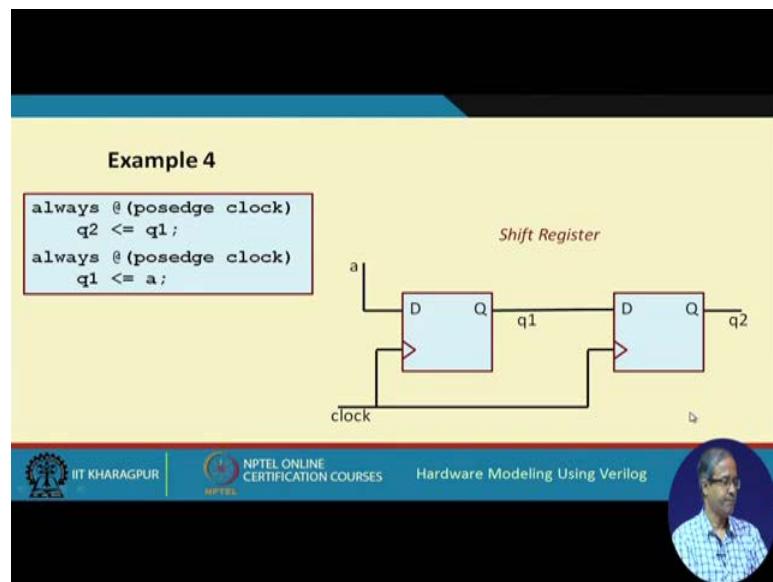
Secondary we are saying some other vary because after this is completed only then you will come here, a will go to q_1 . So, some other value a ; a will go to q_1 . So, the old value of q_1 will go to q_2 . This a will be going to q_1 . You see what we are trying to say is nothing, but a shift register, if you write it in the other way around by interchanging the order; what you are saying is just a 2-bit shift register, q_1 will be shifting to q_2 and then this a will be shifting into q_1 , right. So, the same thing will happen for the non-blocking case also, Non-blocking means the same thing, q_1 will go to q_2 , a will go to q_1 , the order is not important because in whatever we just order you mentioned.

So, the hardware that will be synthesized well be a 2-bit shift register, q_1 will go to q_2 , a will go to q_1 . So, here also a goes to q_1 , a goes to q_1 , q_1 goes to q_2 , q_1 goes to q_2 . So, the previous values of a and q_1 , previous value of a , previous value of q_1 , they will be assigned to q_1 and q_2 . This will be assigned to q_1 , previous value of q_1 will be assigned to q_2 . So, both of these descriptions model a shift register. So, one thing you understand for a blocking case it is very peculiar. So, if you write the 2 statements in a particular

order then it will generate 2 flip-flops in parallel, but if you simply interchange them it will become a shift register.

So, this you will be able to understand only if you know what is the meaning of the blocking assignments, if there are some statements what exactly is the meaning and how they will execute and what will be the final result, right.

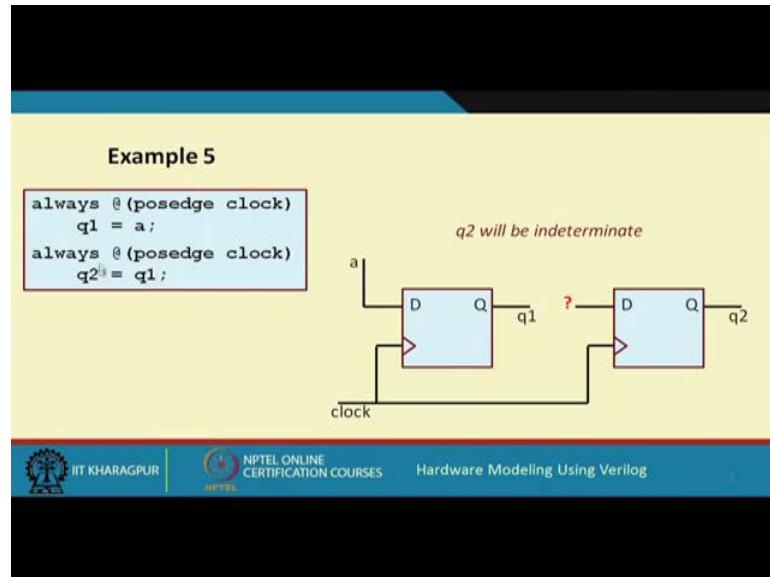
(Refer Slide Time: 12:28)



So, let us continue, let us take another example, here there are two blocks, two procedural blocks, both on positive edge, here you are saying at the positive edge q1 will go to q2, also the positive edge, a will go to q1. Just if you move back in the previous slide you see here we used a single procedural block, but you are doing the same thing, at the positive edge, a was going to q1, q1 is going to q2.

But here we are meaning the same thing maybe we have using two different blocks, but both are activating in positive edge and the assignment will take place in a very similar way. So, here also you will be generating the same shift register, right. So, this is fairly simple. So, if you just put them in this order then also you can generate a shift register. But if you put them, the same code instead of non-blocking if you put blocking what will happen?

(Refer Slide Time: 13:27)



Let us understand; here there are two different always blocks. So, in the earlier case, we said that the two statements are inside a single always block which means they were executing one after the other.

But here I am saying; there are two different always blocks, they are executing concurrently maybe there are blocking assignments, but $q1 = a$ and $q2 = q1$; both will be executing together whenever posedge of the clock comes. So, now, you see; what will happen; well, $q1$ will go to $q2$ is fine, but a will be going to $q1$. Now in terms of the hardware just see. So, what we are trying to say is this first statement says, a will go to $q1$; that means, a will go to $q1$, second statement says $q1$ will go to $q2$. So, some $q1$ will go to $q2$, both at the positive edge, but the value of $q2$ will be indeterminate.

Because what will be this $q1$ here, will it be the old value of $q1$ or this new value of $q1$ which you have just now loaded into $q1$ from a , this description is a little ambiguous. If you use blocking statements here, this description is ambiguous and you should avoid this because simulator may also confuse; this $q2$ will be, $q2$ will become indeterminate because it is simulator first execute this. So, whatever is the old value of $q1$ it will be assigned to $q2$, then a will be assigned to $q1$, but if its executed this first, then the new value of a will be assigned to $q1$ and that $q1$ will be assigned to $q2$.

So, it will act as shift register. So, the interpretation of the meaning may not be unique depending on the order of the execution that will be scheduled by this simulator. The

synthesis tool will also give you a warning that there is a confusion and the output value will be indeterminate. So, you should. So, the crux is that you should try to avoid clock triggered assignment on blocking statement as far as possible. Whenever you want to carry out assignment with clock triggering, better to use non-blocking then all these problems will not occur, fine.

(Refer Slide Time: 16:24)

Example 6

What circuit will the synthesis tool generate?

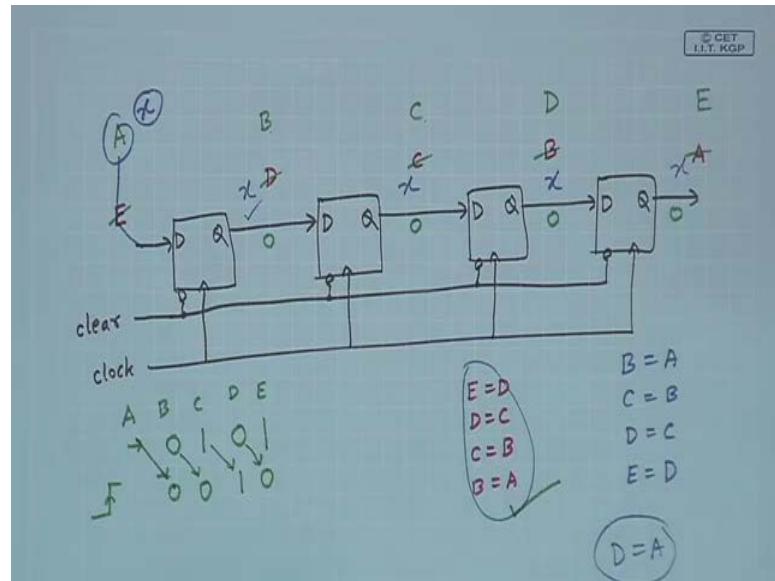
```
module shiftreg_4bit (clock, clear, A, E);
    input clock, clear, A;
    output reg E;
    reg B, C, D;

    always @ (posedge clock or negedge clear)
        begin
            if (!clear) begin B=0; C=0; D=0; E=0; end
            else begin
                E = D;
                D = C;
                C = B;
                B = A;
            end
        end
    endmodule
```

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

Let us look at another example which is slightly more complex. And it will bigger; well here our objective was to design a 4-bit shift register, suppose this was what we started to design.

(Refer Slide Time: 16:41)



So, we wanted to design a 4-bit shift register like this; there 4 flip-flops; the output of one flip-flop will go to the input of the second flip-flop. So, this is what we are trying to model let us say and of course, there will be a clock; there will be a clock signal which will be feeding in parallel to all the flip-flops and of course, there will also be a clear input, clear will be active 0. So, whenever it is 0 it is activated. So, I am showing it as a bubble. Suppose, this is what I was trying to model and let us say we came up with a Verilog code like this.

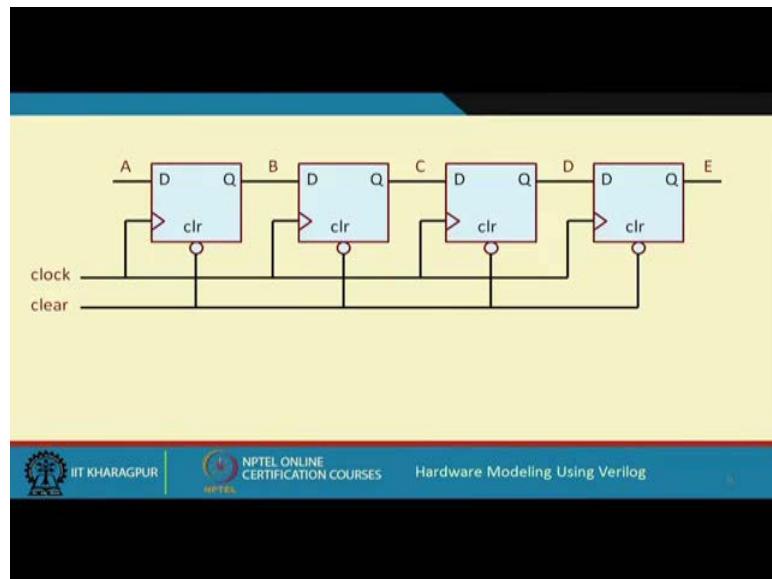
Let us see whether this Verilog code means actually models are shift register like this or not, right, let us see. Here as I said our parameters are clock, clear, A and E. So, what is our A and what is our E? This E is our input and A is our output you see A, the input and E is the output, fine E is the input and A is the output. So, actually what we want here is that let us say we want to shift register when we wanted to call these as E, we wanted to call this A and the intermediate point; let us call this as B, C and D. So, you can either do it like this or you can do it in the previous way that will be easier I think let us keep it in the previous way that will be easier. Let us call this E, let us call this A, then this will be B, this will be C, this will be D.

So, it really does not matter. So, let us just referred back in the previous one; let us see. So, here the intermediate points B, C, D; we are declaring as reg. Now in the block what we are doing? We are saying, always @ (positive edge clock or negative edge clear). So,

whenever either there is a clock going high or a clear is becoming 0, you enter this block if clear is 0, if not clear in this begin-end block, you clear B, C, D, E to 0s. So, there are four flip-flops B, C, D, E. You initialize them to all 0s, 0000, so, whenever clear is 1, clear is 0, not clear is 1.

Else clock has come you are trying to do something, you are writing four things, you are writing $E = D$, you are writing $D = C$, you are writing $C = B$ and you are writing $B = A$. So, if you write like this, what is meaning $D = E$ means, whatever is D you are copying to E; $D = C$, so, whatever is C, you are copying to D; $C = B$, whatever in B, you are copying to C; whatever B, B you are, A you are copying to B, right.

(Refer Slide Time: 20:48)



So, this is like the shift register thing. So, this is the shift register which will be trying to model like this, right and this is the description, $E = D$ means.

So, in this order, the statements will be executed, these are all, these are all blocking assignments; first D will be assigned to E. So, in a shift register what happens, suppose the shift register my bits were 0, 1, 0, 1. Now if I apply a clock; what will happen? a clock comes; this will be shifted right. So, this 0 will come here; this 1 will come here, this 0 will come here, this 1 will come out and a new 0 will come in. So, you see D equal to E, this is E, this is D, this is C, this is B; $D = E$ whatever was D; this goes to E, this is the first step, then C goes to D; whatever is C, it goes to D, then B goes to C; whatever is B; it goes to C; then A is the input; A is the input from outside, A goes to B.

So, A goes to B. So, it is actually shift register, first, we are shifting this, then you are shifting this, then you are shifting this, then you are shifting this; this is correct behavior of a shift register. So, when you are actually trying to model a shift register, this order is correct. So, I think here let us correct this. It is actually A should be input and this E should be output. This is correct description. This A is input and E is output. This is correct, fine. So, this is a correct way of modeling a shift register like this. Now suppose we make a small change; just these four assignments which are there, we simply reverse their order, we just write B = A first, C = B next, D = C next and E = D last.

So, we just reverse the order of the procedural assignments. The remaining thing is unchanged; just these four statements, we reversed. See, means ultimately you may think that well, we are actuated to a shift register. So, whether you specify that first and then this or this first and then this, should be the same, it should not matter, but suppose you do it like this, then what will happen? Let us see, well, this A is coming from outside, right, A is the external input which is coming here. Now you first say; you first saying B = A. So, whatever is A that is first coming to B. So, A, suppose the value of A was x let us say.

So, x will come here, then this statement will execute; B is going to C. So, B has already got x, that x will go to C, this x will go to C; then D = C, C is got x; x will go to D; x will go to D; E = D, that x will go to E. So, whatever was the value of A is directly going to E. So, it is not exactly like a shift register it is working because the earlier case it was not so.

(Refer Slide Time: 23:46)

The slide is titled "Example 6a". It contains two sections of text: "We just reverse the order of the procedural assignments." and "What circuit will the synthesis tool generate?". Below this is a Verilog code block:

```
module shiftreg_4bit (clock, clear, A, E);
    input clock, clear, A;
    output reg E;
    reg B, C, D;
    always @ (posedge clock or negedge clear)
        begin
            if (!clear) begin B=0; C=0; D=0; E=0; end
            else begin
                B = A;
                C = B;
                D = C;
                E = D;
            end
        end
    endmodule
```

The footer of the slide includes the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the title "Hardware Modeling Using Verilog".

So, the previous value of D was going to E then C was going to D, then B was going to C, then A was going to B. So, A was not overwriting everything. So, if I change the order; whatever new value I have applied to A that will be overwriting B, then C, then D, then E; all of them will become x.

So, actually the synthesis tool will also do this analysis and it will find out well actually what you are meaning is $D = A$. See, the synthesis tool whatever you specify, it will always try to do some optimization even if you specify a function, it will do some minimization and try to get the best circuit.

(Refer Slide Time: 25:48)

The slide contains a list of bullet points explaining the effect of sequential assignments:

- The effect of the assignment made by the first statement ($B = A$) is immediate.
- Thus, B changes, and the updated value is used in the second statement ($C = B$).
- The updated value of C is used in the third statement ($D = C$).
- The updated value of D is used in the fourth statement ($E = D$).
- The statements execute sequentially.
 - But at the same time step of the simulator.
 - The four statements are equivalent to a single statement that assigns A to E .

On the right side of the slide is a logic diagram of a flip-flop. It has an input A , an output Q , and three control inputs: $clock$, clr (clear), and set . The $clock$ input is connected to the D input of the flip-flop. The clr input is connected to the set input of the flip-flop. The set input is connected to the Q output of the flip-flop.

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

Here also, you have provided some specification, but the tool does some analysis and finds that well what you mean is actually $D = A$, right. So, instead of a shift register, what it will be generating is nothing, but a single flip-flop, whose $E = A$, not $D = A$. So, the A will be going directly to E .

So, the intermediate things are not generated. So, although you might have wanted a shift register to be generated, but the synthesis tool had applied some intelligence and found out that the way you have specified using blocking statements, it is not really a shift register specification, you are directly just assigning the input to the final output. So, it will be generating a single flip-flop, right. So, whatever I have said is mentioned here, effect of assignment; A goes to B , B goes to C , C goes to D , D goes to E , so because they are executed sequentially. The final result is A will be going to E . So, synthesis tool will be generating a single flip-flop like this, right.

(Refer Slide Time: 26:50)

The slide has a yellow header section containing the title 'Example 6b' and a bulleted list of recommendations for modeling sequential circuits. The main content is a Verilog module definition:

```
module shiftreg_4bit (clock, clear, A, E);
    input clock, clear, A;
    output reg E;
    reg B, C, D;
    always @(posedge clock or negedge clear)
        begin
            if (!clear) begin B<=0; C<=0; D<=0; E<=0; end
            else begin
                E <= D;
                D <= C;
                C <= B;
                B <= A;
            end
        end
endmodule
```

The footer of the slide includes logos for IIT Kharagpur and NPTEL, and the text 'NPTEL ONLINE CERTIFICATION COURSES' and 'Hardware Modeling Using Verilog'.

So, see if you had done the same thing using non-blocking assignments, then irrespective of the order in which you give the statements, it will still be a shift register. These four statements you can do any permutation; you can bring the first statement third; last statement first; any order you specify, but the meaning is same. All the right hand sides will be evaluated at the same time, read and they will be assigned at the same time. So, shifting will be shifting; it will not matter, depends on in which order you are just putting or writing the statements in the block. So, here if you are using non-blocking statements, the statements can appear in any order, still it will be a shift register.

So, as you can see; as I mentioned earlier that for this kind of clocked sequential circuits, non-blocking assignment is a much safer option because here is one big example. That even if you just interchange these lines by mistake; still your final circuit will be a shift register, it will not change. It will be a right hand side expressions are evaluated all in parallel because they are evaluated in parallel. The order is not important. They will finally, go to E, C will finally go to D and so on. The old values will go to this, it does not depend on the order in which put these statements. So, it will generate a shift register like this.

So, this is what I am emphasizing repeatedly that whenever you are modeling a synchronous sequential circuit, it is always a very good practice to use non-blocking assignments. So, that the errors like the one, I try to highlight can be avoided.

So, with this we come to the end of this lecture.

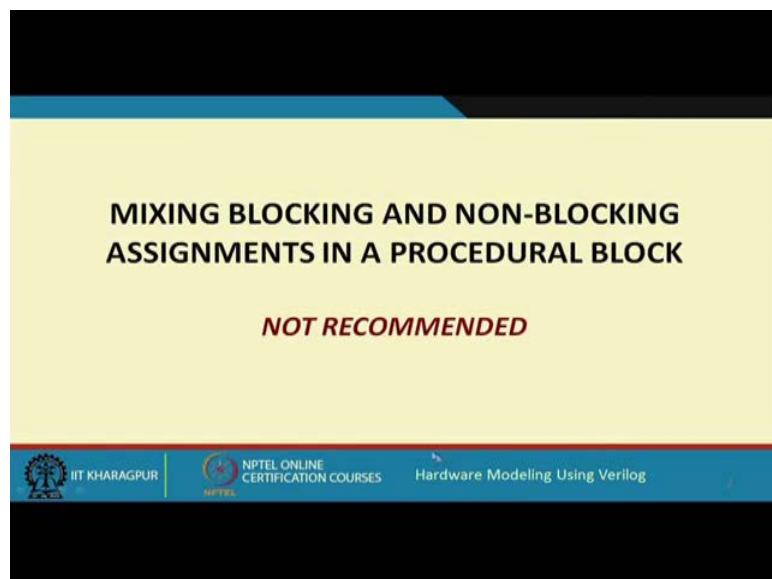
Thank you.

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

Lecture - 19
Blocking / Non-Blocking Assignments (Part 4)

So, in this lecture we shall be looking at some features of Blocking and Non-Blocking Assignments which are not recommended to be used. Like mixing both blocking and non-blocking statements in the same procedural block, this is the first thing. We shall be looking at and then we shall be talking about a feature in a verilog, there is a statement called generate with which you can automatically and dynamically generate code in verilog. So, we shall see these features in this lecture, let us see.

(Refer Slide Time: 01:02)



So, the first thing that we will be looking at very briefly because this is not recommended I shall not go into the detail this is mixing blocking and non-blocking assignments within an insider single procedural block. This I am not going into the detail because this is a very bad design practice, ok.

(Refer Slide Time: 01:27)

Basic Idea

- It is possible to combine both blocking and non-blocking assignments in the same procedural block (viz. "always").
 - Simulator or synthesis tool supports this type of usage.
- However, interpretation of the circuit behavior under such mixed usage is not very straightforward.
 - Not recommended for designers.
 - We shall explain the semantics using two simple examples.
- Such mixing should be avoided in a design.

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

So, the idea is that the simulation and the synthesis tools that are available, this support using both blocking and non-blocking assignments inside this same always our initial procedural blocks. But again like the examples that you took in the last lecture there we looked at only blocking assignment still then we saw that there are a lot of confusions, the order of the statements that you put that becomes very important which is not so for non-blocking, but if you mix them up the problem becomes even more complicated, right.

So, so if you mix them up the interpretation exactly what the circuit is supposed to behave like is not very straightforward and hence it is not recommended. So, we shall be looking at two very simple examples, and I am repeating such mixings should be avoided in an actual design, ok.

(Refer Slide Time: 02:31)

Example 1

```
always @(*)  
begin  
    x = 10;  
    x = 20;  
    y = x;  
    #10;  
end
```

```
always @(*)  
begin  
    x = 10;  
    x = 20;  
    y <= x;  
    #10;  
end
```

- Same result will be shown for both the versions:
 - "x" will be assigned 10 and then 20, both at time 0.
 - The value of "x" at time 0 will be assigned to "y".

x = 20, y = 20

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

So, let us take a simple example like this. In the first example we are using only blocking type assignments, always @(*), x = 10, x = 20, y = x and then a delay of 10. Here and this is repeating, and here the last one we are using a non-blocking assignment. You see for both this case, what is the meaning blocking assignment means, you first execute the first statement, 10 is assigned to x; then execute the second statement, 20 is assigned to x; then execute the third statement, 20 is assigned to y. So, this x will get 20 and finally, y will also get 20, both x and y will get 20. Now in the second case you see the first two are blocking, third one is non-blocking. So, the blocking statements because we are not given in a delay, they will be executing at time t equal to 0. So, at time t equal to 0 x will be getting 10, and then x will be getting 20 in this order.

But both will happen at time t equal to 0 only. So, the final value of x will be 20. And this non-blocking assignment will take or evaluate the right hand side at time t equal to 0. And at time t equal to 0, this x has already become 20 because the time has not advanced, time is still t equal to 0 inside this loop. So, that latest value of 20 will be taken and the same 20 will be assigned. You see this is actually quite confusing, you should not mix it like this, but if you use this you will see, if you simulate this you can see that it is getting, x = 20, y = 20, same value we are getting, right. So, this is one example.

(Refer Slide Time: 04:44)

Example 2

```
always @(*)  
begin  
    x = 10;  
    y = x;  
    x = 20;  
    #10;  
end
```

```
always @(*)  
begin  
    x = 10;  
    y <= x;  
    x = 20;  
    #10;  
end
```

- Same result will be shown for both the versions:
 - "y" will be assigned 10 at time 0.
 - The final value of "x" will be 20.

x = 20, y = 10

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

And the other example is that slightly different example $x = 10$, $y = x$, $x = 20$. And here the middle statement we have changed it to a non-blocking assignment. Here let us see, the first one, let us try to interpret, these are all blocking assignments, first x gets 10, then 10 gets assigned to y , then 20 gets assigned to x .

So, the final value of x will be 20 and the value of y will be 10. But here what will happen, here you see the order is important. The blocking assignment will interpret is that well once everything before me has finished only then I will start. So, x will get the value 10, then it will wait because there is something before it. So, this 10 will be assigned to y , then this will come, 20 will be assign to x . So, this is again very confusing.

Now this simulator interprets it like this if you simulate it using Iverilog, you will see that you simulate both the designs will be getting the same result $x = 20$ and $y = 10$, but I am repeating you should not use this kind of mixing inside a procedural block. So, you will see some examples later you will see that you can have multiple always blocks. So, in one of the always blocks you can use only non-blocking assignments, in some other or always block you can use only blocking assignments. Such things are sometimes required in design. We shall see later when you talk about designing sequential circuits, designing finite state machines and so on, fine.

(Refer Slide Time: 06:47)

Generate Blocks

- “*generate*” statements allow Verilog code to be generated dynamically before the simulation or synthesis begins.
 - Very convenient to create parameterized module descriptions.
 - Example: N-bit ripple carry adder for arbitrary value of N.
- Requires the keywords “*generate*” and “*endgenerate*”.
- Generate instantiations can be carried out for various Verilog blocks:
 - Modules, user-defined primitives, gates, continuous assignments, “initial” and “always” blocks, etc.
 - Generated instances have unique identifier names and can be referenced hierarchically.

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

So, we shall not be discussing any more on this. Now we come to a very useful feature that is available in Verilog which is called “generate”. Let me try to give you a brief motivation what is this all about. Well, we took an example of a ripple carry adder earlier if you recall. So, a ripple carry adder is constructed by simply connecting a number of full adders in cascade. The carry_out of a full adder is connected to the carry_in of a full adder, the carry_out of the second is connected to the carry_in of the third and so on.

So, if I want to design a 4-bit ripple carry adder I will be connecting 4 such full adders. Now if I want an 8-bit carry look ahead adder, I will be connecting 8 such. Suppose I tell you I wanted to design a 32-bit ripple carry adder then using the way which you have learned so far, you will have to instantiate the full adder 32 times. Isn't it? 32 copies of full adders are required and they will be connected like that you will have to mentioning the carry values also, carry_out will be carry_in of the next one.

So, 32 lines of code you have to write. Now you may think, so isn't it quite logical that the language should provide a feature of some kind of iteration or loop, so I can say that I want 32 copies of full adder. I specified only once and 32 copies will be generated. So, this generate block does exactly that, it dynamically creates multiple copies of your designer specification whatever written in Verilog to suit a particular design. So, we shall be looking at some examples, generate blocks. So, there is a statement called generate, generate is a keyword. Generate statements as I mentioned, this allows the simulator or

the synthesizer to generate Verilog code dynamically before they can be used. This is a very convenient feature to create so called parameterised module descriptions.

So, I mentioned this parameter earlier you can define a constant. So, you can define an N-bit ripple carry adder for arbitrary value of N. This is an example of a parameterised design. N is a parameter, you change the value of N, you get a 4-bit adder, 8-bit adder or 16-bit adder, whatever you want just change the value of N, but the Verilog code remains the same, you need not have to change the code. The generate block will automatically do it for you, right. So, there are two keywords you require “generate” and “endgenerate”.

So, what generate block does is it creates some kind of instantiation of Verilog blocks, number of times as you want. Now this Verilog blocks that you can repeat or you can generate, they can consist of modules, user driven primitives, gates, continuous assignments, initial block, always block anything almost any kind of blocks, you can use for this kind of repetitive generation. And also we shall see through some examples that when you create say for example, multiple copies of the full adder.

So, each copy of the full adder will be given some name, so that you can refer to the signals in for example, full adder number 5, the sum of the full adder number 5, you can refer to by a name that name dot sum, which will mean that will be sum of that particular full adder, we will see how to do it. So, the generated instances will have some unique identifier names which you can refer. And these names are typical, typically hierarchical.

(Refer Slide Time: 11:22)

- Special “*genvar*” variables:
 - The keyword “*genvar*” can be used to declare variables that are used only in the evaluation of generate block.
 - These variables do not exist during simulation or synthesis.
 - The value of a “*genvar*” can be defined only in a generate loop.
 - Every generate loop is assigned a name, so that variables inside the generate loop can be referenced hierarchically.

So, in order to use generate you need some special variables which tell you how many times to repeat or generate, these are called “genvar” type variables. The keyword “genvar” is used for the purpose.

So, you can use this keyword to declare such variables, which are used only for the generation of the blocks. These variables will be ignored during simulation or synthesis. These variables will only be required or used during the generation of the code, right. And these genvar variables they are defined only within a generate loop and can be used there and means I mentioned earlier. That you can have a hierarchical naming convention, that every generate loop you define, you can assign a name to it. And that name by default can be used to refer to the variables of the generated copies. We shall see some examples how to do it, but these are the basic ideas.

(Refer Slide Time: 12:44)

The screenshot shows a slide titled "Example 1" containing Verilog code. The code defines a module `xor_bitwise` with parameters `N = 16`, inputs `a` and `b`, and output `f`. It uses a `genvar p` and a `generate` block to create `N` instances of a `xorlp` block. The `xorlp` block performs a bit-by-bit exclusive OR operation between `a[p]` and `b[p]`. The `endgenerate` and `endmodule` statements close the definitions.

Below the code, there is another module definition:

```
module generate_test;
reg [15:0] x, y;
wire [15:0] out;

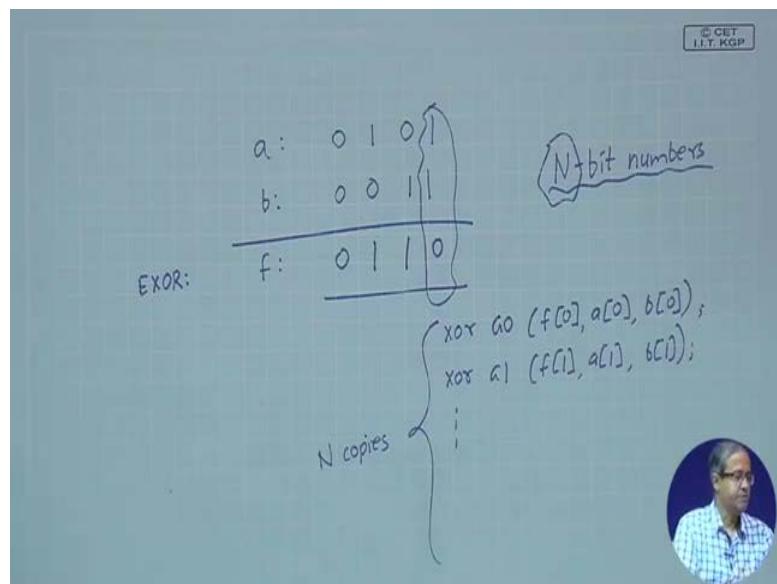
xor_bitwise G (.f(out), .a(x), .b(y));

initial
begin
$monitor ("%b, %b, %b", x, y, out);
x = 16'haaaa; y = 16'h00ff;
#10 x = 16'h0f0f; y = 16'h3333;
#20 $finish;
end
endmodule
```

The slide also includes the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and a portrait of a professor.

So, let us straight way look at an example, this is the best way to think. Here we are taking a very simple example, a very simple example of a bitwise exclusive OR. Like let us say what I am saying, I am taking a problem.

(Refer Slide Time: 03:11)



Let us say I have a variable `a`, I have a variable `b`, which is multibit variable. Let us say this is 0101, this value is 0011, let us say and suppose I am wanting to carry out bit by bit exclusive OR of these bits. And the result let us call it `f`. So, 1 and 1 if you take an exclusive OR, it is 0; 0,1 will be 1; 1 and 0 will be 1; 0 and 0 will be 0. So, this bitwise

EXOR I want to do a parameterize design using genvar using generate, where I can use it for any arbitrary N for any arbitrary N-bit numbers.

So, the idea is that if you want to do one XOR, one bit XOR you need to make one instantiation of an XOR gate. Like for example, a, b or that you can write XOR, you can give the name say a name G0 that you can write $f(0)$, $a(0)$, $b(0)$, like this, then for the second bit you can generate another XOR gate, XOR G1 ($f(1)$, $a(1)$, $b(1)$) and so on. So, what I am saying?

Now, is that we want to define a parameter N, so, that N copies of such XOR's will be automatically generated. So, I do not have to write so many XOR's. This N can be 32, 64 whatever, right. So, this is the example let us see, this is my Verilog code and this is the test bench, this is the verilog code. So, whatever written let us see, this is the name of the module, 3 parameters f, a and b. And here you have defined a parameter N which for the sake of example we have taken it to be 16.

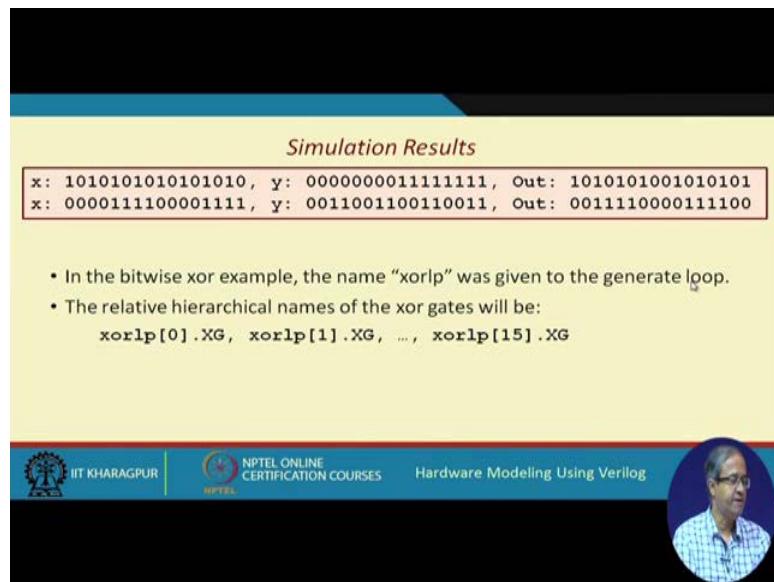
So, a and b these are the inputs, we are declaring it to be inputs from bit number 0 up to $N-1$, a and b, and similarly f is the output, similarly as a vector 0 to $N-1$. And we are declaring a genvar p, which p is a variable which we will be using inside this loop, here you see we use p. And this is how we can use a generate loop, generate a for loop, generate for $p = 0$, $p < N$, $p = p+1$, it is exactly like a for loop, but it starts with a keyword generate and it ends with endgenerate, right.

So, what it will mean is that whatever is there inside this begin-end, this will be instantiated N times, 0 up to $N-1$. So, what I have given inside this loop, you see I have defined a label, xorlp is the label. I have given a label of this loop, and inside this I have instantiated one xor gate, xor. I have given the name as XG ($f(p)$, $a(p)$, $b(p)$), you see here we are given $f(0)$, $a(0)$, $b(0)$, $f(1)$, $a(1)$, $b(1)$. So, generally I am calling it as p, right. So, I call it $b(p)$, $a(p)$, $f(p)$ this is all. So, if I just write this and I give it to the Verilog simulator or the synthesizer. It will be generating N copies of this XOR gates. So, here I have written it only once. So, suppose N is 16, 16 such XOR lines will be generated automatically.

So, we have a means, we can basically test this code out, using a test bench like this, I have also showed a test bench, you can try it out here. We have instantiated this XOR bitwise function, we just use this named convention of passing arguments. This f, a, b

was here, and here in this test bench we are calling them x, y and out. x, y are 16-bits, out is 16-bits. Initial we are monitoring x, y and out, and we are just applying two sample inputs. The first input is that first input x, you are applying a 16-bit hexadecimal number aaaa and y, you are applying 00ff. And after a delay of 10, we are applying x equal to 0f0f and y is 3333 finish.

(Refer Slide Time: 18:36)



So, if you actually run this you get an output like this. You see aaaa and 00ff. This a is 1010, you see 1010, a, 10101010100 and 00ff, bit by bit XOR, 1 and 0 is 1, 0 and 0 is 0. So, the first 8- bits would be like this 1010, and the last one will all be inverted. 1 and 0 XOR is 1, 1 and 1 XOR is 0 and so on.

So, this is one verify. And in the second test case we have given 0f0f and 3333. So, 0 and 3 if we XOR it will be 3, f and 3 if we XOR it will become c1100, then again 0011 again 1100. So, this is the simulation result. Now you see this xorlp and this xor. Now the point is that xorlp, this was the name we had given to the generate loop, right. So, these 16 copies of the XOR gate that will be generated, we have given the name as XG. But there will be 16 such gates, what will be the names. Because in this example I trying to write, I was writing G0, G1, G2 like this. But here I wrote it only once XG. So, what will be the name that will be generated for the 16 copies of these XOR gates? They will be like this xorlp[0].XG, xorlp[1].XG, ...up to xorlp[15].XG.

So, whatever loop variable you have defined that will be treated as a vector and you can use it with an index. Put a dot and whatever variable name you have defined you can use it after that. So, like this you can uniquely refer to the 16 XOR gates, right.

(Refer Slide Time: 20:50)

Example 2: Design of N-bit Ripple Carry Adder

```
// Structural gate-level description of a full adder
module full_adder (a, b, c, sum, cout);
    input a, b, c;
    output sum, cout;
    wire t1, t2, t3;
    xor G1 (t1, a, b), G2 (sum, t1, c);
    and G3 (t2, a, b), G4 (t3, t1, c);
    or G5 (cout, t2, t3);
endmodule
```

How to use "generate" to dynamically create N copies of full adder,
and connect them to make a N-bit ripple-carry adder?

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

Let us take another example, just like the ripple carry adder I talked about. Well, we start by giving you a structural description of a full adder. This is a gate level description of full adder that consists of 2 XOR gates, 2 AND gates and 1 OR gate. I showed this circuit earlier you can just check it out. This is a compact gate level representation of a full adder. So, the first 3 parameters are the inputs then sum and carry out (cout). So, what do we want to do now?

So, we want to use it using generate to create an N-bit ripple carry adder.

(Refer Slide Time: 21:41)

```
module RCA (carry_out, sum, a, b, carry_in);
parameter N = 8;
input [N-1:0] a, b;    input carry_in;
output [N-1:0] sum,   output carry_out;
wire [N:0] carry; // carry[N] is carry out
assign carry[0] = carry_in;
assign carry_out = carry[N];
genvar i;
generate for (i=0; i<N; i++)
begin fa_loop
    wire t1, t2, t3;
    xor G1 (t1, a[i], b[i]), G2 (sum[i], t1, carry[i]);
    and G3 (t2, a[i], b[i]), G4 (t3, t1, carry[i]);
    or G5 (carry[i+1], t2, t3);
end
endgenerate
endmodule
```

Let us see how. This is the code, you see the code is so compact. See this means we are not instantiating this we are just picking this code directly. You see these 3 wire and xor, and, or. So, whatever was this we have directly picked up these 4 lines of code. So, these 3 temporary lines and these 3 xor, and, or. So here you see we are using a parameter N, let us say 8, we are creating an 8-bit adder. So, a and b are 0 to N-1, sum is also 0 to N-1. Carry, there will be one carry means that the output of every ripple carry adders stage. So, it will be N-1 plus 1 carry_out. So, that is why I am defining 0 up to N. And this last bit carry_in that is nothing but the carry_out, I am just doing a continuous assignment using assign. Similarly, the carry_in of the adder that is your carry[0], the rest will be generated by the full adders.

So, I have generated a variable genvar I; generate this loop again 0 to N-1, this is the name of the loop fa_loop and the description, just the description which is given same description here, right. Now you see this generate will be creating 8 copies of these 4 statements are there 4 lines. So, total 32 lines will be generated. Now you see there are so many variables here, there are gates G1, G2, G3, G4, G5. There are some wires t1, t2, t3. So, how do you refer them for the copies? Same way this fa_loop[0].t1, fa_loop[0].t2, fa_loop[0].G1, fa_loop[1].t1 and so on, in the same way.

(Refer Slide Time: 23:48)

- Some of the relative hierarchical instance names that are generated are:
 - fa_loop[0].G1, fa_loop[1].G1, fa_loop[7].G1, etc.
- Some of the nets ("wires") that are generated are:
 - fa_loop[0].t1, fa_loop[1].T2, fa_loop[0].t3, etc.

So, this kind of hierarchical looking conventions you can write. So, fa_loop this index you can use to indicate the copy number, you can indicate the gate number, the intermediate net number anything you want, right. So, this is a very convenient to this, t will be small, right. So, you see using this generate you can very conveniently represent some designs which are inherently iterative in nature.

Like this means, adder is a very natural example. So, means whatever design comes to your mind which can be created by replicating or repeating the same building block a number of times, you can use these kind of general statements very conveniently. So, it makes your module description very small. And you are putting the responsibility on the synthesis or the simulation tool to expand them, right. So, this we come to the end of this lecture.

So, we shall be continuing with the discussion in the next lecture.

Thank you.

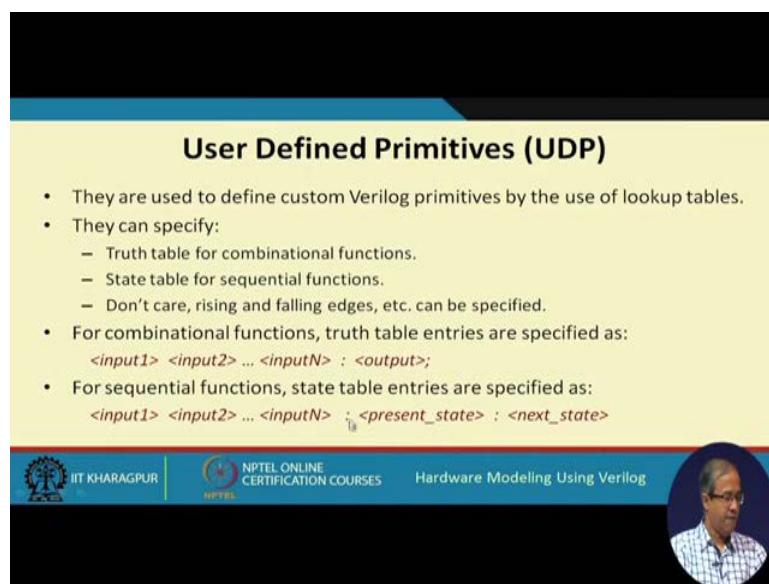
Hardware Modeling using Verilog
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 20
User - Defined Primitives

So, in this lecture, we shall be discussing a new feature of the Verilog language. It is called user defined primitives. This is the title of our lecture; User - Defined Primitives. So, we have seen so far that how you can write modules in Verilog; we can write modules and when you write modules; the description can be in terms of the behavior or it can be in terms of the structure where we instantiate either some built in primitive gates or some other modules inside this module we are writing.

Well, here in this user defined primitive, we are saying; this is also a way of specifying the functionality of a block in a behavioral way, but very specifically, we are specifying the behavior in terms of a truth table for a combinational circuit and a state transition diagram or a state transition table in the case of a sequential circuit. So, many a times, when we are just creating a design where some of the functional blocks; we want to specify in terms of the behavior and for some of them the behavior can be very conveniently expressed in terms of the truth table or the state table, there this user driven or user defined primitives can be quite helpful.

(Refer Slide Time: 01:52)



User Defined Primitives (UDP)

- They are used to define custom Verilog primitives by the use of lookup tables.
- They can specify:
 - Truth table for combinational functions.
 - State table for sequential functions.
 - Don't care, rising and falling edges, etc. can be specified.
- For combinational functions, truth table entries are specified as:
 $<\text{input}_1> <\text{input}_2> \dots <\text{input}_N> : <\text{output}>;$
- For sequential functions, state table entries are specified as:
 $<\text{input}_1> <\text{input}_2> \dots <\text{input}_N> ; <\text{present_state}> : <\text{next_state}>$

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



So, let us see what this is. So, this user defined primitives in short UDP; they can be used to define some as it said behavioral specification of some blocks in terms of look up tables. So, we specify the behavior in terms of tables in a particular format; we shall see. Now using UDP as I had said, we can either specify for combinational functions, its behavior in terms of the truth table or for sequential functions, we can specify the behavior in terms of its state table. Now when we specify these tables, we can also indicate various events like some of the inputs can be don't cares, some of the inputs can be rising or a falling edge kind of signals. So, these kinds of signals can also be specified; we should see some examples here.

So, for combinational functions; when you specify a particular row of this table; this truth table, the format is like this, the value of the inputs, we first give separated by spaces. Suppose there are n inputs and then there is a colon, then we give the expected output; what is the output of that function when this input is applied. So, if it is a truth table, we shall be giving this kind of a line, several times one per row of the truth table. Similarly, when it is a sequential function, then when we specify the inputs as usual and after colon, we specify what is the present state of the circuit or the machine and after another colon, we specify the next state.

Now this present state and next state can also be in terms of the bit values. So, this is the state table description one row of it.

(Refer Slide Time: 04:00)

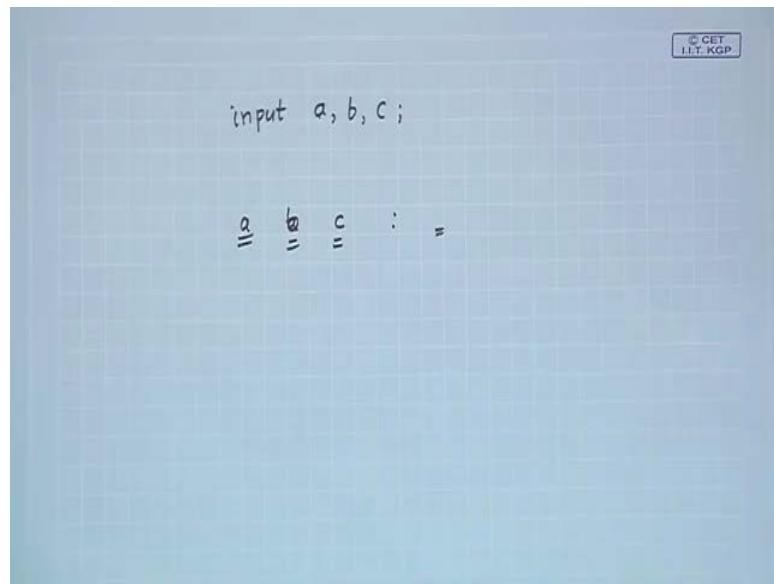
Some Rules for using UDPs

- The input terminals to a UDP can only be scalar variables.
 - Multiple input terminals can be used.
 - The input terminals are declared as "*input*".
 - Input entries in the table must be in the same order as the "*input*" terminal list.
- Only one scalar output terminal must be used.
 - The output terminal must appear in the beginning of the terminal list.
 - For combinational UDPs, the output terminal is declared as "*output*".
 - For sequential UDPs, the output terminal is declared as "*reg*".
- For sequential UDPs, the state can be initialized with an "*initial*" statement.
 - This is optional.

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

So, there are some rules that we need to follow when you are using this kind of user defined primitives. The rule are as follows that the input terminal, the inputs that we are giving, they cannot be vectors, they can only be individual scalar variables, but of course, we can use multiple such variables, multiple input terminals can be used and they have to be declared as input using the keyword input and the point to note is that in the input description the order in which we apply the input for example.

(Refer Slide Time: 04:41)



In the input description we write; let us say input a, b and c. So, when we give the table, we have to first give the value of a, then give the value of b, then give the value of c and then the expected output. So, the order in which you specify the variables in the input declaration, the input variables should appear in the table in exactly the same order. This is something you have to remember.

So, the input entries in the table must be in the same order as they are specified in the input declaration and another point to notice that in one UDP, you can only define a single output function. So, only one scalar output terminal can be used. It cannot be a vector, a single bit and in the terminal list when you declare the terminals. I shall give an example later, the output terminal must be the first one appearing followed by the input terminals and just like in a module the output terminal has to be distinguished using the keyword output, but for sequential circuits, the output terminal should be declared as reg, right.

Particularly for sequential UDPs where we are talking about the present state and the next state, sometimes we may have to initialize the state of the machine; suppose unless we specify the initial state; whenever I apply an input, we really do not know what the next state will be because they do not know the present state. So, for sequential UDP and additional facilities there, you can include an initial statement one using which you can initialize the present state of the machine.

(Refer Slide Time: 06:40)

Some Guidelines

- User defined primitives (UDPs) model functionality only.
 - They do not model timing or process technology.
- A functional block can be modeled as a UDP only if it has exactly one output.
 - If a block has more than one outputs, it has to be modeled as a module.
 - As an alternative, multiple UDPs can be used, one per output.
- Inside the simulator, a UDP is typically implemented as a lookup table in memory.
- The UDP state tables should be specified as completely as possible.
 - For unspecified cases, the output is set to "x".

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

So, this state can be initialized with an initial state; however, this is optional; you may or may not give this. Some guidelines are as follows well means here as it said, when you declare or define UDPs, you define them in the form of a table or a truth table or a state table. So, it models only the functionality just the input output behavior. So, you do not specify any kind of delay or process technology, see we are not talking about anything related to the hardware that you are going to design, we are simply defining our functionality in terms of black box. This will be my input, this will be my output; that is it, we are not talking about any kind of gates, any kind of delays, any kind of process technology, what kind of delays are the things will be encountered and so on.

So, as I said earlier also that in an UDP, you can have exactly one output terminal. So, if you need to define a block which has more than one outputs, then it has to be declared as a module and not as a UDP, but; however, you can use multiple UDP's, one per output that is also permitted. Now when you simulate a design that contains UDP typically

inside this simulator, this UDP is implemented as a table. So, if it is a truth table, it maintains that table in memory.

So, whenever some input is applied, there is a table lookup and the output can be determined. Similarly, the state table of a sequential circuit that is also maintained like a table in memory; so, it maintains the input output behavior in terms of a lookup table in simulator memory. And when you are declaring sequential circuit behavior sequential functions, the state table should be specified as completely as possible, well, if you have some combinations where the inputs are not specified for those cases, the outputs will be automatically set to undefined or x values.

(Refer Slide Time: 08:59)

The screenshot shows a code editor with two side-by-side examples of Verilog User-Defined Primitives (UDPs) for a full adder.

Left Example:

```
// Full adder carry generation
primitive udp_cy (cout, a, b, c);
    input a, b, c;
    output cout;
    table
        // a   b   c   cout
        0   0   0   :  0;
        0   0   1   :  0;
        0   1   0   :  0;
        0   1   1   :  1;
        1   0   0   :  0;
        1   0   1   :  1;
        1   1   0   :  1;
        1   1   1   :  1;
    endtable
endprimitive
```

Right Example:

```
// Full adder carry generation
// Using don't care ("?")
primitive udp_cy (cout, a, b, c);
    input a, b, c;
    output cout;
    table
        // a   b   c   cout
        0   0   ?   :  0;
        0   ?   0   :  0;
        ?   0   0   :  0;
        1   1   ?   :  1;
        1   ?   1   :  1;
        1   1   ?   :  1;
    endtable
endprimitive
```

The code editor interface includes the IIT Kharagpur logo, NPTEL Online Certification Courses logo, and the title "Hardware Modeling Using Verilog".

Let us now look at some examples; first modeling of combinational circuits; let us start with something we already know about the sum output of a full adder. You see this is the declaration of a user defined primitive of this sum output. So, it looks exactly like a module in terms of declaration, but instead of the module keyword, we use a “primitive” keyword here and a “endprimitive” keyword here. So, the name of the block in this case, they will user defined primitive. So, UDP sum is the name we have given and these are the arguments. So, as I said the output must be the first one appearing in this list, sum is the output and a, b, c are the inputs.

So, inputs are declared by the keyword input, the output is declared by the keyword output, then the truth table is specified using the keywords “table” and “endtable” just for

convenience, I have included a comment line here just to tell you that this is a, this is b, this is c and this is sum. So, the truth table there will be 3 input, there will be 8 rows. So, all the 8 combinations are specified if the input is 000, sum will be 0; if it is 001, sum will be 1 and so on, if it is 111, sum will be 1. This is how you can specify the truth table of a sum output.

So, in this example, we have specified the output value for all possible input combinations, but there can be functions where some of the inputs you can specify as don't cares also, now here the don't care values are indicated by a question mark. So, we shall be illustrating with another example; just the carry output of the full adder. This is just the other example, well, this is instead of sum this is the carry.

So, I have shown 2 alternate descriptions, both are equivalent, first one is the one where we have expressly specified all the 8 combinations in the truth table. So, it is exactly like this. Some example we took earlier. So, here the output variable is cout; carry out and in the comment, I have just mentioned just for convenience these are a, b, c; this is cout. So, for each input value; what is the expected value of carry out? We have just listed them in this table.

Now the description on the right is exactly the same thing, but here we have used don't cares. Well, how we have used don't cares, you see we are talking about generation of carry. So, if any 2 of the inputs are 0, there cannot be any carry. So, if a and b are 0, but c is an don't care, there cannot be a carry. Similarly, if a and c are 0 and b is a don't care, then also no carry and if b and c are 0, then also no carry. Well, the condition for carry generation is at least 2 of the inputs must be 1. So, either a, b is 1, c is don't care or a and c is 1, b is don't care or a and b is or this should be actually I think, there is a small typo; this should be a; a should be don't care and this should be 1, fine.

(Refer Slide Time: 12:43)

```
// Instantiating UDP's
// A full adder description
module full_adder (sum, cout, a, b, c);
    input a, b, c;
    output sum, cout;

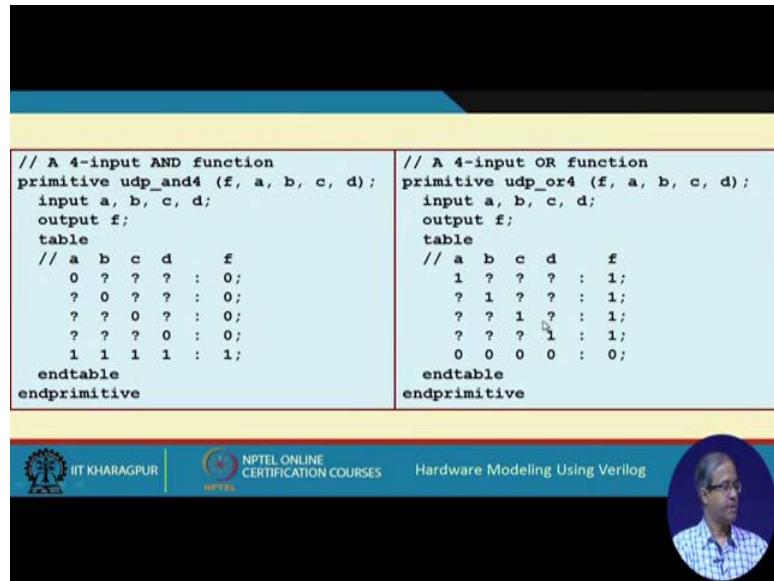
    udp_sum SUM (sum, a, b, c);
    udp_cy CARRY (cout, a, b, c);
endmodule
```

The slide also features the IIT Kharagpur logo, NPTEL Online Certification Courses logo, and a circular portrait of a man.

So, this is how you can use don't care in the truth table description. You see instead of 8 here, we need only 6 rows to specify it. There are some examples of combinational functions. Now once you declared these kind of sum here and carry here, these you can instantiate in a module just like module description which you saw earlier. Suppose we have declared these two, this UDP is `udp_sum`, other is called `udp_cy`, we instantiate them just like their modules.

So, for instantiation it really does not make any difference whether it is another module or it is a UDP, the way for instantiation or the rule for instantiation is the same just you specify the name of that module; give it a instantiated name and the parameter list. So, the way to include or instantiate a UDP is the same as that for a module. So, for a complete full adder description, you can have a description like this; you instantiate the `udp_sum`, instantiate the `udp_cy` and have the full adder description.

(Refer Slide Time: 13:59)



```
// A 4-input AND function
primitive udp_and4 (f, a, b, c, d);
  input a, b, c, d;
  output f;
  table
    // a b c d   f
    0 ? ? ? : 0;
    ? 0 ? ? : 0;
    ? ? 0 ? : 0;
    ? ? ? 0 : 0;
    1 1 1 1 : 1;
  endtable
endprimitive

// A 4-input OR function
primitive udp_or4 (f, a, b, c, d);
  input a, b, c, d;
  output f;
  table
    // a b c d   f
    1 ? ? ? : 1;
    ? 1 ? ? : 1;
    ? ? 1 ? : 1;
    ? ? ? 1 : 1;
    0 0 0 0 : 0;
  endtable
endprimitive
```

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



So, as I said, they can be instantiated just like any other Verilog module. Let us take some other very simple functions where you will see that we can use a lot of don't cares. Just consider we want to implement or we want to specify a 4 input AND and a 4 input OR function. So, for AND function, again the output will be the first one appearing then the 4 inputs a, b, c, d; they are declared. So, again just in a comment, I have shown the inputs and the outputs.

Now for AND function, the output will be 1, if all the inputs are 1. So, 1, 1, 1, 1 that is the only combination when the output will be 1 and the output will be 0, if at least one of the inputs is 0. So, if a is 0; b, c, d are don't cares; then 0 or if b is 0 others are don't cares; c is 0 or d is 0, f will be 0. So, you see instead of 16 rows in the truth table, here we are able to specify it only in 5 rows. Similarly, for OR, it is just the reverse, for an OR gate; the output will be 0, if all the inputs are 0s. This is the last row is specified and if any one of the input is 1; the others can be don't cares; the output will be 1. So, as you can see here also, we need 5 rows to specify.

(Refer Slide Time: 15:26)

```
// A 4-to-1 multiplexer
primitive udp_mux41 (f, s0, s1, i0, i1, i2, i3);
    input s0, s1, i0, i1, i2, i3;
    output f;
    table
        // s0 s1  i0 i1 i2 i3  :  f
        0 0    0 ? ? ? : 0;
        0 0    1 ? ? ? : 1;
        1 0    ? 0 ? ? : 0;
        1 0    ? 1 ? ? : 1;
        0 1    ? ? 0 ? : 0;
        0 1    ? ? 1 ? : 1;
        1 1    ? ? ? 0 : 0;
        1 1    ? ? ? 1 : 1;
    endtable
endprimitive
```

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

Let us look at a slightly more complex example of 4 to 1 multiplexer; you see functional blocks like multiplexers or decoders; they are pretty convenient to specify in terms of the input output behavior or truth table. So, multiplexer is one classical example where this UDP description can be pretty convenient. Let us see how we can do it. So, we are declaring a 4 to 1 multiplexer. So, we had said, we cannot define vectors the inputs have to be all scalars.

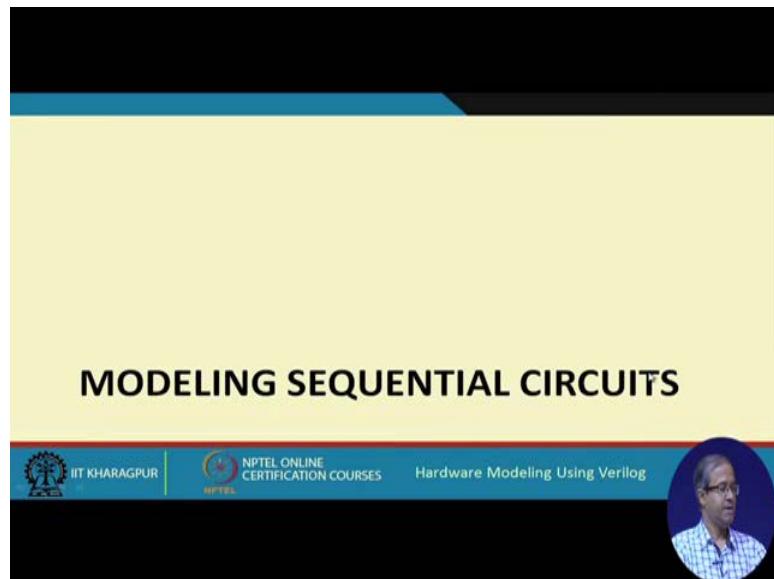
First one is the output, then next two are the select inputs, next four are the primary inputs to the OR, the inputs to the multiplexer, the 4 inputs. So, this 6 are the inputs s0, s1 and i0 to i3 and f is the output. Now in the description, we have again given a comment; the first 2 are the select lines; this is the least significant, next one is the most significant and then the 4 inputs i0, i1, i2, i3, then colon f. See this line, we give normally in order to understand that which bit is what, just to have a feel. Now the idea is that the inputs there will be exactly coming in the same order you have specified here s0, s1, i0, i1, i2, this should be same order the last one will be the output.

So, select line; there are two, there can be four combinations, I have shown them by colors; there can be 00, they can be 01, they can be 10 or they can be 11. So, if it is 00, then i0 is supposed to be selected and it should be going to f. So, we have written two rows, if i0 is 0, f will be 0; if i0 is 1, f will be 1; the other set don't cares. Similarly, when the inputs are 1 and 0, s1 is 1 and s0 is 0. So, s0 is 1, s1 is 0 then i1 is supposed to be

selected. So, if i_1 is 0 output is 0, i_1 is 1, output is 1, other said don't cares. Similarly, for 10 combination, i_2 is selected and for 11 combination, i_3 is selected.

So, you see; there are 6 inputs. So, the complete truth table is supposed to contain 64 rows, 2^6 , but in this compact district description in terms of the functionality of the multiplexer, we need only 8 rows to specify. So, and these 8 rows are sufficient to specify the functions of the multiplexer, exactly, what you are trying to do? We want to use this s_0 and s_1 to select one of the inputs. So, whether they are getting selected or not that is what we have specified here, just 8 rows.

(Refer Slide Time: 18:32)



Next let us see; how we can model sequential circuits. Now in a sequential circuit, you remember, you recall. So, it is unlike a combinational circuit where we apply input you get an output, well in a sequential circuit there is something called a state, it remembers some previous history. So, whenever we apply an input, the output will depend not only on the input you are applying but also in terms of this state where the machine is in. So, in the UDP of a sequential circuit; you can specify the input, you can specify the present state, you can specify the next state. Now the restriction of a UDP is your next state that is the output has to be a single bit, single scalar variable, it cannot be multiple.

(Refer Slide Time: 19:34)

The screenshot shows a Verilog code editor with a yellow background. The code defines a primitive Dlatch. It includes an initial block setting q = 0, a table defining the next state based on inputs d, clk, and clr, and an endprimitive block. The table rows are as follows:

d	clk	clr	q	q_new	Notes		
?	?	1	:	?	:	0;	// latch is cleared
0	1	0	:	?	:	0;	// latch is reset
1	1	1	:	?	:	1;	// latch is set
?	0	0	:	?	:	-;	// retains previous state

```
// A level-sensitive D type latch
primitive Dlatch (q, d, clk, clr);
    input d, clk, clr;
    output reg q;

initial
    q = 0; // This is optional

table
// d  clk  clr      q   q_new
//      ?   ?   1       :   ?   :   0;   // latch is cleared
//      0   1   0       :   ?   :   0;   // latch is reset
//      1   1   1       :   ?   :   1;   // latch is set
//      ?   0   0       :   ?   :   -;   // retains previous state
endtable
endprimitive
```

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

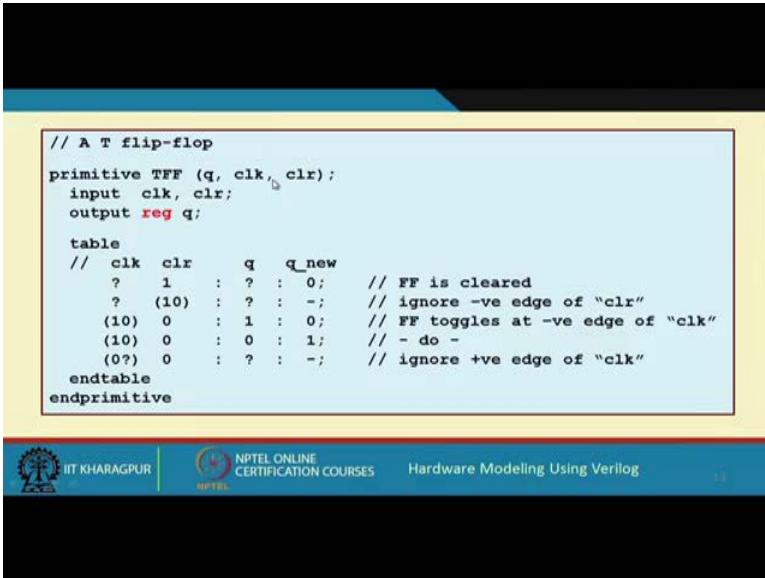
So, if there are multiple variables you want to generate or declare; you have to declare it in terms of a module not a UDP. Let us see, this is a very simple description of a D-type latch; let us see just like module we have declared primitive, endprimitive name is Dlatch the inputs are d, clk and this clk is like an enable because if the level sensor is not real clock, it is like enable whenever clock is high, d will go to q and clr to clear the output.

So, d, clk and clr are the inputs and this d for a sequential circuit you have to declare the output of the reg, I told you. So, I am declaring output reg q. So, in terms of the sequence circuit description, I told you, you have to first specify the inputs in this case with specifying in this order d, clk clr, then the present state, then the next state, we are just referring to as q and q_new. Just look at the combinations, first row says that I have applied clr = 1, d and clk are don't care. So, if clr = 1, irrespective of d and clk, irrespective of the present state, my next state will be 0, it is cleared, right and if I apply d = 0 and make clk equal to high and not clear, if clr is 1, it look cleared; if clr is 0, then the output will be 0, here also I made a typo, they should be 0, fine.

So, if I apply d = 0 and clock high, the next state will be 0 and if I apply d = 1 and clock high, this 1 will go, next state will be 1 and this will be irrespective of what my present state is and of course, my clear should be 0, not clear and last row specifies that if my clock is not active my enable is not active my clear is also not active, then you see in the next state, I put a dash [-]; dash means no change, retains previous state. So, for a d-type

latch the next state is not directly dependent on the previous state, but on what you are applying, the only combination that depends on the previous is the last one, dash, it will remember whatever it was previously, right.

(Refer Slide Time: 22:14)



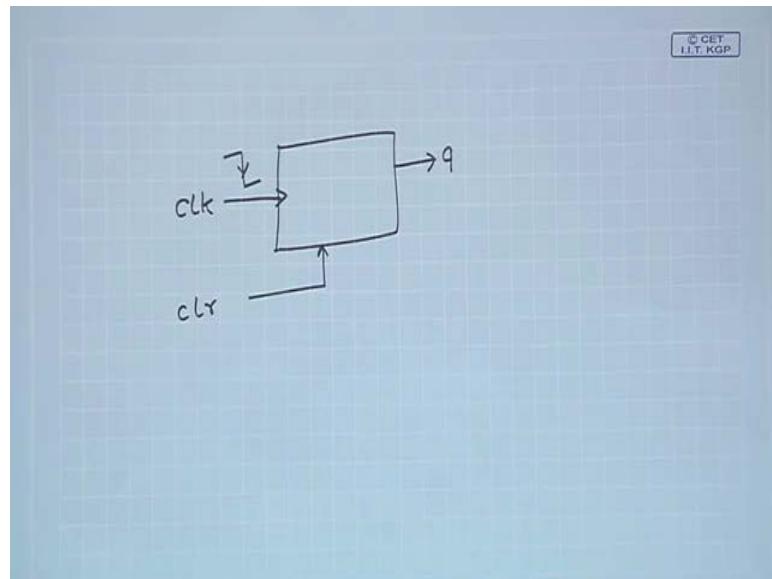
```
// A T flip-flop
primitive TFF (q, clk, clr);
    input clk, clr;
    output reg q;

    table
        // clk  clr      q   q_new
        ?    1       : ?  : 0;    // FF is cleared
        ?    (10)   : ?  : -;   // ignore -ve edge of "clr"
        (10) 0     : 1  : 0;    // FF toggles at -ve edge of "clk"
        (10) 0     : 0  : 1;    // - do -
        (0?) 0    : ?  : -;   // ignore +ve edge of "clk"
    endtable
endprimitive
```

The screenshot shows a Verilog code editor with a yellow background. The code defines a primitive TFF with inputs clk and clr, and an output q. It includes a table that specifies the behavior of the flip-flop based on the values of clk and clr. The table entries are: (1, 0) leads to q=0 (FF is cleared); (10, ?) leads to q=- (ignore -ve edge of "clr"); (0, 1) leads to q=0 (FF toggles at -ve edge of "clk"); (0, 0) leads to q=1 (- do -); (0?, 0) leads to q=- (ignore +ve edge of "clk"). The code ends with endtable and endprimitive statements.

Now, let us look at a flip-flop where there is a clock signal, proper clock signal. So, here we are defining a toggle flip-flop or a T flip-flop where I mean our descriptions is like the q, clk and clr. So, our description of this flip-flop is that there is a clk input, there is a q output and there is a clr input.

(Refer Slide Time: 22:31)



So, whenever there is a clk signal, I am assuming it is falling edge triggered 1 to 0; the output will be toggled, if it was 1, it will become 0; if it was 0, it will become 1, so, whenever there is a clock. This is a simple description of a sequential circuit, I am assuming there is a clk input, a clr input and q, whenever there is a clock, the output will toggle, if it is a clear, then output will be 0, let us see.

So, again clk and clr are the inputs and q is the output, declare as a reg. So, we are specifying in this order. First row specifies if clear is high, then irrespective of whether clock is coming or not, whatever was the previous state, the output is becoming 0, next state is 0, q, q_new. Next was specifies my clear is becoming 1 to 0, I have withdrawn the clear, it was clear; I withdrawn it. So, the output value will remain in the previous state. So, if there is a negative agent clear we are ignoring it, we are retaining the previous state. Now there is a clock, you say within bracket, if we write 10, it means a falling edge, see in clock within bracket we are writing 10 and this means 1 to 0 transition, there is a transition from 1 to 0, that means, a falling edge.

So, in the next 2 rows, we are saying that there is a falling edge on the clock, not clear and my previous state was 1 and 0. So, now, if my previous state was 1, it will be toggled, my current state will be 0; if my previous state was 0, my current state will be 1 and if there is a positive edge of the clock, it was 0 to question mark, was anything, then there will be no change. So, that is indicated by a dash, this is a very concise description of a T flip-flop in terms of the state table. Let us take some more examples.

(Refer Slide Time: 24:52)

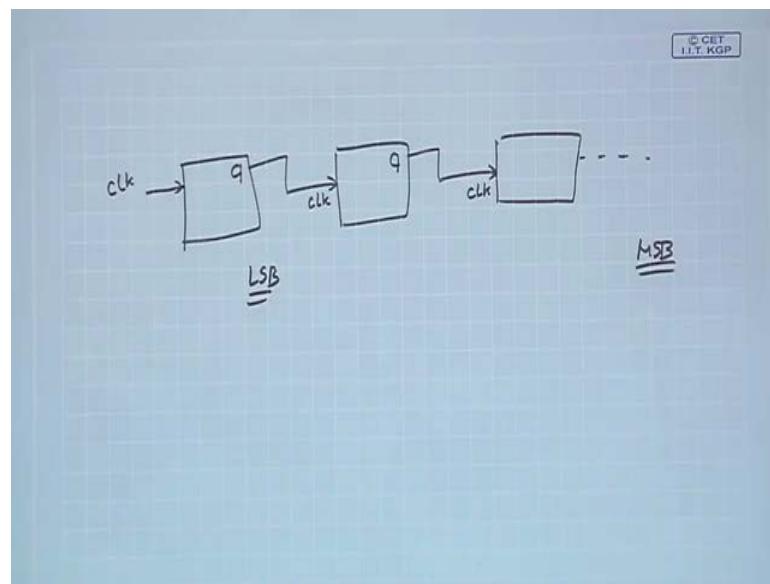
The screenshot shows a presentation slide with a yellow header and footer. The header contains the text "NPTEL ONLINE CERTIFICATION COURSES" and "Hardware Modeling Using Verilog". The footer features a portrait of a man. The main content is a Verilog code snippet:

```
// Constructing a 6-bit ripple counter using T flip-flops
primitive ripple_counter (count, clk, clr);
    input clk, clr;
    output [5:0] count;

    TFF F0 (count[0], clk, clr);
    TFF F1 (count[1], count[0], clr);
    TFF F2 (count[2], count[1], clr);
    TFF F3 (count[3], count[2], clr);
    TFF F4 (count[4], count[3], clr);
    TFF F5 (count[5], count[4], clr);
endprimitive
```

This T flip-flop that I have just now defined, well we can connect a number of such T flip-flops to create a ripple counter. So, how do create a ripple counter, a number of T flip-flops connected in cascade.

(Refer Slide Time: 25:11)

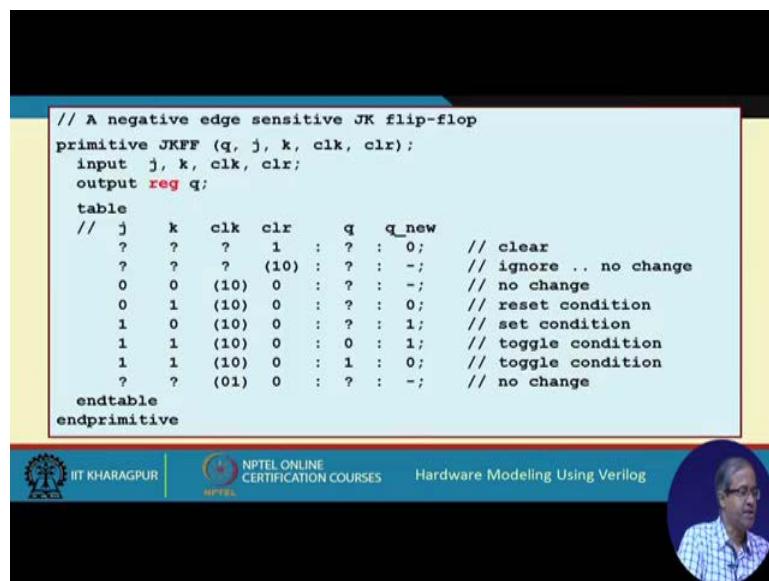


So, how do we do it, suppose I have several such flip-flops? So, the output of one flip-flop, I am connecting into the clock input of the next flip-flop, the output of the next I am connecting to the clock input of the next flip and so on.

And this is my final clock. My initial clock is applying. So, it will be counting this and the q outputs which will be coming, this will be your LSB and on the right side, this will be a most significant bit, this will be your counter. So, here I am just showing you how this flip-flop can be instantiated to create a 6-bit ripple counter. So, what we have done? We have taken this same flip-flop, you see first one is q, second one is clk, third one is clr. So, we have taken the same one, we have instantiated it 6 times, 6 copies of the flip-flop, for the flip-flop on the left hand side the first one.

So, the output you are calling us count[0], clk, we are directly applying like you see the clk you are applying from outside, we are just applying clk from outside, that is why the input clk is connected here. clk and clr is connected to all of them, second flip-flop the output I am calling it as count[1], instead of clk I am connecting the output of the first flip-flop here, you see what we did here again the output of the first flip-flop is connected to the clk of the next, similarly here. So, we have done the same thing, this output of this flip-flop is connected as a clk here, output of this is connected as clk here, like this. So, in this way, we have created a 6-bit ripple counter.

(Refer Slide Time: 27:07)



```

// A negative edge sensitive JK flip-flop
primitive JKFF (q, j, k, clk, clr);
    input j, k, clk, clr;
    output reg q;
    table
        // j   k   clk   clr   q   q_new
        ?   ?   ?   1   : ?   : 0;   // clear
        ?   ?   ?   (10) : ?   : -; // ignore .. no change
        0   0   (10) 0   : ?   : -; // no change
        0   1   (10) 0   : ?   : 0; // reset condition
        1   0   (10) 0   : ?   : 1; // set condition
        1   1   (10) 0   : 0   : 1; // toggle condition
        1   1   (10) 0   : 1   : 0; // toggle condition
        ?   ?   (01) 0   : ?   : -; // no change
    endtable
endprimitive

```

The screenshot shows a Verilog code editor with a highlighted block of Verilog code. The code defines a primitive JKFF with inputs j, k, clk, and clr, and an output q. It includes a table section with 16 rows of truth values for the four inputs. The table entries are as follows:

j	k	clk	clr	q	q_new
?	?	?	1	:	0;
?	?	?	(10)	:	-;
0	0	(10)	0	:	-;
0	1	(10)	0	:	0;
1	0	(10)	0	:	1;
1	1	(10)	0	:	0;
1	1	(10)	0	:	1;
?	?	(01)	0	:	-;

The code ends with an endprimitive statement. At the bottom of the editor window, there are logos for IIT Kharagpur, NPTEL, and the course title "Hardware Modeling Using Verilog". To the right of the editor is a circular portrait of a man.

Now, just one thing, here we have called it primitive, but if you want you can also call it a module because it is as good as a module description. So, I have given primitive and endprimitive, but you can as well express it as a module, this can be a module where the 6 flip-flops have been instantiated, right.

Now, let us look at slightly more complex kind of flip-flops, here is a negative edge sensitive JKFF. So, in a negative edge sensitive JKFF, there is an output q; j, k are the inputs, there is a clk and a clr; j, k, clk, clr are the inputs, output q is the output. So, we have specified them in this order and these are the output. So, the first row says again if clear is 1, irrespective of anything else; output will be 0 and if the clear changes from 1 to 0, there will be no change. Next row says if j, k is 0, 0; you just recall the definition of a JKFF, if JK is 00, the output is supposed to remain same, no change and there is a clk, clk goes from 1 to 0, it is not clear. So, output will not change, but if its 01, then the output will be 0, if there is a clock, the irrespective of the previous output; 10 the output will be 1; but if it is 11, the output will be toggled.

So, here we specify the previous, if it is 11, if the previously output was 0, now it will be 1; if previously it was 1, now it will be 0 and if the clk is rising 0 to 1, not the active edge then again no change.

(Refer Slide Time: 29:10)

```

// A positive edge sensitive SR flip-flop
primitive SRFF (q, s, r, clk, clr);
    input s, r, clk, clr;
    output reg q;

    table
        // s   r   clk  clr      q   q_new
        ?   ?   ?   ? : ?   : ?   : 0;   // clear
        ?   ?   ?   (10) : ?   : ?   : -;  // ignore .. no change
        0   0   (01) 0 : ?   : ?   : -;  // no change
        0   1   (01) 0 : ?   : ?   : 0;   // reset condition
        1   0   (01) 0 : ?   : ?   : 1;   // set condition
        1   1   (01) 0 : ?   : ?   : x;   // invalid condition
        ?   ?   (10) 0 : ?   : ?   : -;  // ignore .. no change
    endtable
endprimitive

```

↳

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog | 16

So, this is a very concise description of a JKFF. Similarly, you can have an SR flip-flop, let us give the name properly SRFF, fine. So, this is positive edge sensitive SRFF same way. You see the first few rows are identical to JK, first two if 00, no change; 01, it will be 0; 10, it will be 1; but 11 is an undefined condition for a SR flip-flop. So, when the inputs are 1 and 1 and if there is a clock, suppose I am saying; it is a positive edge

sensitive 0 to 1, then the output I am marking as x (undefined) and if the clock is on the other edge, no change, right.

So, in this way, we can actually specify any kind of simple combinational or sequential circuit blocks provided there is a single output. So, we have taken examples of flip-flops where there is one output, we have taken examples of gates and multiplexers where there is one output, only such functional descriptions can be specified using such UDPs.

(Refer Slide Time: 30:26)

The slide has a black header bar and a yellow main content area. At the top of the yellow area, the title 'Some Rules to Follow' is centered. Below the title is a bulleted list of six rules. At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the text 'NPTEL ONLINE CERTIFICATION COURSES', and the course name 'Hardware Modeling Using Verilog'. To the right of the footer bar is a circular profile picture of a man.

Some Rules to Follow

- The "?" symbol cannot be specified in an output field.
- The "-" symbol, indicating no change in the state value, can be specified only in an output field.
- The shortcut "r", indicating rising edge, can be used instead of (01).
- The shortcut "f", indicating falling edge, can be used instead of (10).
- The shortcut "*" indicates any value change in the signal.

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

So, there are a few rules that you need to follow that the question mark symbol is the don't care symbol. This you can use only for the inputs, you cannot use in the output field and dash indicates no change. This again you cannot use in the input, you can use only in the output field and (01) as it said indicates it changes from 0 to 1. So, instead of (01), you can write a shortcut "r" also, "r" is equivalent to rising. Similarly, (10) you can write "f" falling and "*" (star) in place of an input signal means any value change, any change in that value is indicated by star. So, you can also include this in your table. So, with this way come to the end of this lecture.

Now, in this lecture, as I said, we had looked at a new feature of the language Verilog namely the user defined primitives. So, if you want you can include such UDP descriptions for some blocks in a design, bigger design you can instantiate the UDPs in your main design modules.

Thank you.

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

Lecture - 21
Verilog Test Bench

So, in our previous lectures we had looked at a number of examples in Verilog, and we have also seen how to write test benches for such modules that we have described. We had looked at some of the features of the test benches; how you can apply this stimulus values for combination circuits, how you can apply clocks for sequential circuits and so on.

Now, in this lecture, we shall be looking at the test benches, the different commands that are available in a more formal and elaborate way and in the next lecture, we shall be looking at some examples on some of the features that we will be discussing in the process. So, our lecture is titled Verilog Test Bench.

(Refer Slide Time: 01:09)

Verilog Test Bench

- What is test bench?
 - A Verilog procedural block that executes only once.
 - Used for simulation.
 - Test bench generates clock, reset, and the required test vectors for a given *design-under-test* (DUT).
 - The test bench can monitor the DUT outputs and present them in a way as specified by the creator.
 - Print the values of the signal lines.
 - Dump the values in a file from where waveforms can be viewed.

Now, these are the things which I have already discussed earlier. So, let us have a quick recap. So, a test bench is a Verilog module, it is a procedural block that runs or executes only once. You recall a procedural block can be of two types; the always procedural block and the initial procedural block. Particularly, the initial procedural block is used for writing test benches because there are certain things which you need to run only once

and the initial block is meant to be run only once; of course, there are some parts of the test bench like generating a clock which you want to do it repeatedly that part you can do using always if you want, fine.

So, such this initial blocks and these test benches written using those; they are used only for simulation. So, if your task is to synthesize a circuit, then you do not write the test bench, you will be writing test bench, if you want to verify your design by applying some test inputs and see what the outputs are coming then only you write the test bench, fine. So, roughly speaking what does the test bench do, they generate the different inputs to your circuit or your design like clock, reset and the other functional inputs.

So, suppose you have created a design, let us call it a design under test or DUT; you can apply such inputs to DUT and it can also monitor the outputs. You see when you apply the inputs; after that you may want that I want to see what the outputs are. So, your test bench can just print those values on the screen or if you want it can also dump it in a file. So, that you can later on see the signal values in a nicely graphical interface, the waveforms, the timing diagrams and so on in that form. So, you can view the outputs in whatever way you want to by specifying it appropriately, right.

(Refer Slide Time: 03:44)

The slide has a yellow background with a black header bar at the top. The text is organized into two main sections:

- Basic requirements:**
 - The inputs of the DUT need to be connected to the test bench.
 - The outputs of the DUT needs also to be connected to the test bench.
- Points to note:**
 - Test benches use the “*initial*” procedural block that executes only once.
 - Can also use “*always*” for generating some test inputs, like a clock signal.

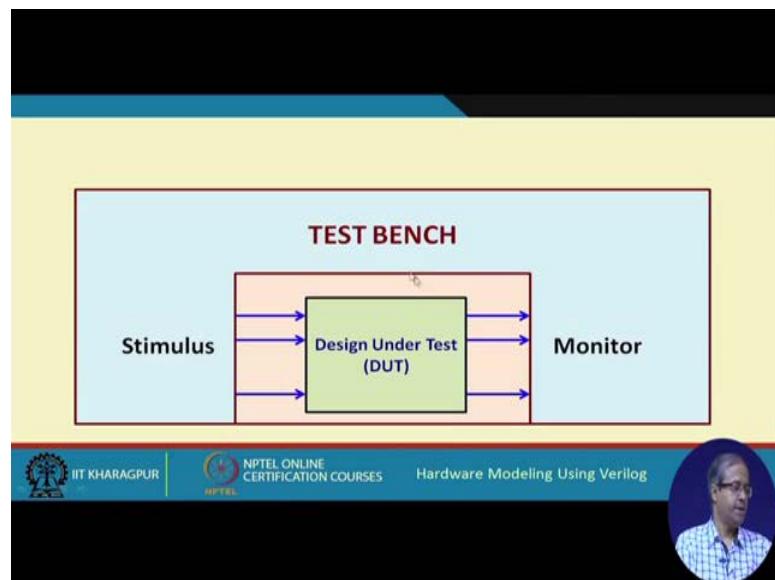
At the bottom of the slide, there is a footer bar with the following elements:

- IIT KHARAGPUR logo
- NPTEL ONLINE CERTIFICATION COURSES logo
- Hardware Modeling Using Verilog
- A circular portrait of a man, likely the speaker.

As I said, you can either print the values or you can dump them in a file from where you can view them in the form of waveforms, fine.

So, the basic requirement is the design that you have created, the inputs of it, should be connected to the test bench because the test bench will be applying the inputs to the DUT. Similarly, the outputs of the DUT should be connected to the test bench because the test bench would be reading those values and will be printing or displaying whatever you want to. So, these are the things I have already mentioned. So, in a test bench you can either use initial blocks or always blocks also. So, initial procedural block is the one which executes only once, but always block is repetitive. So, for generating some periodic signals like clock you can use always, ok. Pictorially whatever I told just now you can visualize like this.

(Refer Slide Time: 04:38)



Let us say this is your design which you have written in Verilog, I am showing it as a box and this is the test bench you have written. So, the first thing what you do is that you instantiate a copy of your design within the test bench, then you connect the input lines of your design under test to some stimulus output lines from your test bench. So, your test bench outputs will be your inputs to your DUT, similarly whatever DUT generates as outputs that will be read by your test bench and it will be monitored, you can either print it, you can display it whatever.

(Refer Slide Time: 05:38)

A Simple Example

```
module example
(A,B,C,D,E,F,Y);
input A,B,C,D,E,F;
output Y;
wire t1, t2, t3, Y;
nand #1 G1 (t1,A,B);
and #2 G2 (t2,C,~B,D);
nor #1 G3 (t3,E,F);
nand #1 G4 (Y,t1,t2,t3);
endmodule
```

```
module testbench;
reg A,B,C,D,E,F; wire Y;
example DUT(A,B,C,D,E,F,Y);

initial
begin
$monitor ($time," A=%b, B=%b, C=%b,
D=%b, E=%b, F=%b",
A,B,C,D,E,F,Y);
#5 A=1; B=0; C=0; D=1; E=0; F=0;
#5 A=0; B=0; C=1; D=1; E=0; F=0;
#5 A=1; C=0;
#5 F=1;
#5 $finish;
end
endmodule
D
```

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

So, stimulus and monitoring these two mechanisms are also part of your test bench. This gives you the overall picture, fine. A very simple example to start with. Let us say here we have a simple module description where there are 4 gates in our design, there are seven parameters A, B, C, D, E, F, Y. Where these 6 parameters A, B, C, D, E, F are the inputs and Y is the output and t1, t2, t3 and this Y are some wires; t1, t2, t3 are the intermediate lines and Y is the final output.

So, this is just an arbitrary description, this is not any design which is meaningful, it does not compute any meaningful function, just for illustration. So, you instantiate the 4 gates, you appropriately specify the input values and some of them can be inverted also like not B, you can write and this t1, t2, t3 are the outputs of G1, G2, G3 which will be the inputs of the last gate here. So, finally, the output is Y. So, this kind of test bench you have already seen earlier. So, in the test bench what you do, first thing is that you instantiate this module, this A, B, C, D, E, F, Y, here let us say we will give the same names A, B, C, D, E, F, Y.

So, whatever were the inputs of here A, B, C, D, E, F; here they will be outputs actually and you have to declare them as reg because you will be assigning them inside a procedural block. You will be assigning some values to A, B, C, D, E, F that is why they will be reg and for a module, Y was the output, but for your test bench, this Y you declare just as a wire because this will be an input, you can just read it and you can just

print it. So, inside the initial block, you can give several statements; we shall explain this meaning, first is a monitor command, monitor actually means that whenever any of these variables whatever you specify in this list they change, you print them in this format, first to print the time, simulation time, then A=%b, b means binary A equal to this value, B equal to, C equal to and so on.

And with a delay of 5, 5, 5 where applied the test vectors, C; first we have applied A=1, 0, 0, 1, 0, 0 then you have set 0, 0, 1, 1, 0, 0, then you have set only A=1, C=0, it means the other variables are not changing, B is still 0, D is still 1, E is still 0, F is still 0 and in the last line, we have written F=1; that means, the remaining 5 variables are same only F is changing, F is 1 and then we finish our simulation. So, this test bench if you just run, you will get the outputs for this Y in this order.

(Refer Slide Time: 08:55)

How to write test benches?

- Create a dummy template
 - Declare inputs to the design-under-test (DUT) as “*reg*”, and the outputs as “*wire*”.
 - Because we have to initialize the DUT inputs inside procedural block(s), typically “*initial*”, where only “*reg*” type variables can be assigned.
 - Instantiate the DUT.
 - Initialization and Monitoring
 - Assign some known values to the DUT inputs.
 - Monitor the DUT outputs for functional verification.

Now, there are a few things points. So, when you write this test bench, here is an example, you create a dummy template, module test bench n module, then you instantiate your DUT or design under test inside. So, first task for writing test bench is to create a dummy template. So, insert the dummy template of course you will have to instantiate your design under test and whatever signals were inputs to your duty, they will now be declared as reg and whatever signals were outputs in your duty, they will be declared as wire. So, you see exactly that we did here. So, whatever our inputs here A, B, C, D, E, F; they have declared as reg and Y was output, Y you have declared as wire,

right, the reason I have already mentioned. So, these you have to declare reg because you will be assigning them to values inside the initial procedural block.

So, here let us say inside this initial block, you are assigning values to A, B, C, D, E, F and we said earlier that inside any procedural block whenever you assign values the left hand side must be a reg type variable, that is the reason. So, means after that you will have to initialize and monitor, will have to apply some values to the inputs of your DUT.

(Refer Slide Time: 10:41)

- For synchronous sequential circuits:
 - We need some clock generation logic.
 - Various ways to specify clock signal.
- Test bench can include various simulator directives:
 - \$display, \$monitor, \$dumpfile, \$dumpvars, \$finish, etc.
- Important point:
 - We do not need test bench when we are synthesizing a design.
 - Required only during simulation.

And you will have to monitor the DUT outputs in a suitable way to verify whether it is working correctly or not. So, for synchronous sequential circuits, in addition to your inputs you may be having some clocks, reset, clear, this kind of inputs. So, you need something more for sequential circuits. So, whatever example I took just now that is for a combinational circuit, but for a synchronous sequential circuit you will also need some clock generation logic. There are various ways to specify clock signals, we shall see later and through some examples how to do it and the test bench can include some simulated directive sometime.

These are called tasks also, simulated directives, these are very common tasks, display it starts with a dollar symbol, display, monitor, dumpfile, dumpvars, finish. These you have already seen in our example which you saw earlier and again, I am repeating that you need to write a test bench only when you are carrying out simulation. So, if your objective is to synthesize a design then you do not need a test bench, right.

(Refer Slide Time: 11:53)

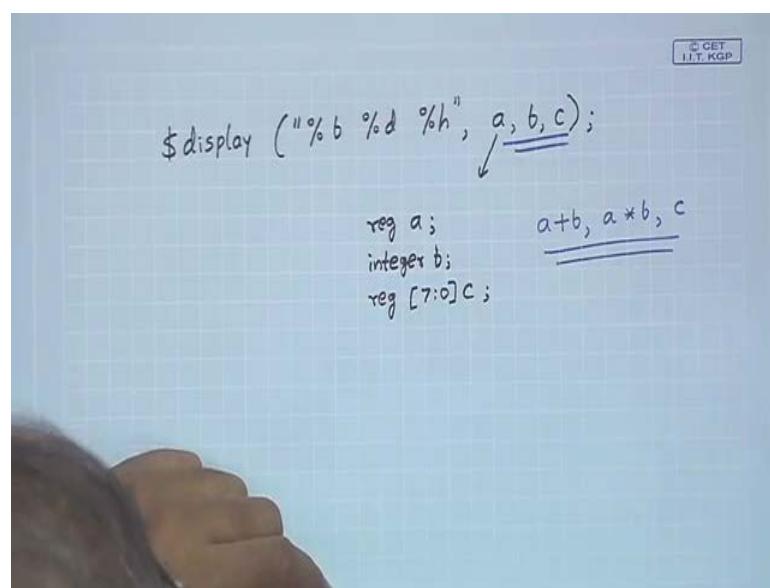
The Simulator Directives

- **\$display ("<format>", expr1, expr2, ...);**
 - Used to print the immediate values of text or variables to stdout.
 - Syntax is very similar to "printf" in C.
 - Additional format specifiers are supported, like "b" (binary), "h" (hexadecimal), etc.
- **\$monitor ("<format>", var1, var2, ...);**
 - Similar in syntax to \$display, but does not print immediately.
 - It will print the value(s) whenever the value of some variable(s) in the given list changes.
 - Has the functionality of *event-driven* print.

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

Let us now explain the meaning of the simulator directives. The first similar directive we explain is display, see display is exactly similar to the printf command that you have in language like C. In C whenever you want to print the value of some variables, you give a printf, you specify the list of the variables you want to print and also you specify a format string that tells you how you want to print. So, exactly the same way in the display directive, dollar display within bracket, within double quotes, you first specify the format string, then you specify the variables or the expressions that you want to print or display. So, some examples I am giving.

(Refer Slide Time: 12:55)



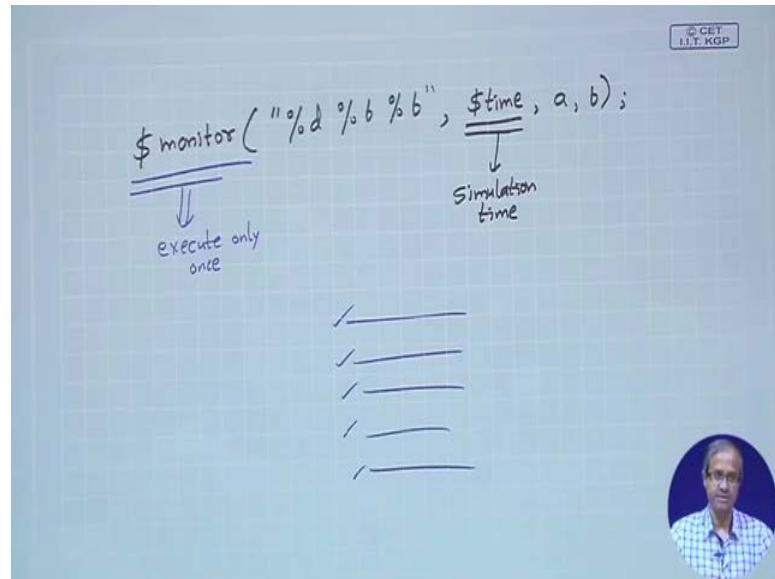
So, you can give a command like this; [\$display ("%b" %d %h, a, b, c)] dollar display within double quote you can say percentage b, percentage d, percentage h, suppose I specify the names of 3 variables a, b and c, where this a can be a bit variable, I can declare them; let us say as a reg a, single bit variable; b can be lets an integer b, can be an integer variable.

So, I printed using the integer description d and h is a hex a hexadecimal. So, let us say C can be a vector that say C is an 8-bit vector. So, I am saying you display this 8-bit number in hexadecimal form. So, this will be displayed in hexadecimal. This is, what is meant by display, you specify a list of it not only variables, you can also have an expression like instead of a, b, c, you can also write I want to print a+ b; I want to print a*b, multiplied b and then I want to print c, I can also write like this. So, any expressions in general, not necessarily only variables, right.

So, this display directive will be printing the variables whenever displays encountered or whenever it is executed, used to print the immediate values, whatever are the immediate values. So, whenever display command is encounter that line is encountered, whatever values of these variables are there at that point in time, they will be immediately displayed, fine.

So, as I said you can have specifies like binary, hexadecimal and so on. Well you have another directive which is very similar to display monitor. So, the format is also quite similar syntax. So, initially within double quotes you give the format how you want to display then you specify some. Well in monitor you typically give variables not expressions because you want to monitor the value of some variables, but there is one difference from display. Display will print these values as soon as it is encountered, but monitor will not print the values immediately, it will print the values whenever the value of at least one of the variables change.

(Refer Slide Time: 15:55)

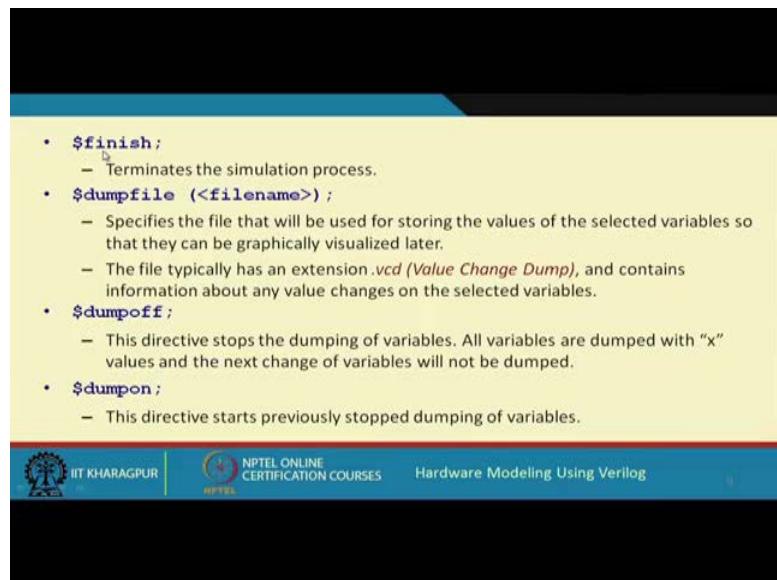


There is a concept of even driven printing approach like suppose I say dollar monitor percentage d, percentage b, let us say I first want to display my time, simulation time, this dollar time means, the present simulation time, this is a special variable, it indicates the current simulation time.

Now, let us say I want to display 2 scalar variables a and b, now monitor means, it will not print the values immediately, you execute this monitor only once, this monitor statement will execute only once. This you can give inside an initial statement only once, but when you see the output which we will be generating, you may see that output will be generating many lines. So, whenever any one of these variables change, there will be a one line of output generated. So, you can see how the variables are changing with time during simulation.

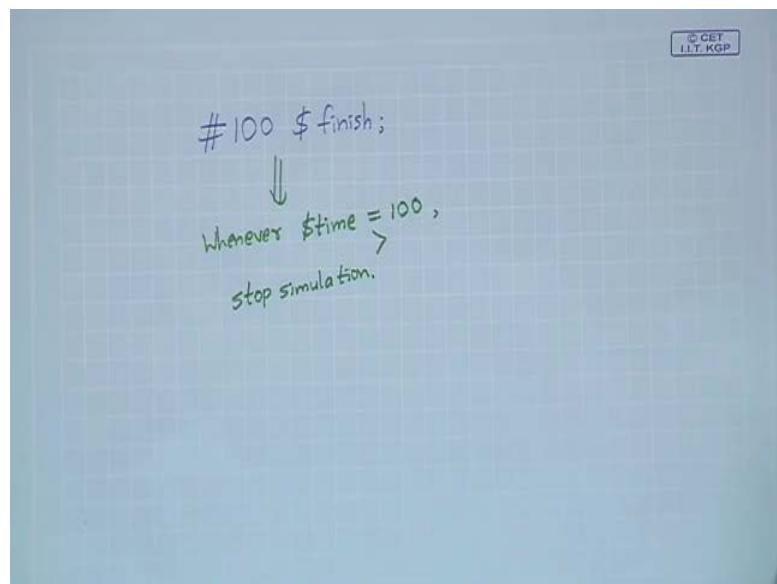
So, this monitor command is very suitable or handy in such situation, right. So, this is the display or this is the mainly the difference between display and monitor. So, one is an immediate print other is an event driven print.

(Refer Slide Time: 17:34)



Now, whenever you want to finish this simulation, you give this command dollar finish typically, you give a delay before that let us say #100 \$finish; that means, at time 100. So, whenever this simulation time is has reached 100, you finish.

(Refer Slide Time: 17:57)

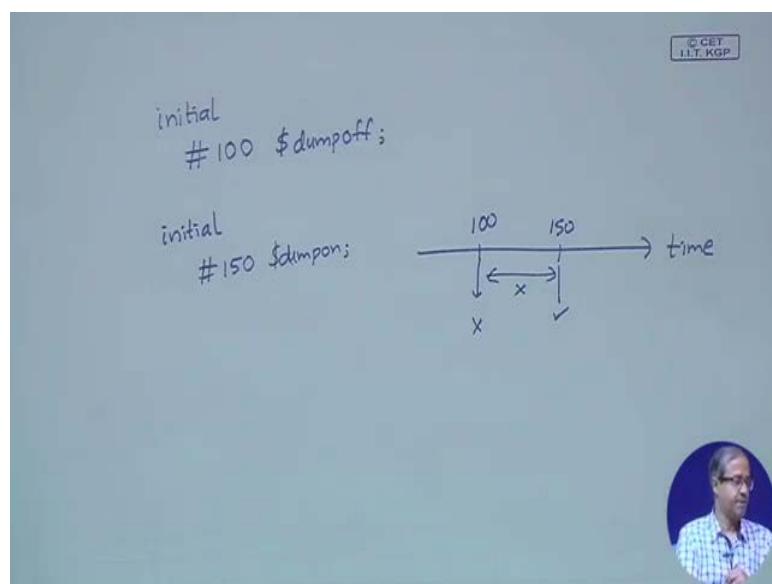


Like you can give as I said #100 \$finish. So, what will this statement mean; this statement will mean that whenever the current simulation time which is held in the variable dollar time, this equals to 100 or is exceeding 100 or it is greater then stopped simulation, the meaning is this, right.

So, there is another command called dumpfile where you can specify the name of a file in the argument. So, here you are actually specifying that where to dump the values of all the variables which you can view later. So, this file name will specify, specifies the name of that file which will be used to store the values of some selected variables, we shall see how to select them variables. There is another command for that some variables will be dumped into the file so that you can see them later in a suitable way and this file which is specified here this typically has an extension dot vcd [.vcf].

This is quite commonly used file format, this is the acronym for value changed dump and this file contains information about all changes that take place in the variables that you have specified and during simulation wherever you encounter dollar dumpoff this dumping into the file will stop and whenever you again encounter dumpon, so, again this dumping will start like for example, if you give a command like this lets say suppose you give initial 100 dumpoff.

(Refer Slide Time: 20:12)



There is another initial block, let us say you give 150 dumpon, this will mean that in the axis of time so, whenever time has reached, this is time. So, whenever time has reached 100, you stop the dumping temporarily pause, then again when the time has reached 150; you again resume dumping, dumpon. So, you can for certain period of time you can stop the dumping operation if you want, all these controls you have with you, right fine.

(Refer Slide Time: 21:09)

- **\$dumpvars (level, list_of_variables_or_modules);**
 - Specifies which variables should be dumped to the .vcd file.
 - Both the parameters are optional; if both are omitted, all variables are dumped.
 - If *level=0*, then all variables within the modules from the list will be dumped. If any module from the list contains module instances, then all variables from these modules will also be dumped.
 - If *level=1*, then only listed variables and variables of listed modules will be dumped.
- **\$dumpall;**
 - The current values of all variables will be written to the file, irrespective of whether there has been any change in their values or not.
- **\$dumplimit (filesize);**
 - Used to set the maximum size of the .vcd file.

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

So, dumpvars actually tells you, see here we have seen dumpfile, means you are saying some selected variables will be dumped, but which variables, that you can specify in dumpvars; dumpvars. How we can specify, level is something which can be 0 or 1; say if level is equal to 0, if I give 0 in the first parameter, then in second parameter I give the name of a module, then all the variable name of module or list of modules, then all the variables within the modules will be dumped.

(Refer Slide Time: 21:57)

\$dumpvars (0, module1, module2);
 ↓ ↓
 ✓ ✓
\$dumpvars (1, a, b, c, d);

Like I can specify it either in this form, I can give dumpvars, the first parameter I can give 0, level equal to 0, then I can give a list of modules, I can write let us say module1, module2, like these whatever modules are there.

So, what this will do? So, all the variables that are inside module1. all the variables such as there is in model2, they will all be dumped. dumpvars 0 means what, means that. And if I write dumpvars 1, then I can give a list of name of variables a, b, c, d, the variables that I want to dump. So, you see. So, if level equal to 0, then all variables within the modules from this list will be dumped; if level equal to 1, then only the listed variables will be dumped, right. This is the main difference and if you give dollar dumpall without giving in a list, then all variables will be written to the file.

So, irrespective of there is a change in the variable or not and sometimes when you are just handling a very complex design and your simulation is going on for a long time, your dump file may become very large. So, there is also a facility where you can specify that well I will not allow my dump file to cross a certain limit, like I can say it should not be more than 5 megabytes let us say. So, I can specify a file size limit in by using the command dumplimit, dumplimit file size, this can be used to set the maximum size of the dump file, right.

(Refer Slide Time: 23:58)

```
timescale 1ns / 100ps
module comparator (x, y, z);
    input [1:0] x, y;    output z;
    assign z = (x[0]&y[0]&x[1]&y[1]) |
               (~x[0]&~y[0]&x[1]&y[1]) |
               (~x[0]&~y[0]&~x[1]&~y[1]) |
               (x[0]&y[0]&~x[1]&~y[1]);
endmodule
```

A Complete Example :: 2-bit equality checker

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

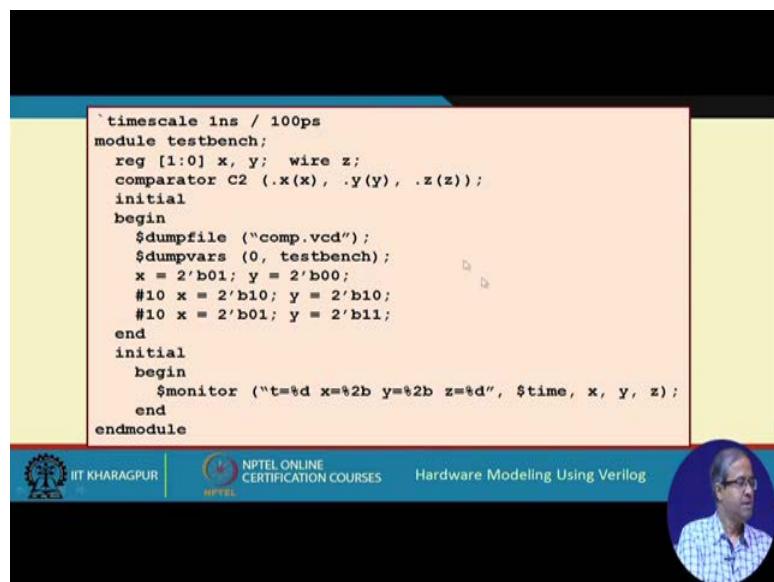
So, let us take a small example. So, here we are taken an example of a 2-bit equality checker, well we talked about the time scale command earlier you remember. In this time

scale which appears in the beginning of a module, the first parameter indicates the unit, like in the delay whenever if I write hash 5, it means 5 into 1 nanosecond and this second one, this 100 ps, picoseconds, this means the resolution. This simulation will be carrying out the or will be computing the results correct up to 100 picoseconds.

So, this is just module comparator which compares whether 2 numbers are equal or not. There is one number is $x[1] x[0]$ other is $y[1] y[0]$, you see this assign z equal to there are 4 conditions separate by OR this, or this, or this. So, what does the first condition say it says $x[0]$ AND $y[0]$ AND $y[1]$ AND $x[1]$, it means all of them are 1, that means, this x is also 11 y is also 11; second one says $x[0]$ is NOT, $y[0]$ is NOT, $x[1]$ is 1, $y[1]$ is 1, which means 10 and 10 both are 10, 10 and this is all are NOT, NOT, NOT; that means, both all are 00, x is also 00, y is also 00 and here $x[0]$, $y[0]$ are 11, $x[1]$, $y[1]$ are NOT; that means, 01, 01.

So, for all the 4 combinations if they are equal, z will be 1, just we are writing down the expression for that.

(Refer Slide Time: 25:50)



```

`timescale 1ns / 100ps
module testbench;
    reg [1:0] x, y; wire z;
    comparator C2 (.x(x), .y(y), .z(z));
    initial
    begin
        $dumpfile ("comp.vcd");
        $dumpvars (0, testbench);
        x = 2'b01; y = 2'b00;
        #10 x = 2'b10; y = 2'b10;
        #10 x = 2'b01; y = 2'b11;
    end
    initial
    begin
        $monitor ("%t=%d x=%2b y=%2b z=%d", $time, x, y, z);
    end
endmodule

```

The screenshot shows a Verilog testbench code. At the top, a blue bar displays the code. Below it, a yellow background contains the Verilog code. At the bottom, there is a red bar with the IIT Kharagpur logo, the NPTEL logo, and the course title "Hardware Modeling Using Verilog". To the right of the red bar is a circular portrait of a man.

This is a simple test bench using the commands that we have mentioned, you see we have given a time scale at the beginning. This is the module test bench, this is the comparator module, this was comparator, we are giving it name C2, 2-bit comparator. Well, this is the recommended style of including or instantiating a module, I mentioned earlier also instead of just writing x, y, z.

It is good to write using dot the name of these variables and the variables that are using here in well. Here in this example the names are the same, but you could have used the names a, b, c also or you could have included the variables in a different order, like you put z first, then x, then y; like that it is always good to specified like this. So, see here we have given 2 initial blocks, in the second initial block there is only a monitor statement. You see, $t = \%d$, we are displaying the time then x, y, z, 2'b means bit within a 2 space.

So, the first one will be blank. So, x, y, z will be, z is integer, you see z, it is output. We are printing as d; this you could have also given as b; no problem and this x and y are 2-bit number that said 2'b; 2'b. And here we have given a dumpfile specified file name, what the values will be dumped and 0 comma name of the module, test bench is a name of the module, inside test bench whatever variables are declaring and the modules you are instantiated, all those variables will be dumped then my stimulus. First I am applying $x=2'b01$, $y=2'b00$; then after time 10, I am applying $x=2'b10$, $y=2'b10$; then again after time 10, I apply $x=2'b01$, $y=2'b11$. So, only for this second one there would be a match, right.

So, with this I come to the end of this lecture. In the next lecture, we shall actually be seeing some examples of the constructs that we have seen in this lecture; how you can use them and what are the different ways you can generate or apply the test stimulus to a design under test. This we shall be discussing in the next lecture.

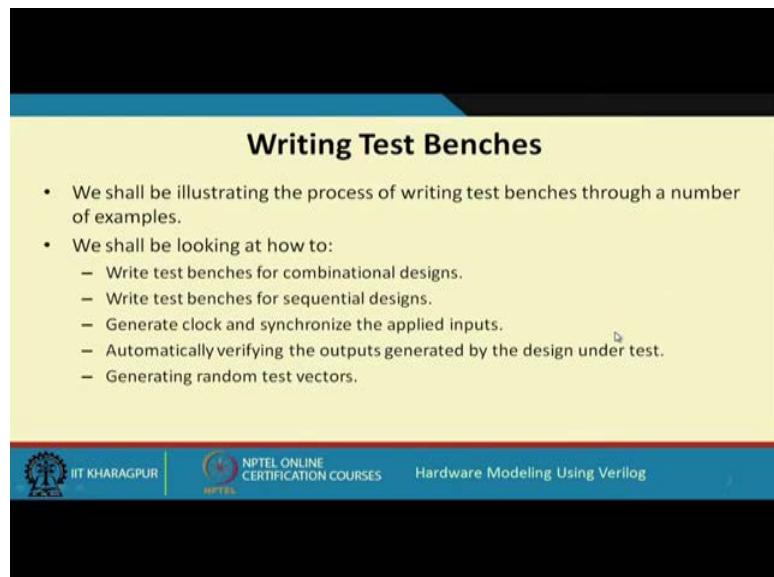
Thank you.

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

Lecture - 22
Writing Verilog Test Benches

So, continuing with our previous lecture, in this lecture, we shall actually be seeing some examples and we shall see how we can write test benches in various different ways. So, writing Verilog test benches is the topic of the lecture.

(Refer Slide Time: 00:38)



Writing Test Benches

- We shall be illustrating the process of writing test benches through a number of examples.
- We shall be looking at how to:
 - Write test benches for combinational designs.
 - Write test benches for sequential designs.
 - Generate clock and synchronize the applied inputs.
 - Automatically verifying the outputs generated by the design under test.
 - Generating random test vectors.

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

So, in this lecture, as I said, we shall be illustrating various ways of applying the stimulus using examples. Specifically, we shall be looking at the few things; the following things; we shall see how we can write test benches for combinational designs, not only that how we can apply the test patterns; how we can generate the test patterns. Then similar things for sequential designs, generation of the clock and synchronization of the inputs is very important; how we can do that. And also we shall talk about that how could automatically verify the output generated.

Like for example: let us say; I am designing an adder my test bench should be such that it is applying the inputs, it is reading out the output and it will automatically compare whether the output is coming correctly or not and it will just tell me that the design is good or bad. So, it will not just show me a list of all the inputs and outputs, it can also

tell me whether the output is coming as per the specification or not and lastly we shall also look at how to generate random test vectors which may be useful in many test bench applications, fine.

(Refer Slide Time: 02:07)

Example 1: Full Adder

```
module full_adder (s, co, a, b, c);
    input a, b, c;
    output s, co;
    assign s = a ^ b ^ c;
    assign co = (a & b) | (b & c) | (c & a);
endmodule
```

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



Let us start with a very simple example; the simplest example a full adder. So, we have a behavioral description of the full adder using assign statement. So, s and co are the sum and carry and a, b, c are the inputs.

(Refer Slide Time: 02:30)

```
module testbench;
    reg a, b, c; wire sum, cout;
    full_adder FA (sum, cout, a, b, c);

    initial
        begin
            $monitor ("%time,%a=%b, %b=%b, %c=%b, %sum=%b, %cout=%b",
                      a, b, c, sum, cout);
            #5 a=0; b=0; c=1;
            #5 b=1;
            #5 a=1;
            #5 a=0; b=0; c=0;
            #5 $finish;
        end
    endmodule
```

0 a=x, b=x, c=x, sum=x, cout=x
5 a=0, b=0, c=1, sum=1, cout=0
10 a=0, b=1, c=1, sum=0, cout=1
15 a=1, b=1, c=1, sum=1, cout=1
20 a=0, b=0, c=0, sum=0, cout=0

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

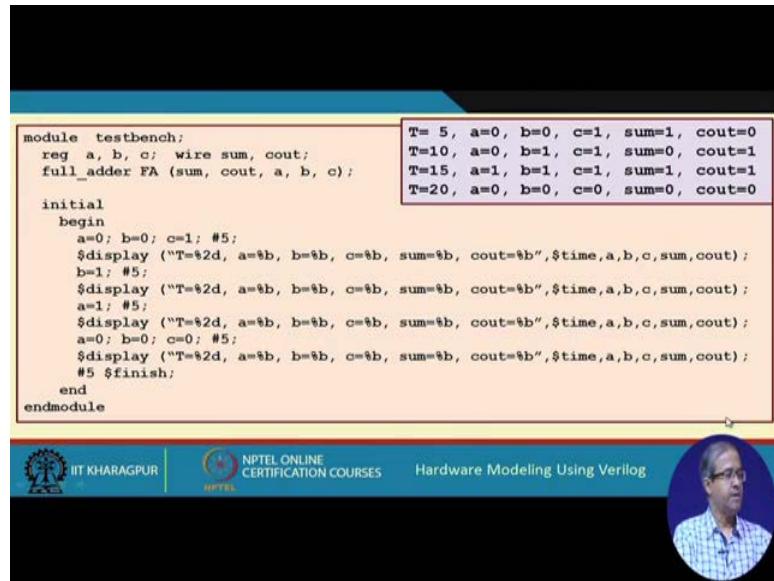


So, sum is the XOR and carries ab OR bc OR ca. So, let us see different ways of doing it, this is the first approach and this kind of test bench, you have already seen earlier. So, what we have done? This was our full adder module, we have instantiated the full adder module, we call it FA. So, we have specified the parameters; the same order sum, carry, a, b, c; here we are calling it sum, cout, carry out; a, b, c are the inputs. So, the inputs were declaring as reg as usual and the output sum and cout, we are declaring as wire. So, here because it is a combinational circuit, we are making it simple, we are using a single initial block.

So, we are executing monitor only once, here you are saying we want to display, whether also give it in the beginning dollar time, comma, you can also give it like that. So, there the time will be displayed in the beginning, like this, then within quote a equal to, b equal to, c equal to, sum equal to and cout equal to, all b means binary. We are displaying these 5 things. So, you recall monitor means these 5 variables who will display it whenever at least one of the variables they change state, fine. So, the first input, I am applying is a=0, b=0, c=1, then I change b to 1. So, delay is 5. So, after delay of 5, I am doing this, then a =1 and last factor is 0,0,0 and then after another 5, I finish. So, let us see the simulation output what we get on the screen if we run it.

So, initially; so, whenever the time is initialized to 0 that will also be a change. So, a, b, c some x; all are undefined, all are x. At time 5 I have initialized, at time 5, a becomes 0, b=0, c=1, my sum becomes 1, carry is 0; then 10, I mean b =1, so now, b is 1, a=0, b=1, c=1, so now, sum is 0, carry is 1. So, again after time 5, a is 1, this is at time 15, a is also become 1; 1, 1, 1. So, sum is 1, carry is 1 and at the end all are 0; 0, 0, 0 sum is 0, carry is also 0.

(Refer Slide Time: 05:09)



The screenshot shows a Verilog testbench code and its corresponding simulation results. The code is as follows:

```
module testbench;
  reg a, b, c; wire sum, cout;
  full_adder FA (sum, cout, a, b, c);

  initial
    begin
      a=0; b=0; c=1; #5;
      $display ("T=%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b", $time,a,b,c,sum,cout);
      b=1; #5;
      $display ("T=%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b", $time,a,b,c,sum,cout);
      a=1; #5;
      $display ("T=%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b", $time,a,b,c,sum,cout);
      a=0; b=0; c=0; #5;
      $display ("T=%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b", $time,a,b,c,sum,cout);
      #5 $finish;
    end
  endmodule
```

The simulation results show four lines of output:

T	a	b	c	sum	cout
5	0	0	1	1	0
10	0	1	1	0	1
15	1	1	1	1	1
20	0	0	0	0	0

The interface at the bottom includes the IIT Kharagpur logo, NPTEL Online Certification Courses logo, and the course title "Hardware Modeling Using Verilog". There is also a circular portrait of a man.

So, this is a very simple way of generating these tests. Now the same example I am showing in a slightly different way, since earlier case I use a monitor, of course this is more convenient, but I am just illustrating the use of display. Instead of monitor here what I am doing using same full adder instantiation, I am just declaring inputs are reg, the outputs are wire. Here I apply the first input a=0, b=0, c=1, then I apply a delay of 5, then I display, what, time a, b, c, sum, cout; T=%2d, a, b, c, sum, cout; these are all b, binary; display will immediately display this line t=5, a=0, 0, 1, 1, 0, then b=1, again a delay of 5. So, now, b becomes 1, again after a delay of 5, I put another display, same line.

So, the next line is displayed, then a=1; there is again a delay of 5; again display third line, then again a=0, b=0, c=0, again delay of 5, again display fourth line and then finish. So, see that if you give monitor, we need to give only once and whenever the variable changes, this simulator will automatically understand that well we are supposed to monitor the variables and we want to print it and display means, wherever I want the values to be printed I insert a display command like that just like printf in C. So, in this example, I have inserted 4 printf statements corresponding to the 4 display statements and for the display statements 4 lines are printed this is the difference between monitor and display. So, in a test bench I can use display if I want, I can use monitor if I want. See one drawback of monitor is that if over a long period during simulation variables do not change their values nothing will get printed.

But I want to see that every, after every gap of fifty what is the outputs coming I want to see then I want to give a display that irrespective of the values whether they are changing or not changing I want to see their values then I have to use the display command rather than monitors, all right.

(Refer Slide Time: 07:54)

```

module testbench;
reg a, b, c; wire sum, cout;
integer i;
full_adder FA (sum, cout, a, b, c);

initial
begin
  for (i=0; i<8; i=i+1)
    begin
      {a,b,c} = i; #5;
      $display ("T=%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b",
                $time, a, b, c, sum, cout);
    end
  #5 $finish;
end
endmodule

```

T= 5, a=0, b=0, c=0, sum=0, cout=0
T=10, a=0, b=0, c=1, sum=1, cout=0
T=15, a=0, b=1, c=0, sum=1, cout=0
T=20, a=0, b=1, c=1, sum=0, cout=1
T=25, a=1, b=0, c=0, sum=1, cout=0
T=30, a=1, b=0, c=1, sum=0, cout=1
T=35, a=1, b=1, c=0, sum=0, cout=1
T=40, a=1, b=1, c=1, sum=1, cout=1

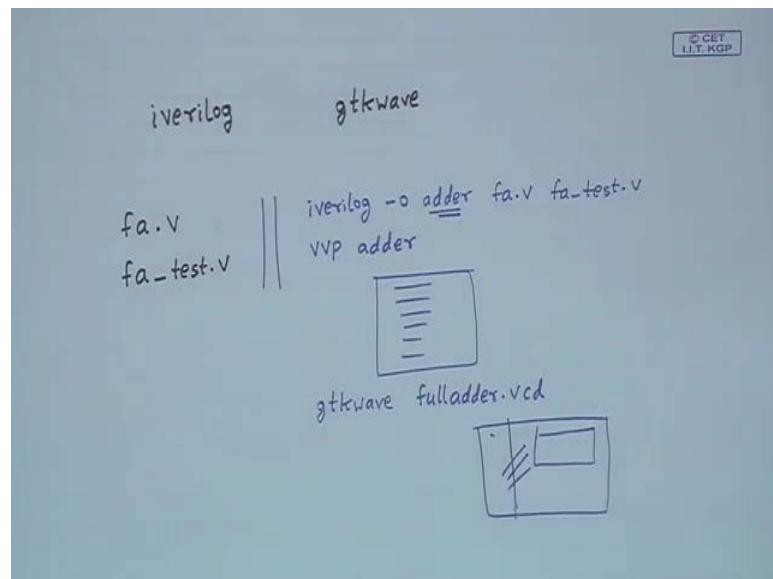
Now, the third example same the full adder here, let us say I want to generate all 7 inputs, like the simulation output let us see, then and see how we have done it, say it starting from 000, 001, 010, 011, 100, 101, 110, 111; all 8 combinations I want to generate and I want to print the outputs. Here I have used a for loop for the purpose, well how have we done this, it is quite simple, we have declared an integer of type i because this a, b, c, these are 3 variables, alright, but we can treat them as a 3-bit number. If we concatenated them a, b, c taken together like this within this concatenation operator, they can be treated as a 3-bit number.

So, in 3-bits numbers can go from 0 up to 7. So, I am writing this for loop as for $i=0$ up to $i<8$; that means, up to $i=7$; at the end of every loop $i=i+1$. So, $i=0, 1, 2, 3, 4, 5, 6, 7$ and I just write a statement like this $\{a, b, c\} = i$. So, this will automatically convert this number i into binary and assign the corresponding last 3-bits to a, b, c; this will be done by the simulator automatically. So, the values will get assigned automatically, this like this and after delay of 5, I am displaying. So, in the display the time is showing us 5, 10, 15 like that, I am displaying the current time a, b, c and sum and cout.

So, you can see this is quite convenient because the second approach was a little cumbersome, I had to do so many display statements, but in a loop if I used, it is displaying, I am just writing display only once, but in a loop the patterns are getting generated automatically, there are 8 patterns I am generating 0 to 7, right. Now this is the same for loop version, but we have just included dumpfile and dumpvars command, now I want to dump them to a file also. So, we want to see the simulation output in addition, this is the change we have done.

So, let us see we have specified that we want to dump them into a file whose name is full adder dot vcd and we have given a statement dumpvars 0 comma test bench. So, what is 0 comma test bench means, 0 comma test bench means that in this module whose name is test bench, whatever variables are there you dump all of them. So, when you write dumpvars 0 comma test bench. So, here the variables are a, b, c sum and cout, all these 5 variables will be dumped.

(Refer Slide Time: 11:28)

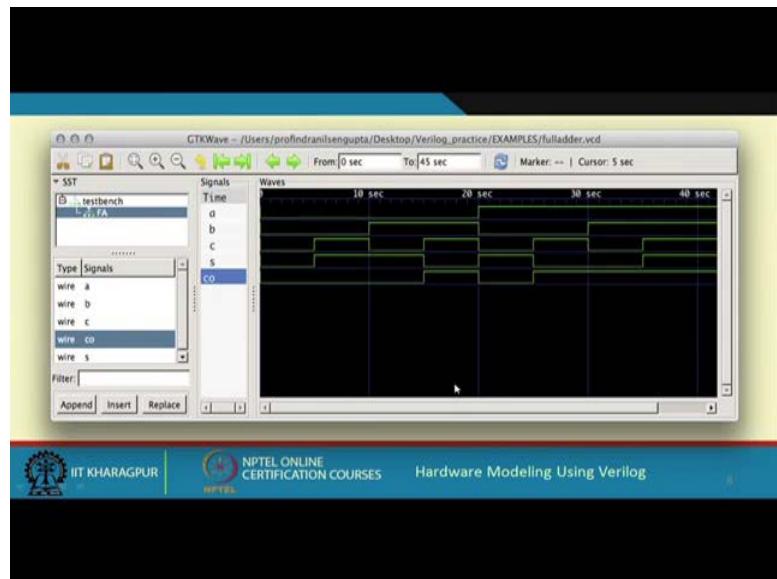


So, well just I recall one more thing that I asked you earlier that you can carry out simulation using the Iverilog tool, the Ikaras Verilog and you can view the waveforms using gtkwave tool. So, just for this example, so let us say, my original file was fa.v and the test bench that I wrote was fa_test.v. So, for simulation what are the commands I am giving first, I am writing or I am invoking iverilog this is a command line parameter. So, I have to open a command window, I have to give it. So, I have to give an output file

name, let us give the output file name as adder then I give the name of these 2 files fa.v and fa_test.v, but after this, well if there any errors it will show errors; there is no errors then the next command will have to give is vvp, this file which is generated adder.

So, once you give this then this simulation output will be generated, you can see it on this screen. Now if you want to see it on the waveform, you want to see the waveforms then you will have to invoke gtkwave, gtkwave, see here we have given a dump file called fulladder.vcd. So, you will have to give the name of the vcd file, gtkwave fulladder.vcd, and then the window will be opened. So, you can click here, you can pull down the variables you want to display and you will see the waveforms here.

(Refer Slide Time: 13:48)



Let us see how. So, the first thing is that once you simulate it, once you give the vvp command, you will be getting the simulation output like this.

Now, when you run gtkwave, you will be getting the waveform like this. Let us try to understand this waveform generation here. You see on the left, here you see the test bench which is the name of the module, under it there was a module which is instantiated call FA, full adder. The variables which are there, they are all mentioned here a, b, c, co and s and these 5 variables are being displayed in the axis of time. Let us go back to the previous one, you see this was the simulation output, at time 5 you applied a, b, c, 0 0 0; at 10, we applied 0 0 1; 15, 0 1 0 and so on.

Let us see whether it is happening, this is my time $t=0$, you see a, b, c, they are all 000 here; at time $t=5$, this c is becoming 1; at time 10, b is becoming 1 and c become. So, 000; 000; here it is 001; here it is 010; then it is 011; then it is 100, four; 101 five; 110 six; like that it will go on, right and the sum and carry you can select and let us for example, if it is let us say 11; here a is 011; 011.

So, sum is 0 and carry is 1 and here 111; a, b, c all are 1, sum is 1, carry is also 1. So, you can actually look at the waveform and you can verify the functionality, you can see the time also, right.

(Refer Slide Time: 15:40)

Example 2:
4-bit shift register

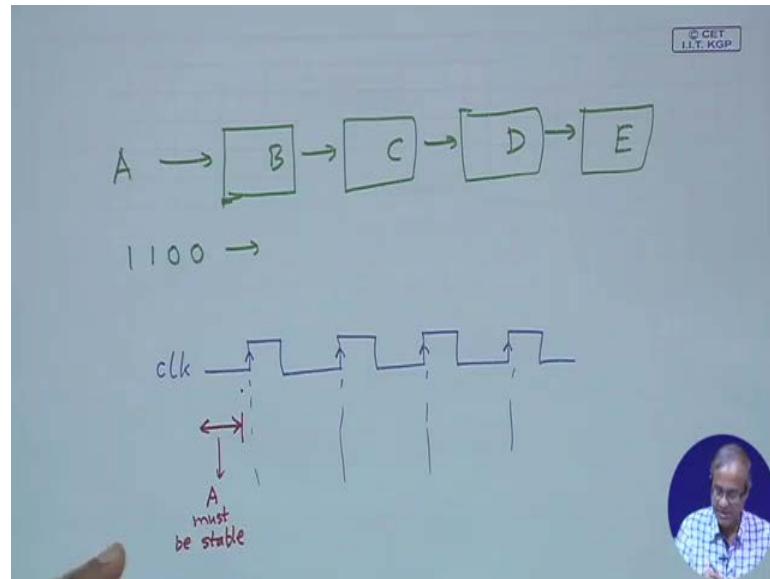
```
module shiftreg_4bit (clock, clear, A, E);
    input clock, clear, A;
    output reg E;
    reg B, C, D;
    always @ (posedge clock or negedge clear)
        begin
            if (!clear) begin B<=0; C<=0; D<=0; E<=0; end
            else begin
                E <= D;
                D <= C;
                C <= B;
                B <= A;
            end
        end
    endmodule
```

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

Let us take another example which is a sequential circuit, this is a 4-bit shift register. Now earlier we have seen that how we can use the non-blocking assignment statement like this to create a shift register. So, I am showing the same example here. So, it is a 4-bit shift register where the input is A and the final output is E and the intermediate flip-flop outputs are B, C and D. So, the description was I am repeating. So, whenever there is a positive edge of the clock or the clear is negative it is negative clear, 0 clear, you execute this block.

So, if the clear is 0, then you initialize all the flip-flops to 0; else you carry out a shifting, D will go to E, C will go to D, B will go to C, A equal to B, all together because this is non-blocking.

(Refer Slide Time: 16:49)



So, the 4 flip-flops they are appearing. So, they are B, C, D and E; this is B, this is C, this is D and this is E and you are applying A from outside, right, this is how the shift register is supposed to work.

(Refer Slide Time: 17:15)

```
module shift_test;
    reg clk, clr, in;    wire out;    integer i;
    shiftreg_4bit SR (clk, clr, in, out);

    initial
        begin clk = 1'b0; #2 clr = 0; #5 clr = 1; end
    always #5 clk = ~clk;

    initial begin #2;
        repeat (2)
            begin #10 in=0; #10 in=0; #10 in=1; #10 in=1; end
    end

    initial
    begin
        $dumpfile ("shifter.vcd");
        $dumpvars (0, shift_test);
        #200 $finish;
    end
endmodule
```

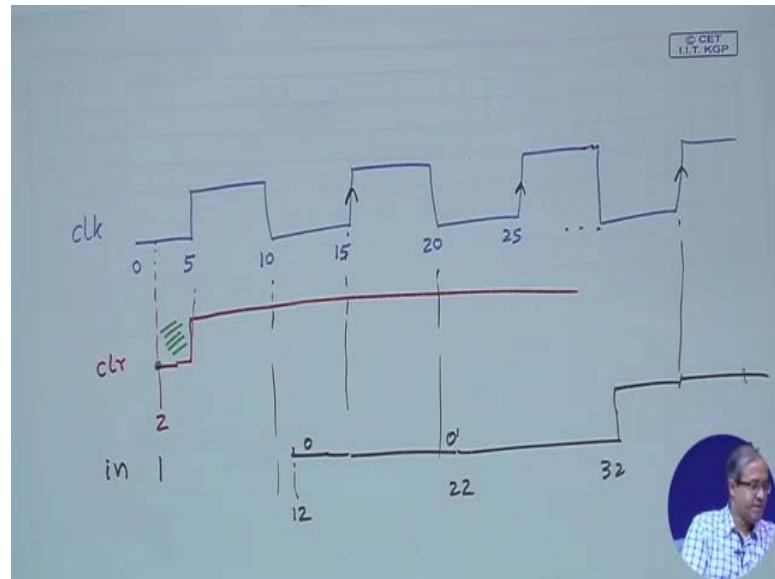
Now, for this; we want to write a test bench, this is a test bench we wrote. Let us try to understand what we are trying to do here, see in this shift register, the inputs are clk and clr and the other side the outputs A, B, C, D.

So, we have instantiated this shift register clk and clr, this A and E, here we are calling it in and out, right, fine. So, this part I am explaining here, let us see what we are trying to do, we are trying to apply some input 0, 0, 1 and 1. Like we have a 4-bit shift register, we are trying to apply first a 0, then a 0, then 1, then 1, and we want to shift these 4-bits inside the register. Now you see one thing we have to ensure in the shift register description where here you assume that is triggered at the positive edge of the clock, pause edge, here I am calling it clk. So, you see suppose my clock signal is coming like this. So, the shift register is supposed to be shifting at the positive edge of the clock. So, what I must ensure is that before the positive edge comes, before that here, before that my input must be stable, here I have only one input A; A must be stable .

So, what I mean to say is that it must not be like this that the signal A, let us say this clock we are applying and at the same time, this signal A which is there; it should not happen that A is also changing at the same time the clock is coming because whenever the clock is coming if A is also changing at the same time, there will be confusion because this A and the clock changing may be the previous value will be taken. So, A must be applied a little before the clock is coming, stabilized and then you apply the clock, then only the correct value of A will be shifted into register, right, this is why I mean, I mean to say.

So, now, let us see this. So, what I have done, here I have initialized clock to 0 because I am not given in delay it is at time $t=0$; at time 2, I have set clear to 0; at time 5, I again has set clear to 1. So, I have cleared the register at time 5, because you see clear was a negedge clear. So, here it was negedge at time 0, at time $t=0$. the counter be cleared. So, at 5, again I am releasing the clear, making it 1, well and then I am saying at 5, $\text{clk} = \sim\text{clk}$.

(Refer Slide Time: 20:49)



So, let us try to see what is happening here, I am generating a clock signal, like this, this is clock. So, at time 0, I am making it, at 0 this is time 0 and at 5 always at after delay 5, $\text{clk} = \sim\text{clock}$. So, after delay of 5, I am making it 1; after another delay of 5, 10, I am making it 0 again; at 15, I am making it again 1; 20 again; 25 and so on like this; the clock is changing, right. Now here in between what I have done, I have set clear to 0 at time 2.

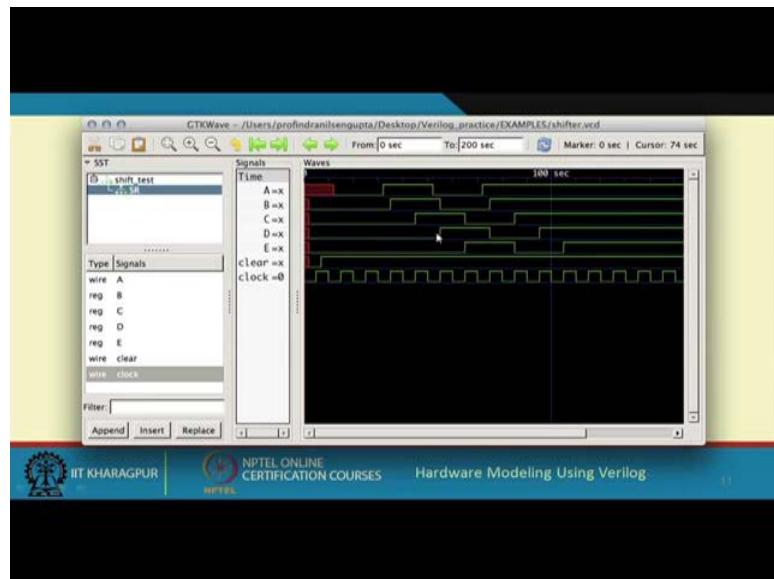
So, at time 2, let us say somewhere here; this is time 2, I have made $\text{clr} = 0$ and again at time 5, I have made $\text{clr} = 1$; at time 5, I have again made $\text{clr} = 1$ and I leave it at 1. So, what it means is that during this period the counter will be cleared because it has got a 0 edge here, it has become 0 out here, fine. Now what I am doing let us see, these 4-bits 0 0 1 1, I am shifting, in this initial block I am initially giving a delay of 2, right, then after delays of 10, 10, 10, 10, I am shifting the 4-bits 0 0 1 1; what does this mean, let us see now. So, we are giving an initial delay of 2. So, initial delay of 2 means we are here 2, this is 10, this is 20, then you are giving a delay of 10 after that; that means, it will be here at time 12, then again we are giving a delay of 10. So, it will be here 22.

So, you see after the falling edge comes, I am applying the next input here. So, this “in” the first $\text{in} = 0$, I am applying here; at 12, I am applying 0; then at 22, I will be applying again as 0; then at 32, later on I will be applying a 1; then again at 42, later I will again apply a 1; 0 0 1 1. You see the points I am applying, I am ensuring that when the next

clock edge comes the input is absolutely stable, when the next clock edge comes it is stable, you see just look at the next one also 20. So, when the next clock edge comes, this input is stable. So, there is no confusion of this edge with the input varying, right.

So, we have written our test bench in such a way that this delays are automatically adjusted, the inputs are stable well before the clock edge comes; clock edge is coming at 15, but at time 12, my input is stable; clock edge is coming at 25, at time 22 my input is stable; it is coming at 35, at 32 this is stable and here in the last initial block of my code, we have just specified the dumpfile and said that at 200 will finish.

(Refer Slide Time: 24:23)



So, you just see the waveform, this is what I wanted you to see; you see clock, this is the clock signal which I am showing; clear, you see clear there is a small gap here. So, at time 2, it has become 0. So, when it has become 0 all A, B, C, D has become 0s, cleared; A is 0, not A, B=0, C, D, E; this 4 shift register outputs have been cleared. Now the thing will start from time 12. So, this is 10, next clock when it comes you see, here it will start counting, this A is 0; 0 will be shifted.

So, you apply 0; here 0 will be shifted, apply 0 again, it will be shifted, then you apply A=1, 1 will be shifted when we apply A=1, you see these are getting shifted 0 0 1 1, 0 0 1 1, 0 0 1 1, 0 0 1 1, they are getting shifted; A is getting B, B to C, C to D, D to E. So, this will go on. So, you see it this 0 0 1 1, I am repeating 2 times, repeat 2. So, 0 0 1 1, 0

0 1 1 that is why you see 0 0 1 1, 0 0 1 1, then it remains 1 and it nicely gets shifted, right, fine.

(Refer Slide Time: 26:04)

Example 3:
7-bit binary counter

```
module counter (clear, clock, count);
    parameter N = 7;
    input clear, clock;
    output reg [0:N] count;

    always @(negedge clock)
        if (clear)
            count <= 0;
        else
            count <= count + 1;
endmodule
```

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

So, now let us take another example. So, it is a 7-bit binary counter. So, clear, clock and count. So, at negedge clock, whenever clock is going from 1 to 0, if synchronous clear, clear is 1, I am clearing otherwise incrementing. So, the test bench you are writing like this.

(Refer Slide Time: 26:26)

```
module test_counter;
    reg clk, clr;
    wire [7:0] out;

    counter CNT (clr, clk, out);

    initial clk = 1'b0;
    always #5 clk = ~clk;

    initial
    begin
        clr = 1'b1;
        #15 clr = 1'b0;
        #200 clr = 1'b1;
        #10 $finish;
    end

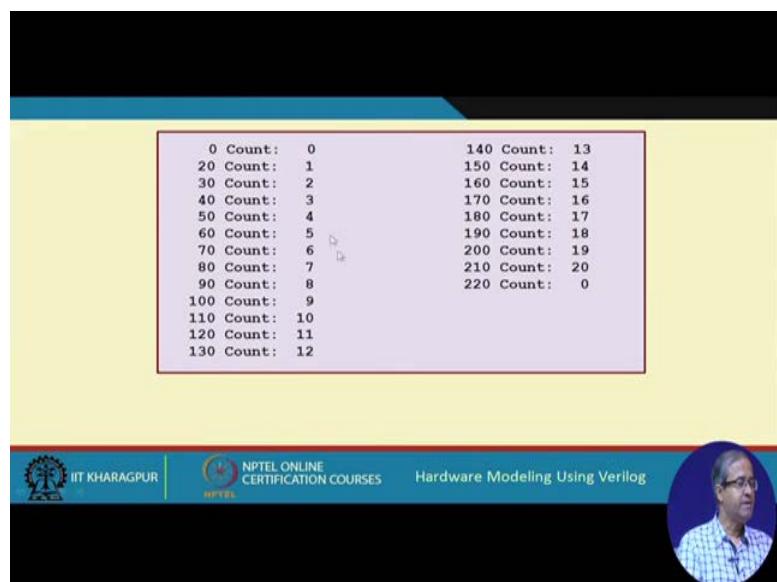
    initial
    begin
        $dumpfile ("counter.vcd");
        $dumpvars (0, test_counter);
        $monitor ($time, " Count: %d", out);
    end
endmodule
```

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

So, for counter it is very simple. So, initially we are saying clock = 1 in one initial block. So, in another always block after delay of 5 clock = ~clock.

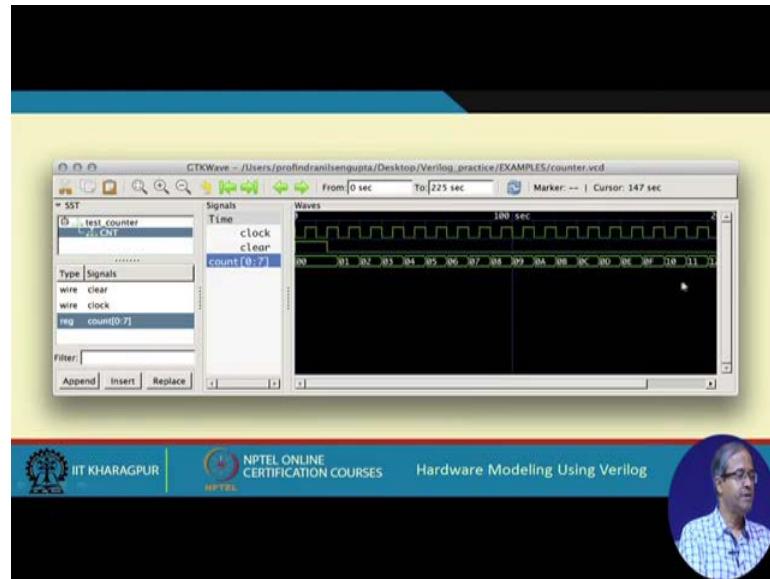
So, in the last initial block, we are specifying the dumpfiles and you are seeing the monitor, we want to see time and count value and here we are specifying that initially we are clearing the counter and at time 15 you are making it 0 again. So, the counting should start from time 15; till then clear was 1 and it will continue to 200. So, 200 again I am setting clear to 1; it will be again cleared; then after time 10, I will finish.

(Refer Slide Time: 27:17)



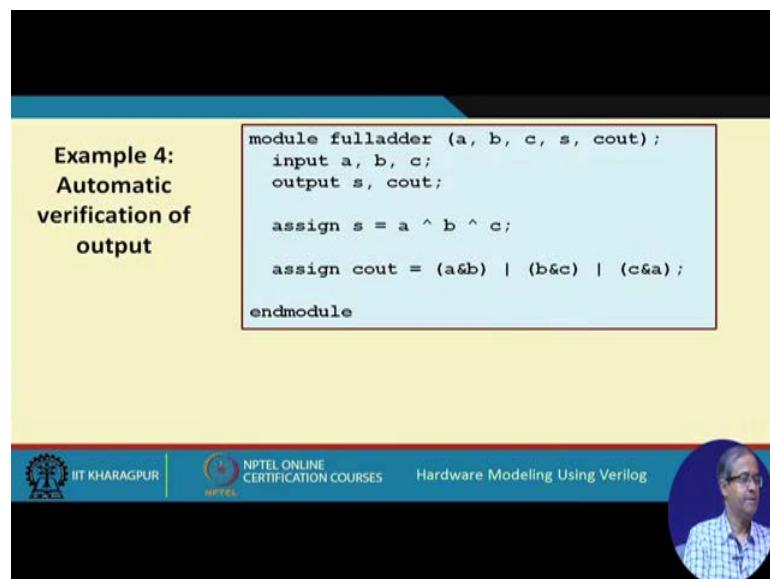
So, if we again simulate this code, similarly you will see that the output generated will be like this, you see you are monitoring time and count. So, there is a time and there is the count, the time will go from 0 and count will start, you see here before 15 count will not start because it is cleared. So, only after 15, next clock comes, it will start counting. So, from 20, the count starts increasing, then clock rate is 10, 30, 40, 50; 1 1 1; it is incrementing.

(Refer Slide Time: 27:57)



So, it will continue till 200; 200 we are again clearing it. So, after 200 after time 10, it becomes 0. So, the timing diagram if you view, it will look like this, you see clock is coming, this is clear; the clear will be cleared at time 15, till then the count value was 0. You see 00 written here, then with every clock falling edge, it becomes 01, next falling at 02, 03, these are the count values, this is shown in hexadecimal 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, and so on, it will continue. So, the counter is counting correctly, you can see, verify.

(Refer Slide Time: 28:38)



Now we are taking an example where we want to verify the output automatically. So, we take the example of the simple full adder again, full adder, sum and carry.

(Refer Slide Time: 28:53)

```

module fulladder_test;
reg a,b,c;
wire s, cout;
integer correct;
fulladder FA (a,b,c,s,cout);
initial
begin
correct = 1;
#5 a=1; b=1; c=0; #5;
if ((s != 0) || (cout != 1))
correct = 0;
end
#5 a=0; b=1; c=0; #5;
if ((s != 1) || (cout != 0))
correct = 0;
#5 $display ("%d", correct);
end
endmodule

```

Shall display 1 if outputs are correct; and display 0 otherwise.

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

Let us say we write the test bench like this, we write the test bench as follows, we instantiate the full adder, we declare a variable called correct of type integer, initialize it to 1 in the initial block, correct = 1; then at time 5, let us say after delay 5, we apply a=1, b=1, c=0; right. Well we give a delay then we check for a, b, c, 1, 1, 0, this sum is supposed to be 0 and carry is supposed to be 1, right. So, we are checking if sum is not equal to 0 [s != 0] or carry is not equal to 1 [cout != 1], then you set the variable correct to 0 then apply the next vector 1 1 1, again give a delay, then if sum is not equal to 1 which is supposed to be, carry is not equal to 1, then again you set the correct to 0 and then the last vector 0 1 0, 0 1 0 supposed to be give s=1 and cout=0. So, if sum is not equal to 1, carry is not equal to 0, correct = 0, at the end you just display correct. So, it will be displaying either 0 or 1.

So, I mean if everything is correct, the full adder design is correct then you should get a 1 in the output, but if any of the outputs are not matching then you should get a 0 as the output, right. So, this is a self-checking test bench, you can verify the output and compare and print this summary result directly.

(Refer Slide Time: 30:32)

Example 5:
Generating
random test
vectors

```
module adder (out, cout, a, b);
    input [7:0] a, b;
    output [7:0] out;
    output cout;

    assign #5 {cout,out} = a + b;
endmodule
```

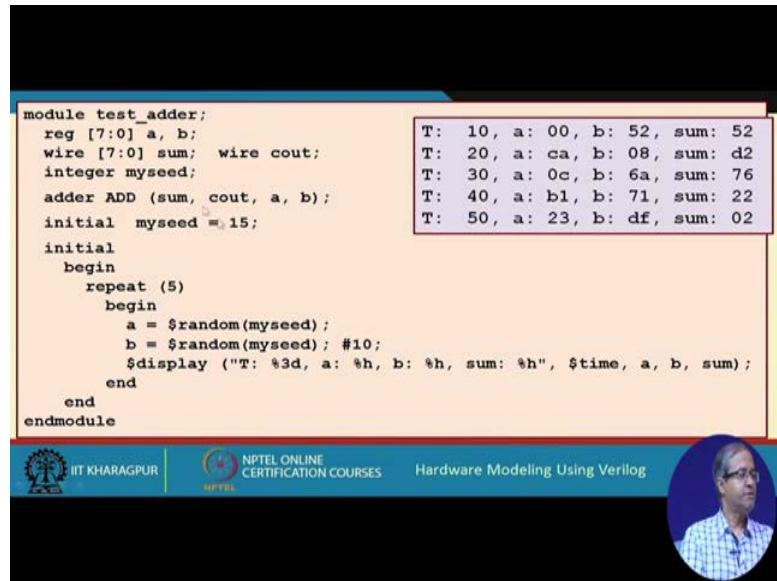
- The system task `$random` can be used to generate a random number.
- It is called as : `$random (<seed>)`
 - The value of `<seed>` is optional and is used to ensure that the same sequence of random numbers are generated each time the test is run.

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

Now, the last example, I take is one of generating random test vectors. Now here let us say we take an adder example that two 8-bit numbers a and b, the output is an 8-bit sum and cout is the carry. So, this is a behavioral description cout and out. The 9-bit sum with the carry is equal to $a + b$. Now we want to test it with some random input pattern a, b. There is a function called dot dollar random [.random] or task sometimes called also, which can be used to generate a random number. This function can be called with a parameter `<seed>` also, this is optional you may or may not give this seed.

So, if you give a particular `<seed>` it means that the same sequence of random number will be generated every time.

(Refer Slide Time: 31:47)



The screenshot shows a Verilog test bench for an adder. The code defines a module `test_adder` with inputs `a` and `b` (reg [7:0]), output `sum` (wire), and output `cout` (wire). It uses an integer `myseed` initialized to 15. An `adder ADD` is instantiated with inputs `sum`, `cout`, `a`, and `b`. In the `initial` block, it repeats 5 times, generating random values for `a` and `b` using `$random(myseed)`, then displaying the time `T`, `a`, `b`, and `sum` in hexadecimal. The displayed results are:

T: 10, a: 00, b: 52, sum: 52
T: 20, a: ca, b: 08, sum: d2
T: 30, a: 0c, b: 6a, sum: 76
T: 40, a: b1, b: 71, sum: 22
T: 50, a: 23, b: df, sum: 02

The slide also includes the NPTEL logo and the title "Hardware Modeling Using Verilog".

So, that you can rerun and reproduce the test, you can verify whether it is working in the same way; but if you do not give a <seed> then it will be really a random thing, if you run it again some other input will come. So, let us see the example, here see we have instantiated the adder add a, b, these are reg; this will be the inputs to the adder and sum I have declare as wire, this cout is also a wire and I declare an integer called myseed. So, which I initialize to 15, initial myseat=15. So, in this initial block I repeat 5 times, I apply 5 random patterns, repeat 5 is what I do, a = \$random(myseed), I generate a random number, put it to a.

So, I put another random number, put it to b, give a delay of 10, then display, what I displayed time t, a, b, in hexadecimal and sum also in hexadecimal. So, you see that time 10, the first random number that that was generated was 00 and 52. So, if we add them, sum is 52; second time, it was ca and 08, it say a and 8 is b, c, d, e, f, 0, 1, 2, so 2 and carry to making d. Similarly, 0c and 6a is 76; b1 and 71, if we add sum will be 22, there will be a carry out; 23 and df, if we add sum would be 02. So, in this way, you can generate random numbers and you can just apply to your design under test.

So, with this we come to the end of this lecture, well, we have seen a number of examples of writing test benches. Of course, this is not the end of thing; we will be taking more examples later and whenever we talk about a design, we shall also be showing the test bench side by side. So, you will always be getting a flavor of something

new on the test benches that will be discussing in the future also, but till then we stop for now.

Thank you.

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

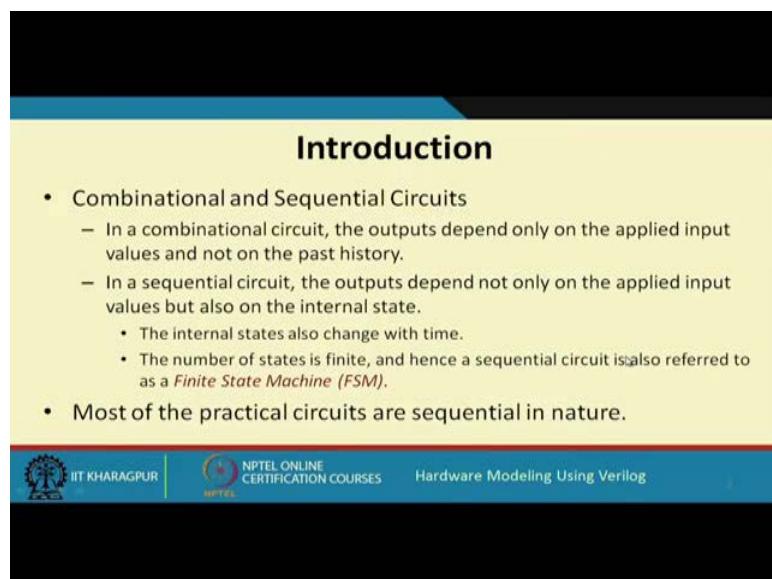
Lecture - 23
Modeling Finite State Machines

So, we have so far looked at the design of both combinational and sequential circuits using Verilog. The combinational circuits that were looked at, were modeled using the continuous data flow kind assign statements, or even using the various kinds of procedural blocks, typically using blocking assignment statements. And the examples of sequential circuits that we saw, were all modeled using procedural assignments both using blocking and non blocking assignments.

Now in this lecture, here we shall be looking again at the modeling of sequential circuits, but not directly from the behavior as we had seen earlier, but in a more formal way from the so called state table or the state transition diagram representation of the sequential circuit.

So, the topic of our discussion today is modeling of finite state machines, ok.

(Refer Slide Time: 01:28)



Introduction

- Combinational and Sequential Circuits
 - In a combinational circuit, the outputs depend only on the applied input values and not on the past history.
 - In a sequential circuit, the outputs depend not only on the applied input values but also on the internal state.
 - The internal states also change with time.
 - The number of states is finite, and hence a sequential circuit is also referred to as a *Finite State Machine (FSM)*.
 - Most of the practical circuits are sequential in nature.

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

So, as I said that you can have two different kinds of circuits; combinational and sequential. So, in a combinational circuit, the idea is that, the output or the outputs will

depend only on the applied input at that point in time. It will not depend on the previous or the past history.

So, as an example you can think about a full adder, where whatever inputs a , b , c were applying right at this point in time, the output sum and the output carry would be generated based on those only. So, it will not depend on the previous history means, what are the a , b , c values, that we had applied in the past, this is what is made by a combinational circuit, the output depends only on the applied inputs. So, in contrast for a sequential circuit, the outputs that are generated, will depend of course on the applied inputs, but also on some of the past history which is represented by the internal state. So, we shall be looking at a number of examples in this regard, to indicate what a state means, like I can give a very simple example, you think of a binary counter say a 4-bit binary counter that counts from 0, 1, 2, 3, 4 up to 15 then again back to 0.

Now, suppose when the value of the count is 5, let us say, and we apply a clock, so, the output that we expect is 6, that depends not only on the clock you are applying, but also on the previous count value that 5, that previous count value that can be regarded as the state of the machine. So, the next output that you get, will depend on some previous information; that is the state and on the applied input. So, just as in the example of the counter its stated, the internal states change with time.

And for any practical hardware implementation of such a sequential circuits, the number of states has to be finite. It cannot be infinite clearly because the states ultimately you will be representing or storing in flip-flops. Suppose it is a 4-bit counter, will be storing the 4-bit of information in 4 flip-flops. So, if I say there are infinite number of states, then it is not practical to build such a machine, because you will be requiring an infinite number of flip-flops, fine. So, because of that the practical sequential circuits are also referred to as a finite state machine.

So, there is a concept of a state, number of state is finite and there is a formal definition of a machine that will see, and most of the practical circuits that we see is around us are sequential in nature, very fewer combinational.

(Refer Slide Time: 04:39)

Finite State Machine (FSM)

- A FSM can be represented either in the form of a *state table* or in the form of a *state transition diagram*.
 - Variations exist, e.g. *Algorithmic State Machine (ASM) chart*.
- Example:
 - A circuit to detect 3 or more 1's in a serial bit stream.
 - The bits are applied serially in synchronism with a clock.
 - The output will become 1 whenever it detects 3 or more consecutive 1's in the stream.

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



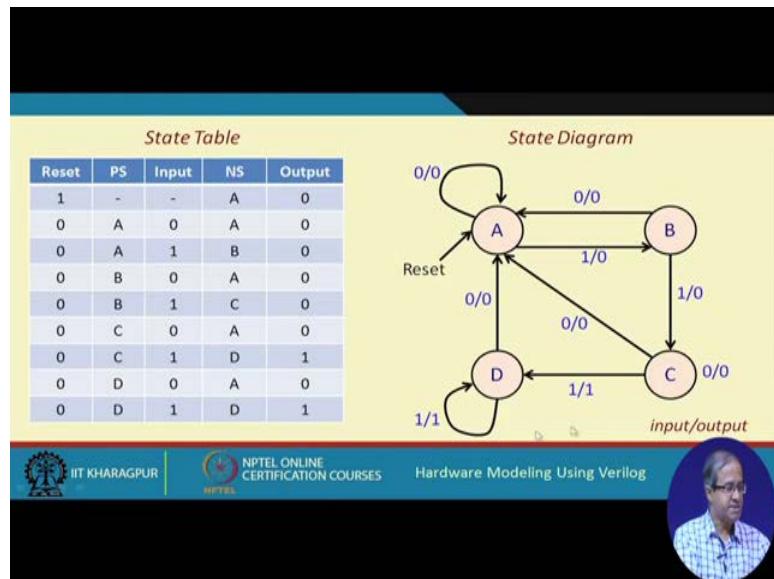
Now talking about a finite state machine; so a finite state machine or FSM in short, can be specified or represented in the form of a state table or in the form of a state transition diagram. Now in the examples that we shall be showing, we shall be using the state transition diagram rotation, but you recall earlier when we talked about the user defined primitives or UDPs, there the sequential functions we represented in the form of table that was an example of a state table. We showed how to create the state table of a d flip-flop, d latch, JK flip-flop, then SR flip-flop and so on.

So, those are examples of state table that same information we can also represent in a diagrammatic form that is easier to visualize sometimes. We shall be showing such diagrammatic representations here; that is called state transition diagram. And of course, there are some variations which many people prefer to use. Say means one such structure that is used is called algorithmic state machine or ASM. So, I mean well just take some examples of ASM later during the course. So, when we talk about designing more complex systems.

So, some examples of FSM, let us say we have a circuit that detects 3 or more ones in a serial bit stream. Means I am assuming that there is one input where a sequence of bits is coming one by one, 1 0 0 1 1 1 0. They are coming continuously in synchronism with the clock let us say. So, here I want to find out whether there are 3 consecutive ones in the bit stream or not. So, every time 3 consecutive ones are detected, the output again a

single bit output should go to 1, otherwise the output should remain at 0. So, the output will become 1 whenever it detects 3, at least 3 or more consecutive ones in the stream. This is what our example here is.

(Refer Slide Time: 06:59)



So, this is the state transition diagram, this is how it depicted of this example, the circuit to detect 3 or more ones. This is the state table description; this is the equivalent state diagram. Let us explain first this state diagram, it is easier to understand, then we shall see the state table. Here the circles they indicate states of the machine. Now how many states will be required for a particular description, it will depend on the problem specification. For this case, because we are trying to detect 3 consecutive ones, we need at least 4 states. What does the 4 states will signify. This A will indicate the initial state; B will mean that we have seen 1, a single 1, so, far; C will mean we have seen 2 consecutive ones; and D will mean we have seen 3 consecutive ones.

So, for this case we need at least 4 states. So, let us see the notation used are, this 0 slash 0 mean, the first one refers to the input, after slash it is the output. So, the circuit has a single input and a single output. So, suppose I am in the initial state. So, when you reset the circuit, it starts with state A, now if I get a 0, so it is not a 1, so I remain in state 0. So, I am waiting for the next 1. So, whenever I get a 1, I go to state B, 1/0. So, still I have not got 3 ones, a single 1, I go to state B. B remembers the information that I have seen a single 1. So, in B if I get another 1, I go to C. So, again the output is 0; c

remembers that I have seen 2 ones, and in C if I get another 1, I go to D with the output being made to 1, because 3 ones have been detected. And while in D if I receive more number of ones, I remain in state D and the output is also made 1. But while in B if the next bit is 0, I have to go back to A, because again I will have to start with the next 1. Similarly, in C if I get a 0, I have to go back to state A; and similarly from D if I get a 0, I have to go back to state A, this 0/0 does not mean anything. So, this is the state transition diagram.

Now this same information can be depicted in the form of a state table. So, what does the state table indicate. It indicates the inputs. So, here my inputs are reset and this input value first bit. I am calling it as input and I have a single output. Now in addition to it there is the notion of the state. I call it present state as PS, next state as NS. So, the idea is that if my reset is high, irrespective of the PS and the input applied, my next state will be A. So, reset it goes to A and the output will be 0. So, if the reset is 0, then my operation starts, if my PS is A and the input is 0, then I go to NS, A with an output 0. This is indicated by this arc.

If my PS is 1, input A and input is 1, then my NS is B with the output 0, it is this edge, input is 1, output is 0, NS is B. So, in this way for every edge in the state transition diagram, I can have one row in this state table. So, these are equivalent representations, but the diagram is more easily, means understandable visually, because this table is a little more complicated to interpret, fine.

(Refer Slide Time: 11:07)

Mealy and Moore FSM Types

- A deterministic FSM can be mathematically defined as a 5-tuple $(\Sigma, \Gamma, S, s_0, \delta, \omega)$ where Σ is the set of input combinations, Γ is the set of output combinations, S is a finite set of states, $s_0 \in S$ is the initial state, δ is the state-transition function, and ω is the output function.
- Here, $\delta : S \times \Sigma \rightarrow S$
 - Present state (PS) and present input determines the next state (NS).
- For Mealy machine, $\omega : S \times \Sigma \rightarrow \Gamma$ (output depends on state + inputs)
- For Moore machine, $\omega : S \rightarrow \Gamma$ (output depends only on the state)

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

Now, let us look at slightly more formal definitions of FSMs. There are two kinds of finite state machines that we distinguish; one is called a Mealy machine other is called a Moore machine. Let us see what these are. Now any finite state machine, well deterministic finite state machine means whenever we apply an input, whenever I know what is the state.

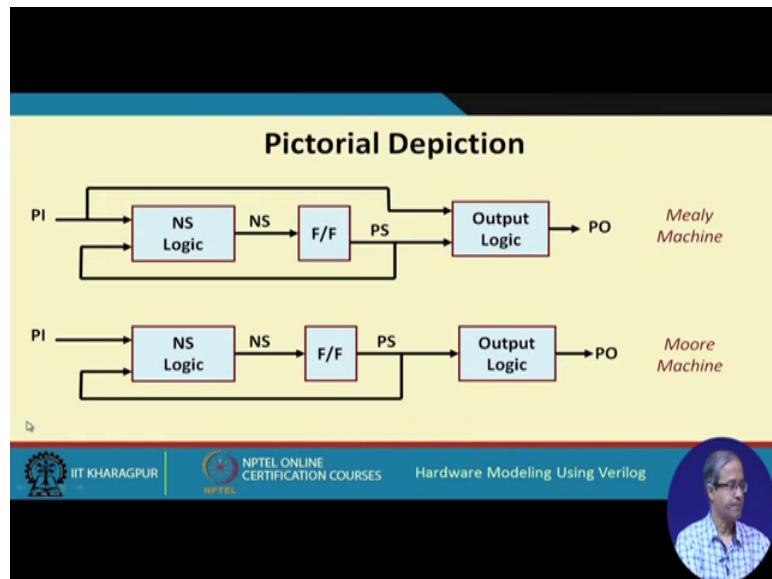
So, the next state and my output is absolutely deterministic. So, all practical circuits are deterministic, but for modeling, theoretically there is another kind of FSM which is also defined which is called non deterministic, but from the practical design point of view, we are only considering deterministic FSMs. So, mathematically it can be represented $\Sigma, \Gamma, S, s_0, \delta$, and ω , where Σ denotes the set of input combinations. Suppose the circuit has two inputs; a and b, there will be 4 input combinations 00, 01, 10 and 11, Σ indicates the set of all the inputs. Similarly, Γ indicates the set of all outputs. So, if there are 3 outputs, the set of all outputs can be 8; 000, 001 up to 111, set of all outputs is denoted by Γ .

S denotes the set of states, and s_0 which is a member of S , is the initial state, so, we start with s_0 . δ is defined as the state transition function and ω is the output function. So, how are they defined? δ is defined as follows, $\delta : S \times \Sigma \rightarrow S$ means this is the definition it means $S \times \Sigma$ means this is a Cartesian product. So, some state, given a state, given a input it will determine the next state. So, given some value from S , given some input from Σ , it will determine some state in S .

So, this will actually tell what the next state will be, present state and present input will determine the next state. And for the output function, here we distinguish between Mealy and Moore machines. For a Mealy machine the output Σ will be determined by this state as well as the present input. So, the output depends on state plus the inputs, but for a Moore machine, the output does not depend on the inputs, it depends only on the present state, output depends only on the state.

So, in this way you can distinguish between Mealy and Moore machines. We shall see some examples.

(Refer Slide Time: 14:09)



Pictorially the Mealy and Moore machines can be shown like this. You see for both Mealy and Moore machines, there are two functions; one is the NS logic other is the Output logic. The NS logic is actually the δ function, it takes a PI (primary input), it takes my PS (present state) which has stored in F/F (flip-flops), and it generates my NS (next state), this is a combinational circuit NS logic, and the Output logic is again similar it takes for Mealy machine, it takes a PI, and the PS, it generates the PO (primary output). So, for a Mealy machine the output is dependent on the PI as well as the state, but for a Moore machine the output is dependent only on this state not on the inputs, this is the difference, fine.

(Refer Slide Time: 15:05)

Example 1

- There are three lamps, RED, GREEN and YELLOW, that should glow cyclically with a fixed time interval (say, 1 second).
- Some observations:
 - The FSM will have three states, corresponding to the glowing state of the lamps.
 - The input set is null; state transition will occur whenever clock signal comes.
 - This is a *Moore Machine*, since the lamp that will glow only depends on the state and not on the inputs (here null).

Let us take an example, this is an example of a Moore machine. So, means our example is very simple. let us assume that there are 3 lamps red, green and yellow. So, there is a circuit, there are 3 output lines, the 3 lamps are connected to the 3 output lines, and there is a clock signal which is coming. There are no separate inputs, there is only a clock, and these 3 lamps are supposed to glow cyclically with a fixed time interval, let us say 1 second interval.

Now clearly here the finite state machine will be having 3 states, because you have to remember that which LED, which lamp had blown in the last state, because it was red, if it was red, then next time the clock comes, it should be green, if it is green next time clock comes, it should be yellow and from yellow it should be red. So, this can be captured by a state transition diagram like this red to green, green to yellow, yellow to red.

So, this is very simple, there are no separate inputs, a very simple finite state machine. This is a Moore machine, because the output depends only on which state it is in, it does not depend on inputs, because there are no inputs in this example, the input set is null, fine.

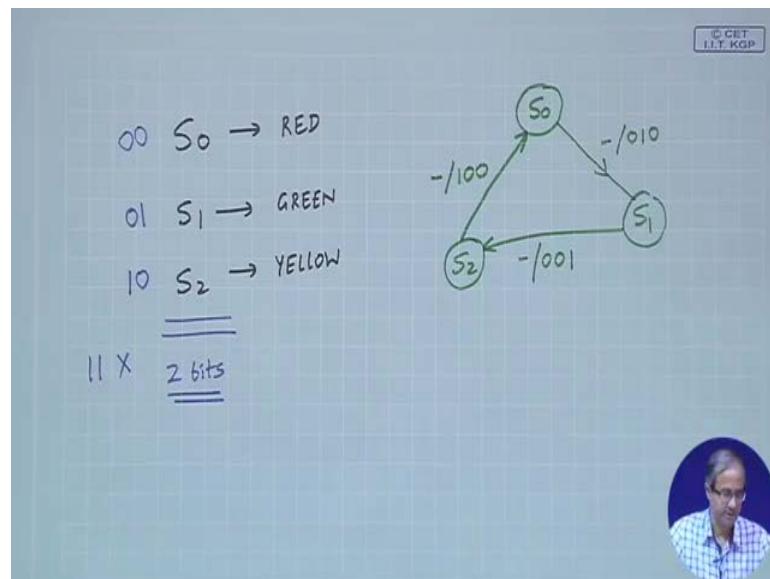
(Refer Slide Time: 16:33)

```
module cyclic_lamp (clock, light);
    input clk;
    output reg [0:2] light;
    parameter S0=0, S1=1, S2=2;
    parameter RED=3'b100, GREEN=3'b010, YELLOW=3'b001;
    reg [0:1] state;
    always @(posedge clock)
        case (state)
            S0: begin          // S0 means RED
                light <= GREEN; state <= S1;
            end
            S1: begin          // S1 means GREEN
                light <= YELLOW; state <= S2;
            end
            S2: begin          // S2 means YELLOW
                light <= RED; state <= S0;
            end
            default: begin
                light <= RED;
                state <= S0;
            end
        endcase
    endmodule
```

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

So, let us see how we can code this example in Verilog, red, green, yellow. So the first thing is that let us make some assumptions. Let us say I define 3 states S0, S1 and S2.

(Refer Slide Time: 16:46)



So I define S0 as red, S1 as green and S2 as yellow. So, talking about the finite state machine, you can have it like this, say this is S0, this is S1, this is S2. So, when you are in S0, if there is a clock, there are no inputs. So, you can write a dash, your, this S1 should glow, S1 means green; red, green, yellow, I means if you put it in that order red

first green then yellow. So, if it is green, so, there should be red, green, yellow; 010 should be the output.

And when in S1, if a clock comes, again there are no inputs, S2 means yellow; red, green, yellow. And in S2, if a clock comes, you go to S1, it is red. So, first one will be 0. So, this will be my state transition diagram, there are no inputs, but there are 3 outputs and there are 3 states. Now the point to know is note that is that here we are talking about hardware implementation. So, there are 3 states, so, we need 2-bits to represent the states. let us say S0, we can indicate by the state 00; this S1, we can indicate by a state 01; and S2 by 10, where is 11 is an invalid state, 11 will be an invalid state, fine. So, let us see how in Verilog, we can code this state transition diagram.

So, we have declared a module, where there is only clock as the input, clock; and light is the output, light is a vector, there are 3 lights red, green and yellow, clock is an input, and we are declaring this state as a 2-bit variable 0, 1, 2-bits. And for convenience, for ease of understanding, we are declaring a parameter constants S0, we are calling it as 0; S1 as 1; S2 as 2. You see in binary 00 means 0; 01 means 1, 10 means 2. So, S0, S1, S2, we are denoting by the decimal number 0, 1, 2. Similarly the 3 colors of the lamps were again defining as a parameter for convenience; red, we are declaring as a 3-bit number 100; green as a 3-bit number 010; yellow as a 3-bit number 001, this will indicate which lamp we are glowing; first one is red, second one is green, last one is yellow.

Now, our description is very simple here we are executing this procedural block triggered by the positive edge of the clock. So, whenever there is a positive edge of the clock, we are seeing in which state we are. Suppose we are in state S0 well and a clock has come. So, my next state will green. So, my state will be changing to S1. This S1 will be indicating the green state you recall, S0 is red, S1 is green, S2 is yellow, and my current output will become green.

Now if I am state S1 which means green and if a clock comes my next will be yellow. So, state will go to S2 and my light will be yellow and if it is in S2 in yellow state will again go back to S0, and it will glow the red light. Now when the circuit starts up, suppose the states, the flip-flops, they start with the 11 state, something else. So, you will have to also give a default state. So, you are not initializing the state. So, if it is

something else other than S0, S1, or S2, these are the 3 values then you start with red as the initial state, state S0, red is assigned to light.

Now, you see here we have used only non blocking assignment statements. So, if this is given to a synthesis tool, the synthesis tool will be generating a circuit, where for this state it will generate two flip-flops, because it has to remember it, and because of the non blocking assignments we have used for light also, it will be using 3 flip-flops, and for every positive edge of the clock these assignments will take place, right.

(Refer Slide Time: 21:41)

```
module test_cyclic_lamp;
    reg clk;
    wire [0:2] light;
    cyclic_lamp LAMP (clk, light);
    always #5 clk = ~clk;
    initial
        begin
            clk = 1'b0;
            #100 $finish;
        end
    initial
        begin
            $dumpfile ("cyclic.vcd");
            $dumpvars (0, test_cyclic_lamp);
            $monitor ($time, "RGY: %b", light);
        end
    endmodule
```

0	RGY: xxx
5	RGY: 100
15	RGY: 010
25	RGY: 001
35	RGY: 100
45	RGY: 010
55	RGY: 001
65	RGY: 100
75	RGY: 010
85	RGY: 001
95	RGY: 100

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

So, I have also shown a sample test bench for this, for this circuit to show how we can write a test bench. You see here we have defined this clk as reg, and the output of this circuit light as wire 3-bit. We have instantiated it, called it LAMP. In this initial block, we have started by initializing clk with 0, at time t = 0, and in this always block after delay of 5, we are writing clk = ~clk, which means I am generating a clock signal with a time period of 10 units. It starts with 0; after 5, it goes to 1; after 5 it goes to 0 again and so on, it repeats.

Well and here I am just applying clocks, and here I am saying, I initialize clock to 0 then this always block will take over and at time 100, I finish, because here I have no separate inputs to apply in this example, the clock is the only input. So, whenever a clock comes the lamp will change from one to the next. And in the just initial thing there is another block where I have specified a dumpfile, where to dump the values, value change dump

and 0, test_cyclic_lamp means, that here all the variables I want to dump, and also I have given a monitor where I am asking that you display the time and RGY equal to b, the value of light.

So, if you just run the simulation, we get an output like this. So, you see that that initially at time $t = 0$, this RGY are not initialized, because clock has not yet come. So, the values are xxx; say at time 5, first positive edge of the clock comes, so it gets initialized to red, because it is the default case, initially it was xxx. So, default it starts with x and with a clock period of 10, 15, 25, 35, 45 like this goes and you see red then green then yellow, again red, green, yellow, in this way it continues, and it will continue till 95, because at hundred you are finishing, so, it stops here, right.

(Refer Slide Time: 24:04)

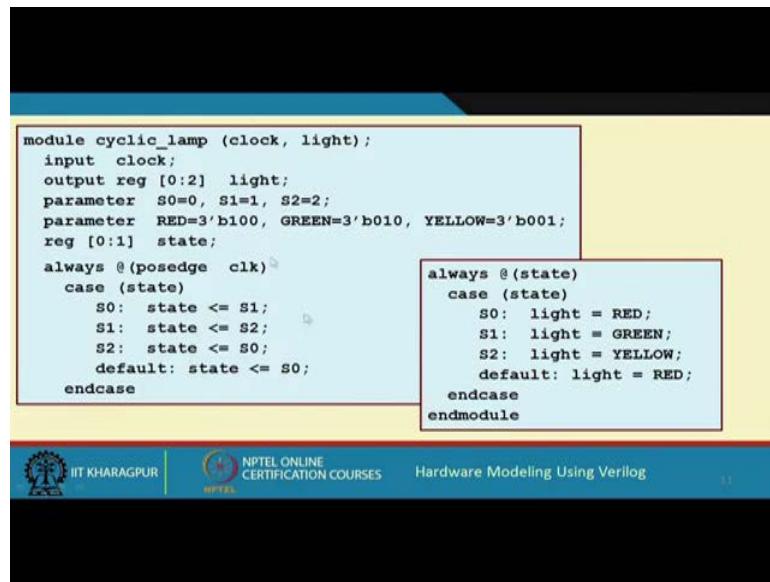
- Some comments on the solution:
 - The synthesis tool will generate five flip-flops – 2 for *state*, and 3 for *light*.
 - The three output lines are also getting stored in flip-flops.
 - We have used non-blocking assignment triggered by clock edge.
 - But actually we do not need separate flip-flops for the outputs, as the outputs can be directly generated from the *state*.
 - How to achieve this?
 - Modify the Verilog code such that all assignments to *light* is made in a separate “*always*” block.
 - Use blocking assignment triggered by state change, and not by clock.

So, as I had said that in this solution 5 flip-flops will be synthesized; two for the state variables and 3 for the light. This is because we have used non blocking assignment triggered by clock for both state and light, this as I said, but you see for this design do you really need this. Do you really need to store the lamp output values in a flip-flop, because if I know this state then I will know that which lamp to glow. So, if I know that I am in state S0, it means red should be activated; if I know I am in state S1, I know green is to be activated. So, I really do not require a flip-flop to store the value of the outputs, storing the state is sufficient.

But because of the way I have specified, the design the synthesis tool is generating flip-flops also for the outputs. So, let us see how we can overcome this. So, we make a modification. So, we split the always block in to two always blocks. First thing is that all assignments to light, will be made in a separate always blocks, where we use blocking assignments and not by clock, because whenever you specify an always block triggered by a clock, the synthesizer will be made to believe that you are trying to do something in synchronism with the clock.

And hence you have to synthesize them as flip-flops or registers, but if you are using a blocking statement then the synthesizer will analyze, that well do you really need to store the values or is it a pure combinational circuit, it will try to decide So the modified Verilog code will look like this.

(Refer Slide Time: 26:07)



```

module cyclic_lamp (clock, light);
    input clock;
    output reg [0:2] light;
    parameter S0=0, S1=1, S2=2;
    parameter RED=3'b100, GREEN=3'b010, YELLOW=3'b001;
    reg [0:1] state;
    always @(posedge clk)
        case (state)
            S0: state <= S1;
            S1: state <= S2;
            S2: state <= S0;
            default: state <= S0;
        endcase
    always @ (state)
        case (state)
            S0: light = RED;
            S1: light = GREEN;
            S2: light = YELLOW;
            default: light = RED;
        endcase
    endmodule

```

The screenshot shows a Verilog code editor with the above code. The code is split into two always blocks. The first always block is triggered by the positive edge of the clock and contains a case statement on the current state (S0, S1, S2). The second always block is triggered by the state itself and contains blocking assignments for the light output based on the state. The code is part of a module named 'cyclic_lamp' with inputs 'clock' and 'light' and parameters for colors.

So, you see, here we have split the always block in to two. So, in the first always block, the first part is the same, no change. So, in the first always block, we are triggering it by the positive edge of the clock, and what is happening here only the state changes nothing else. We are doing a case on state. So, if it is in state S0, my state will become S1; if it is in state S1, it will become S2; if it is in S2, it will become S0 and default it will start with S0. Now in another always block, here I am using, you see blocking assignment statements, and I am triggering it not by clock, but by state. So, whenever state changes

you do this. What you do? If this state is S0, you glow red; If the state is S1 you glow green; if it is S2, then yellow and default since we are starting with S0, it will be red, ok.

So, if you do this, if you make a specification like this, what the synthesizer will see. Synthesizer will find that, well, I see that I have specified the output value for all combinations of state for 00, it will be red; 01, it will be green; 10, it will be yellow and for 11, it will be red. So, it can actually find out that what will be my output just from the state. So, it will be a combinational circuit.

(Refer Slide Time: 27:44)

Comment on the solution:

- The synthesis tool will be generating only 2 flip-flops corresponding to the first clock-triggered “always” block.
- The second “always” block will be generating a combinational circuit that takes *state* as input and produces *light* as outputs.

state (s_1, s_0)	Light (RGY)
S0: 00	1 0 0
S1: 01	0 1 0
S2: 10	0 0 1
11	x x x

Logic expressions after minimization:
 $R = s_0' \cdot s_1'$
 $G = s_0$
 $Y = s_1$

IIT KHARAGPUR
NPTEL ONLINE
CERTIFICATION COURSES
Hardware Modeling Using Verilog

So, just about this solution, this synthesis tool will be generating only the flip-flops for the states, but the second always block, where you are initializing the light, this one. For this no flip-flops or latches will be generated, why, because I can create a truth table where depending on the states, I can uniquely identify the values of the light. So, I know if it is 00, 01 or 10 which light to glow. But if it is 11, it is an invalid state, so, the output will be xxx. So, if you do a Karnaugh map minimization of these 3 functions, this is a two variable function, let us say S1 and S0 are the two state variables.

Then this R, G, and Y outputs can be obtained as $R = S0'S1'$, $G = S0$ and $Y = S1$. So, whenever S1 is 1, then you glow yellow. So, whenever G, this S0 is 1 here, you glow yellow, green and so on. Red will be both are 00, both are 00, red. You see this is a pure combinational circuit description, and synthesis tool will be looking at this, it will find

out that well this is actually a combinational circuit description, and it will be generating a combinational circuit.

So, with this we come to the end of this lecture. So, we had talked about finite state machines, distinguished between Moore and Mealy machines, and worked out one example, that how we can represent the FSM, and how we can code it in to Verilog, and we looked at two alternate kinds of design; one in which unnecessary flip-flops are synthesized, and the other, an improved one, where only the essential flip-flops are generated.

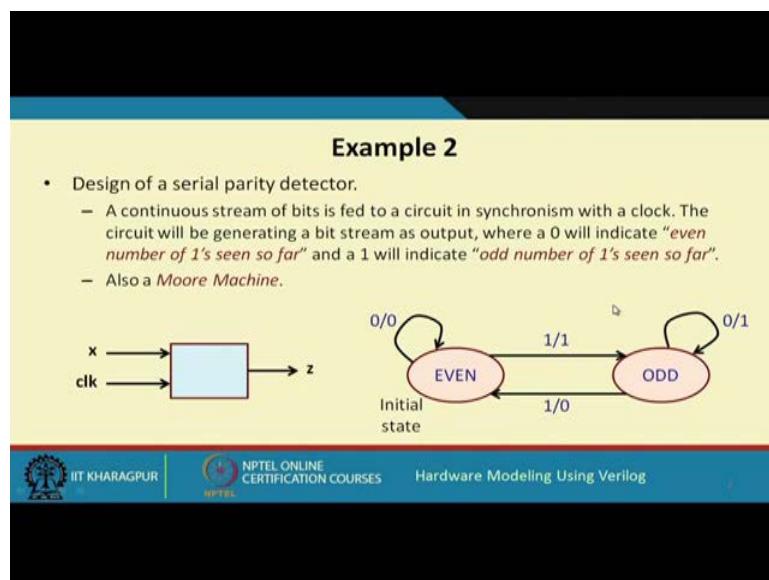
Thank you.

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

Lecture - 24
Modeling Finite State Machines (Contd.)

So, we continue with a discussion on modeling finite state machines. So, if you recall in our last lecture we talked about finite state machines, the different types and how to model finite state machines in Verilog, we considered one example there. So, we continue with our discussion and you look at some more examples of modeling finite state machines. So, our lecture is modeling finite state machines the second part ok.

(Refer Slide Time: 00:50)



So, the example we take now this is also a Moore machine, this is the design of a serial parity detector. Let us first try to understand what is parity. Suppose I have an n-bit word, parity actually indicates whether the number of 1s in that word is odd or even. If it is odd, we say it is odd parity; if it is even, we say it is even parity. Now the parity can be represented by a single bit, let us say 1 can indicate odd; 0 can indicate even or the vice versa, ok.

So, here we are trying to design a circuit where the input number that we are talking about this is not available in parallel, but coming serially bit by bit. So, as it is coming bit by bit, we are continuously computing the parity, and we are continuously outputting a

bit which indicates the parity of this stream that we have seen so far. So, what will be our logic here? We need to store only one piece of information, whether the parity of the bits that we have seen so far is odd or even.

Now if my next bit coming is 0 then I will not change, it will remain, but if my next bit is one then it will change. If it is odd it will become even; if it was even it will become odd, this is our logic. So let us see, how we specify our design. So, here as I said a continuous stream of bits is fed to a circuit, let us say x is that bit stream in synchronism with the clock. This circuit will be generating a bit stream in the output which is here z , and in the output stream a 0 will indicate even parity, that means so far we have seen even number of 1s in x , and output z will indicate that so far we have seen odd number of ones in x , ok.

Now, the output z see here we need to maintain 2 states. Clearly one indicating that whether the number of bits we have seen so far is even and the other number of bits so far is odd. Now you see the output z can directly be generated from this state. So, it really does not depend on the input. So, this is also, this is an example of a Moore machine. So, this state transition diagram we can show like this. This even can be your initial state, initially nothing has come, let us say it is initial. So, if a 0 comes the output is 0 and you remain in the even state. Now if a 1 comes you go to the odd state and in the odd state the output is out to be 1.

Now while in the odd state, if a 0 comes, so, it will remain odd. So, the output will remain 1; but if a 1 comes this odd will become even. So, the output will again go back to 0.

(Refer Slide Time: 04:24)

```
module parity_gen (x, clk, z);
    input x, clk;
    output reg z;
    reg even_odd;      // The machine state
    parameter EVEN=0, ODD=1;
    always @(posedge clk)
        case (even_odd)
            EVEN: begin
                z <= x ? 1 : 0;
                even_odd <= x ? ODD : EVEN;
            end
            ODD: begin
                z <= x ? 0 : 1;
                even_odd <= x ? EVEN : ODD;
            end
            default: even_odd <= EVEN;
        endcase
    endmodule
```

This design will cause the synthesis tool to generate a latch for the output "even_odd".

Modeling Using Verilog

So, my state transition diagram is fairly simple with 2 states. So, here this is the first version of our design, where it is the same kind of a design, but there is an input x. You see in the early example, which we took, there were no inputs separately. There was only 3 states red, green, and yellow, they were cyclic in wherever the clock is coming, but now there is also an input x. Not only the clock, I will have to decide my next course of action depending on what is my next bit x, if it is 0, I will do something; if it is 1, I will do something else. So, so in the first version again I have used a single always statement, you see my x and clock are the inputs and z is an output because it is coming on the left hand side as reg.

And this machine requires a single state variable because there are 2 states, odd and even I need one flip-flop, 1-bit. So, I define a single state variable even_odd that can be either 0 or 1 and I am defining by parameter 0 means EVEN, EVEN; and 1 means ODD, ODD. This is my procedural block. So, whenever the positive edge of the clock comes, I check this state even_odd. If it is EVEN, I do this; if it is ODD, I do this. So, what I do, I update both my output and also my state depending on the value of x.

So, I use a conditional assignment statement here. What does this mean? If x is 1, then assign 1; if x is 0, assign 0. And here it means if x is 1, make this state ODD; if x is 0 make this state EVEN. So, this is exactly as per the state transition diagram. Similarly, if my state is ODD, then I do like this, if x is 1, I assign 0 to the output; if x is 0, I remain in

the ODD state, assign 1; and here if it is 0, EVEN; if it is 1, ODD. And by default because say initially if you do not initialize the state variable, it can the outputs will be all indeterminate. So, here we have included a default statement. So, initially this even_odd will start with EVEN, if nothing matches. So, here again because I am using a clock triggered always block, and you are using non-blocking assignment, there will be one flip-flop generated for z and one flip-flop for the state, even_odd, this even_odd is the state.

(Refer Slide Time: 07:27)

```
module parity_gen (x, clk, z);
    input x, clk;
    output reg z;
    reg even_odd;      // The machine state
    parameter EVEN=0, ODD=1;
    always @ (posedge clk)
        case (even_odd)
            EVEN: begin
                z <= x ? 1 : 0;
                even_odd <= x ? ODD : EVEN;
            end
            ODD: begin
                z <= x ? 0 : 1;
                even_odd <= x ? EVEN : ODD;
            end
            default: even_odd <= EVEN;
        endcase
    endmodule
```

This design will cause the synthesis tool to generate a latch for the output "z".

Modeling Using Verilog

So, this design will cause the synthesis tool to generate a latch for the output z, but again you try to understand the design, just by knowing the state, I can tell what the output is. I really do not need a latch to store the output, if it is EVEN, my output should be 0; if it is ODD, my output should be 1, simple fine.

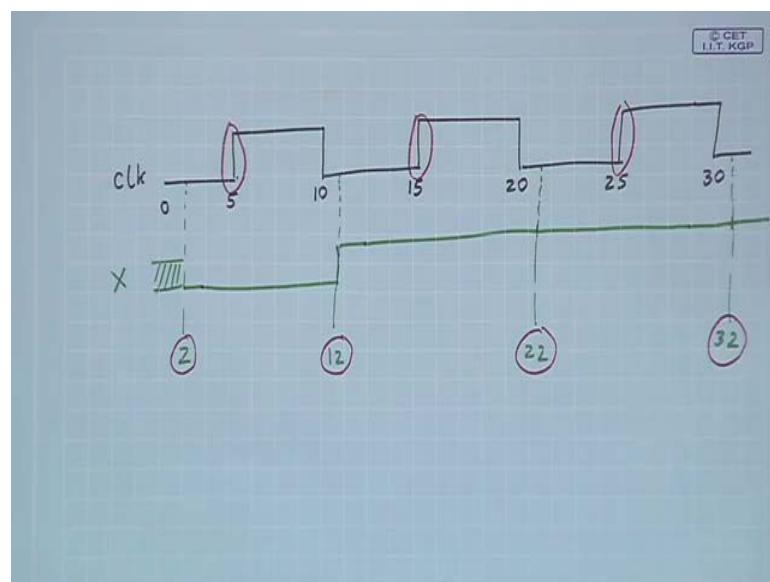
(Refer Slide Time: 07:56)

The screenshot shows a Verilog test bench code titled "test_parity". The code includes an instantiation of a "parity_gen PAR" module with inputs x, clk, and output z. It features an initial block that sets up a dumpfile named "parity.vcd" to capture all variables, initializes the clk to 0, and specifies a clock period of 5 units. Following this, an always block generates a repeating sequence of x values (0, 1, 1, 1, 0, 1, 1, 1, 0) over time. An initial block also contains a \$finish command. The code concludes with an endmodule statement. The background of the interface is blue, and there are logos for IIT Kharagpur, NPTEL, and the course title "Hardware Modeling Using Verilog".

```
module test_parity;
    reg clk, x;    wire z;
    parity_gen PAR (x, clk, z);
    initial
        begin
            $dumpfile ("parity.vcd");  $dumpvars (0, test_parity);
            clk = 1'b0;
        end
    always #5 clk = ~clk;
    initial
        begin
            #2 x = 0; #10 x = 1; #10 x = 1; #10 x = 1;
            #10 x = 0; #10 x = 1; #10 x = 1; #10 x = 0;
            #10 x = 0; #10 x = 1; #10 x = 1; #10 x = 0;
            #10 $finish;
        end
endmodule
```

So, this will be the corresponding test bench for this I am showing one example. So, we have instantiated this parity generator x, clk and z, these are the inputs, you see x, clk, z, same set of inputs. And in this initial block, I am doing 2 things, I am specifying the dumpfile, I am specifying the variables to dump, all variables, and initializing the clk to 0. Because I have not given a delay, this will be done at time t = 0, and again this is the clock generation with a delay of 5, I am complementing the clock.

(Refer Slide Time: 08:43)



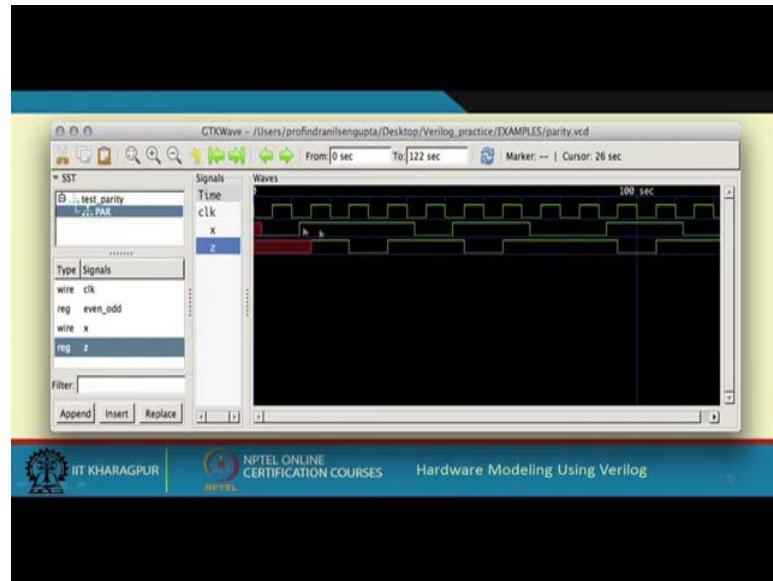
Now, here I am applying the bits. So, you see again now what has happened, the clock will be coming like this. So, we have initialized clock at 0, at time 5 we are complementing. This will be time 10, this will be 15, 20. So, after a gap of 5, we are complimenting the clock, right. 5 again at 30, it will go back to 0 and so on. Now you see what I am doing to x. You see we are giving a delay of 2, such that when the clock edge comes by that time this input should be stable. So, how we are applying x? We are applying x at a time equal to 2.

So, we are applying 0. So, initially x can be anything, I do not know it will be something invalid, but at time 2, I am making it 0. Now again I am applying the consecutive bits with delays of 10, say 0, 1, 1, 1 and so on, I applied many bits. But initial bit, I am applying at 2, then successive bits and applying after delays of 10, 10, 10 like that. So, the first bit 0 will be applied at 2; the next bit 1 will be applied at a delay of 2; that means, at 12; so here if this time is 12.

So, at 12, it will be 1; next bit is 1, the next bit is again 1. So, at time 22, if this time is 22 here I am applying again a 1. So, this 1 remains 1; then again at time 32, I am, because here also I am applying 1, so, this will remain 1. So, like this I am applying the consecutive inputs at times 2, 12, 22, 32 and so on. So, what I am trying to achieve? I am trying to achieve is that whenever the active edge of the clock comes, the rising edge, well before that the input should be stable, because if I change the input right when the clock edge is coming there will be a risk condition.

So, the output may not change properly. That is why we have given a small delay, this delay can be 2 or 3 or 4 no problem, it should be before 5. So, like this we have done it. Before the clock edge comes, the input should be changed and be stable, fine.

(Refer Slide Time: 11:18)



So, here you see the timing diagram of the simulation output. This is the clock signal which is coming, and this is my x , you see. Initially it was undefined at time 2, I am making it 0. You see what I have said here you look at here. So, initially it was undefined at time 2, it was made 0. So, for up to time 12, it will remain 0, then it will go to 1. So, exactly the same thing happening, you see it at time 12, it is going to 1; at time 22 it remains 1; at time 32 it remains 1; but at time 42, it goes to 0, because next bit is 0, you see next bit I have applied 0, 1, 1, 1 next bit is 0.

So, like this the bits are coming. And the outputs, the outputs that will be generated it will be coming only after the clock, the first clock is coming here. The first clock is coming here, so, the output is getting initialized here. And the output will be generated accordingly, then this 1 means ODD, then EVEN, ODD, ODD, EVEN, ODD, ODD, EVEN like that it goes on changing, right. So, this is just an example.

(Refer Slide Time: 12:43)

```
module parity_gen (x, clk, z);
    input x, clk;    output reg z;
    reg even_odd;    // The machine state
    parameter EVEN=0, ODD=1;
    always @(posedge clk)
        case (even_odd)
            EVEN: even_odd <= x ? ODD : EVEN;
            ODD:  even_odd <= x ? EVEN : ODD;
            default: even_odd <= EVEN;
        endcase
    always @ (even_odd)
        case (even_odd)
            EVEN: z = 0;
            ODD:  z = 1;
        endcase
    endmodule
```

This design will not cause the synthesis tool to generate a latch for the output "z".

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

So now we can improve this design to avoid the flip-flop on the output. So, how you do just like the example we considered in the last lecture, we split the always block in to two. So, in the first always block we are only updating the state with respect to the clock edge. So, whenever there is a positive edge of the clock, we check whether this state is even or odd, if it is even depending on the input x, if x is 1, we change the state to odd; if x is 0, will leave it as even. But if this state is odd, depending on x, if x is 1, you make it even; if x is 0, you make it odd. And default is even and in the other always block, you trigger it whenever the state changes, you do a case. So, if the state is even, the output is set to 0; if it is odd, the output will be set to 1, just that, ok.

So, here z can be directly generated from this state value. In fact, you do not need any circuit at all. This state directly will give z, you can just connect a wire, connecting even_odd to z, z and even_odd will be the same value, ok. So, no latch is required here, fine.

(Refer Slide Time: 14:16)

Example 3

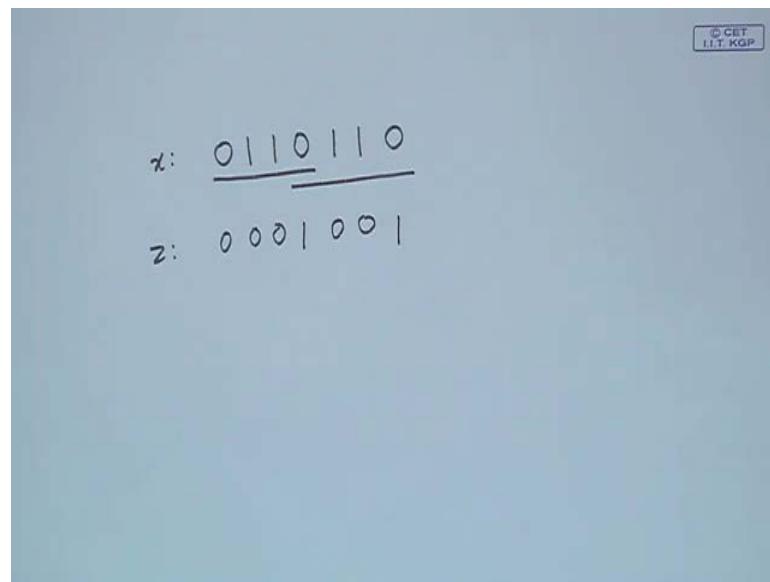
- Design of a sequence detector.
 - A circuit accepts a serial bit stream "x" as input and produces a serial bit stream "z" as output.
 - Whenever the bit pattern "0110" appears in the input stream, it outputs $z = 1$; at all other times, $z = 0$.
 - Overlapping occurrences of the pattern are also detected.
 - This is a *Mealy Machine*.

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

Now let us take the example of a Mealy machine. Now in a Mealy machine you recall the output value will depend not only on the state of the machine, but also on the present input, right. So, here let us see. So, the example that we are taking is that of a sequence detector. So, what is the sequence detector? Sequence detector is the circuit that again there is a serial bit stream x coming, let us say as input and again it will be generating a serial bit stream as output, let us call it z .

So, our design specification says that whenever we encounter the bit stream 0110 in the input stream, this output z will immediately go to 1, and at all other times it will remain as 0. And overlapping occurrences will also be detected.

(Refer Slide Time: 15:22)



For example, If I have 0110 then again 110, then this will be considered as 0110 sequence. And this will be considered as another overlapping 0110 sequence. So, if this is your x, your output z will be here it will become 1, and here also again it will become 1, at all other times it will remain 0, right. This is an example of a Mealy machine because the output also depends on the input. So, this is an example, bigger example, you see this is my x.

(Refer Slide Time: 16:08)

Example 3

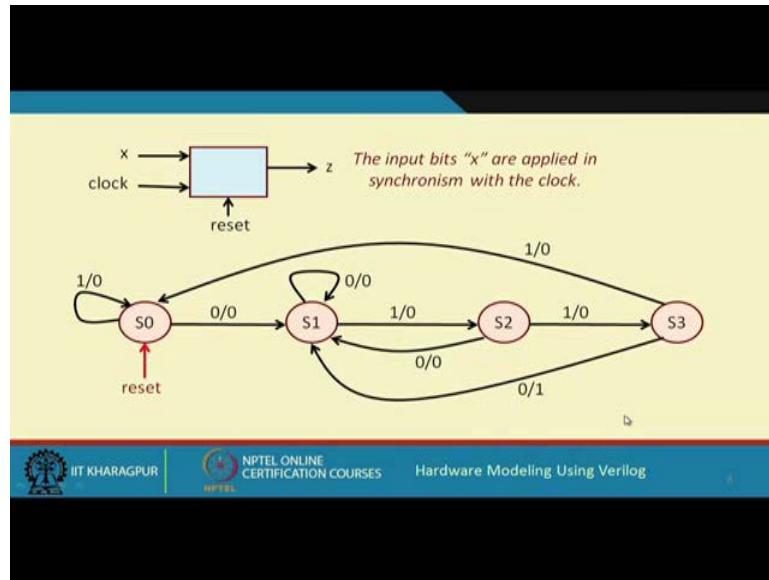
- Design of a sequence detector.
 - A circuit accepts a serial bit stream "x" as input and produces a serial bit stream "z" as output.
 - Whenever the bit pattern "0110" appears in the input stream, it outputs $z = 1$; at all other times, $z = 0$.
 - Overlapping occurrences of the pattern are also detected.
 - This is a *Mealy Machine*.

Example: $x := 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0$
 $z := 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0$

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

And here I get my first 0110. So, this is my first 1 generated, again overlapping 0110, again 1 generated and again you have 1 here 0110, again a 1 generated; this will be your output stream, right.

(Refer Slide Time: 16:25)



Now, what will be my state transition diagram, let us try to understand. You see intuitively here I want to detect 0110. So, you may say that well I start with some initial state 0, I go to one state 1, I go to another state 1, again 0, there should be 5 states. But when you start drawing the state transition diagram, you can find out whether you really need 5 states or we can do it less in this example. You can specify the description using 4 states only. How it is like this? So, I am assuming my circuit is as follows where there is one input x , there is of course a reset input, reset I am showing separately, clock and the output is z . The input bits they are applied in synchronism with the clock. Now here whenever the reset signal is active, I start in state S_0 .

Let us see. Here we are trying to detect this string 0110. And whenever it is that output should be 1. So, let us identify the states, S_0 means the initial state. So, whenever I get the first 0, I go to S_1 , what does S_1 mean? That I have seen the first 0 of the string. Then if I get a 1, I go to S_2 , what does S_2 mean that I have got 0 followed by a 1. Then I go to S_3 , this S_3 means, S_3 whenever 1, so, 011, S_3 means I have seen 011.

Now if I get another 0 what I could do, I could have brought it back to S_0 , you start with another string, but now because I am allowing overlapping of the strings 0110, I am

moving it back to S1, not S0, Why S1, because this last 0 can also be the first 0 in the string. So, as I had given that overlapping example earlier, the last 0 of 0110, that 0 can also indicate the beginning of the next 0110. So, whenever a 0 comes which means I have encountered a 0110, the output goes to 1, and my next state goes to S1 because I am looking for the overlapping string; if there is another 110, so, again a 1 will be generated, right.

But now let us look at the other edges. So, in S0 if I see that a 1 is coming which means where it is not starting with 0, I remain in S0, I wait for the first 1. Now while in S1, I am expecting a 1 to come, but if it is 0, so, I remain in state S1, that means, a 0 is still there may be a 1 is coming. Now in state S2, I have received 01, I am expecting a 1, but if a 0 comes, I go back to S1 because I have seen a 0, maybe I am again starting to look at this string. Similarly, at S3, I have seen 011, but again a 1 comes. So, I have to start from the very beginning, I have to again look for a 0 and like this. So, this will be my state transition diagram for this example, right.

(Refer Slide Time: 20:05)

```

// Sequence detector for pattern "0110"
module seq_detector (x, clk, reset, z);
    input x, clk, reset;
    output reg z;
    parameter S0=0, S1=1, S2=2, S3=3;
    reg [0:1] PS, NS;
    always @(posedge clk or posedge reset)
        if (reset) PS <= S0;
        else PS <= NS;
    always @(PS,x)
        case (PS)
            S0: begin
                z = x ? 0 : 0;
                NS = x ? S0 : S1;
            end
            S1: begin
                z = x ? 0 : 0;
                NS = x ? S2 : S1;
            end
            S2: begin
                z = x ? 0 : 0;
                NS = x ? S3 : S1;
            end
            S3: begin
                z = x ? 0 : 1;
                NS = x ? S0 : S1;
            end
        endcase
    endmodule

```

 IIT KHARAGPUR |
 NPTEL ONLINE CERTIFICATION COURSES |

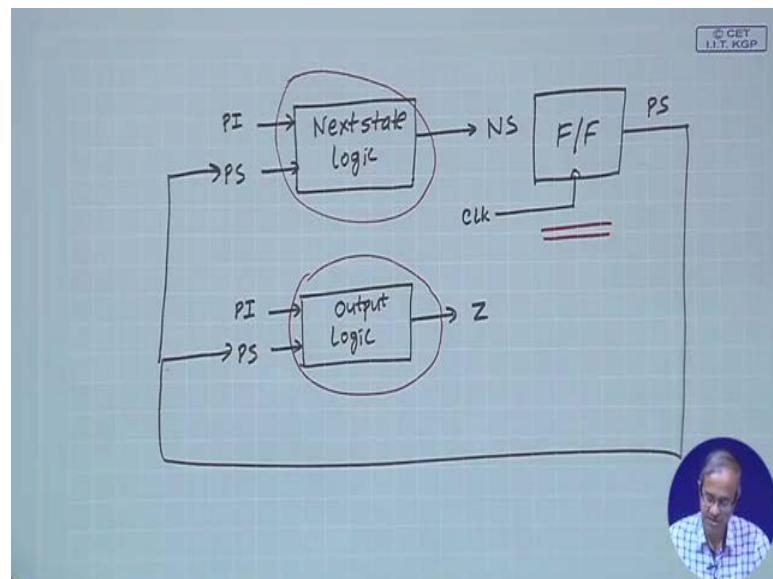



So, this is my Verilog description. So, you see here we have already separated out two always blocks. So, in the one always block, we have just only talked about the state changes, you see the description how we have done it. x, clk and reset are the inputs, z is the output reg. And because there are 4 states, I need 2 state variables reg 0 to 1. So, I call or I define 2 variables, one is the present state and the next state. And using

parameter for convenience, I call them the four state S0, S1, S2 and S3. In the first always block which is triggered by the positive edge of the clock or the positive edge of the reset. What you do, if the reset is active, I initialize the PS to S0, or if the reset is not active and a clock has come.

So, whatever is my NS that will now become my PS; so as the machine progresses clocks are coming. So, I have some inputs, I have generated the NS, when the next clock comes that NS will become my PS; again some input will be coming, I generate the NS. Next time that NS state will again become my PS and this will continue. So, here with the clock, my NS is becoming my PS. And in this always block, we are actually generating the NS and z.

(Refer Slide Time: 21:54)



You see, you just recall for the that model of the sequential circuit we have given. We talked about something called Next state logic. And we talked about another block called output logic. So, what does the Next state logic contain, it will contain the PI and my PS, it will generate the NS. And the output logic, it will also take as input the PI and the PS, it will generate the output, let us say z. Now both of these are combinational circuits.

Now this NS, this is being fed to some flip-flops. This flip-flop is fed with a clock, and the output of the flip-flop, I am calling as the PS. So, whatever is my PS, this is being fed here as well as here, right. So, this is how it works. So, the description of this flip-flop, I have kept in one always block, for basically NS is going to PS. And description of these

two combinational blocks I have kept separate, both of these will be combination. So, that I am doing in this second always block. So, I am just activating it or triggering it whenever PS or x changes, if the state is S0 you see what happens in S0, if the input is 0, I go to state S1; if the input is 1, I remain in state S0.

So, if the, let the next state, if the input is 1, I remain in state S0; if the input is 0, I go to state S1. And the output is 0 under both conditions. So, here you can either write z equal to if x then 0 else 0 or you can simply write z equal to 0, because your z will be equal to 0 irrespective of x. So, this also and this also, you can simply write z equal to 0. So, S1 if x is 1, I go to S2; if x is 0, I remain in S1. Like that just exactly following the state transition diagram, I code these, if then else statements, right, and my description is complete.

(Refer Slide Time: 24:29)

```

module test_sequence;
    reg clk, x, reset;    wire z;
    seq_detector SEQ (x, clk, reset, z);
initial
begin
    $dumpfile ("sequence.vcd"); $dumpvars (0, test_sequence);
    clk = 1'b0; reset = 1'b1;
    #15 reset = 1'b0;
end
always #5 clk = ~clk;
initial
begin
    #12 x = 0; #10 x = 0; #10 x = 1; #10 x = 1;
    #10 x = 0; #10 x = 1; #10 x = 1; #10 x = 0;
    #10 x = 0; #10 x = 1; #10 x = 1; #10 x = 0;
    #10 $finish;
end
endmodule

```

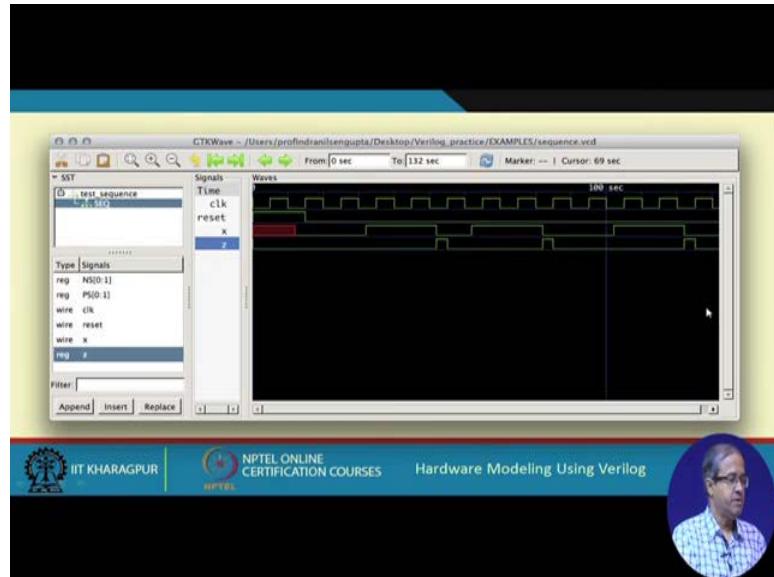
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog | 10

So, similarly I can create a test bench. So, I instantiate my sequence detector x, clk, reset and z. x, clk and reset are the inputs, they are declared as reg, z will be wire. So, again I am creating the dumpfiles. I am initializing the clk to 0 and the reset to 1. So, I am initially resetting and at time 15, I am changing the reset to 0. So, from time 15 onwards my actual operation will start, and clk again with time 5, I am toggling.

So, again I am giving a little delay 12 because the first operation should start at time 15, right. Because till 15 reset is active, after 15 actually operation start. So, just before that at time 12, I am setting or I am sending the first bit. Then I am sending the consecutive

bits after gaps of 10, 10, 10 because clock period is 10. So, 001101100110 like this we are applying, right.

(Refer Slide Time: 25:49)



So, and there are delay, we finish. So, the simulation output comes like this. So, this is I am not showing the whole of it, a part of it, the clock is coming, the reset you see, the reset was activated initially to 1, and it remained till 15, it becomes 0. So, reset was 1, this is time 15, at time 15 it becomes 0 and after that it remains 0. So, my circuit starts operating from that point onward.

And what about x , I have set $x = 0$ at time $t = 12$. So, see at time $t = 12$, I have, before that it was undefined, this red block, red block means undefined. It becomes 0 here, then the successive inputs I am applying 0011, I am applying like that 0011, then 0110., so, 0110 like that. So wherever I get a 0, you see the first 0110 comes as 0110, the first 1 comes here, then there is an overlapping 01,0110, ok.

Then at the end there is again a 0110. So, there will be 3 detections, z will become 1 here 0011, first detection; then overlapping 0011, second time; then again a 0011, third time. So, there will 3 outputs that equal to 1. So, we have seen how we can actually model both Mealy and Moore machines, I means using Verilog. You can take other examples also similarly. So, the process of coding them in Verilog is exactly similar. So, you can take some other example, like for example, talking about the sequence detector again.

(Refer Slide Time: 27:50)

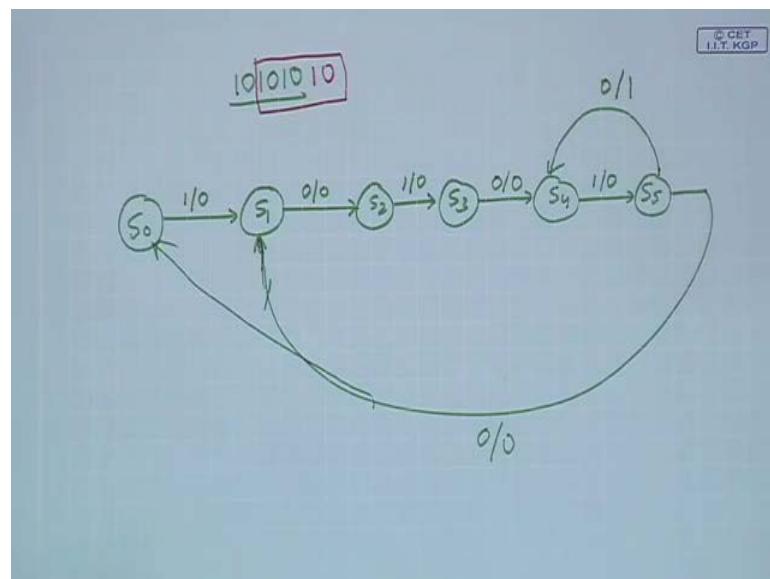
Example 4

- Design a sequence detector for the bit pattern "101010".
 - Work out the state diagram in a similar way.
 - Then code the state diagram in Verilog.

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

Suppose I want to detect this sequence 101010. So, how will my state diagram look like?

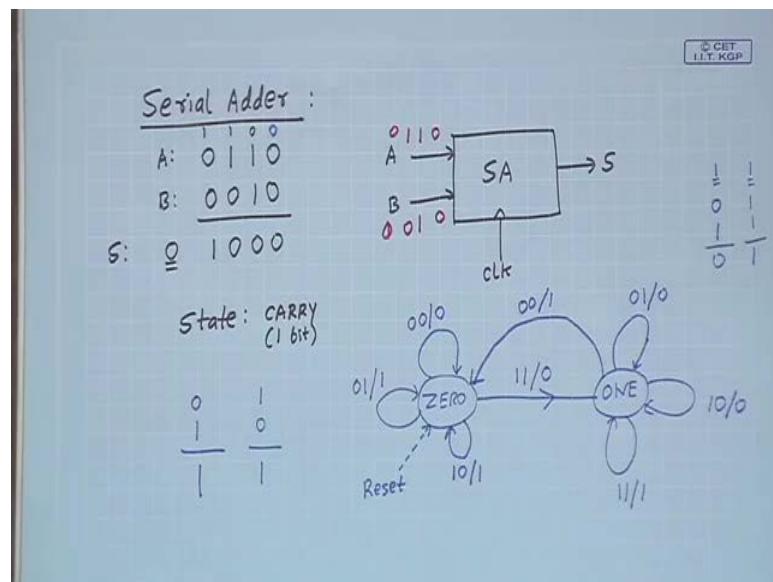
(Refer Slide Time: 28:05)



So, it will look something like this, I mean state S0. So, I am trying to detect the sequence 101010. So, when the first 1 comes, I go to state S1. This is my first 1 comes, then a 0 comes, I go to S2. Then again a 1 comes, I go to S3. Then again a 1 comes, I go to S4, 01010,0 comes. Then the fifth 1, I go to S5. Then if a 0 comes, where do I go now, again if I look at overlapping patterns, see 101010 if there is again a 1 and a 0, I can consider this as overlapping pattern, right. So, if a 1 comes after this.

So, I should actually send it to a state here, where I am expecting a 0 again. So, if it is 1, I have to send it to S1, not really S1, I can actually send it to, I need a 1 and a 0 after that you can send it to S4. If there is a 101010 with an output of 1. So, if there is another 1, I go to S5; it is another 0, I again go to here, but if my next bit is, let us say instead of 1, if it is 0, then not here, I have to send it back to this S0 again, if it is 0. So, the other edges you can define like that, ok.

(Refer Slide Time: 30:08)



Let us also talk about another example, suppose we want to design a serial adder. Well, you know about binary addition, suppose I want to add these 2 numbers, I do a bit by bit addition $0 + 0$ is 0 with a carry of 0; $0 \ 1 \ 1$ is 0 with a carry of 1; $1 \ 1 \ 0$ is 0 with a carry of 1; this is 1 with a final carry out of 0. We do it like this normally here we have seen various kinds of adders like ripple carry adder or the carry look ahead adder, both these designs we have seen earlier. But now let us assume that we want to do this addition bit by bit serially, how? Suppose we have an adder, this is my serial adder. So, my 2 inputs are let us say A and B. So, there is a single bit of A and single bit of B, I am applying at one time. And this is my sum and I am generating one bit of sum at a time. So, if these are my inputs let us say 0 and 0, first I will apply 0 here, I will apply 0 here. Then I will apply 1, then I will apply 1, I will apply 1, then I apply 1, I apply 0, then I apply 0, I apply 0.

So, this I will be doing in synchronism with a clock, there will be a clk signal. And you see internally if you know talk about this state, what do we have to remember? Here for every stage of addition, we only need to remember the previous carry. So, the only state we need to remember is a carry, and that carry will be 1-bit. So now, you can construct a state transition diagram directly by considering the state of the carry, let us say the carry was 0, the carry was 1. So, if you are resetting the circuit, it will start with carry of 0. So, initially there is no carry in, right. So, you start with 0 carry.

Now, let us say. So, if your input is 0 0 what will happen, if the input is 0 0 there will be no carry. So, it will remain in the 0 state and the output will be 0, because if you add 0 and 0 output is 0, but carry is still 0. Now if we apply 0 and 1 what will happen? Say if you are apply 0 and 1, the sum will be 1, but no carry. So, carry will still be 0. So, there will be another edge, if it is 0 1, but now sum will be 1. Similar will be the case if the inputs are 1 and 0 again sum will be 1, but no carry; so 1 and 0, 1; but if it is 1 1 like this then sum will be 0 and carry will be 1.

So now, the carry state changes, if it is 1 1, the sum will be 0, but carry has become 1. But once carry has become 1, let us say my inputs are say 0 1 or 1 0 or 1 1 then there will be a carry, you see 0 1 means what? I have applied 0 and 1, but there is also a carry of 1. So, these 3-bits are added, sum will be 0 and again carry will be 1. So, it will remain in the 1 state, sum will be 0. Similar is for one same thing. But for 1 1, if there is also a carry 1, so, I have a sum of 1 and also carry of 1, so, the output will also be 1. Now the only way to go back to 0 is when the input 0 0 is applied. 0 0 with a carry of 1, this sum will be 1, but next carry will be 0. So, this will be my state transition diagram for a serial adder.

So, in this way you can actually create the state transition diagram of any FSM you can think of, and once you have created this state transition diagram of the state table. Translating it to a Verilog code in the way we have seen is absolutely straightforward.

So, with this we come to the end of this lecture.

Thank you.

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

Lecture – 25
Datapath And Controller Design (Part 1)

So, in the earlier lectures we have seen how we can design both combinational and sequential circuits. Specifically, we talked about finite state machines both Moore and Mealy type machines. So, we looked at the various ways in which we can model those sequential machines in Verilog, but you think of a complex system. So, whenever we are trying to build a complex system, the complex system will consist of a mix of everything. There will be some combinational parts; there will also be some sequential parts. So, in this lecture we shall try to talk about such systems which are called data path and controller design.

So, in such a system we shall see that there are two parts called data path and controller part, and how they are related and how they can be designed ok.

(Refer Slide Time: 01:29)

The screenshot shows a presentation slide with a black header and footer. The main content area has a yellow background. The title 'Introduction' is centered at the top of the yellow section. Below the title is a bulleted list describing the partitioning of a complex digital system into data path and control path. At the bottom of the slide, there is a footer bar containing the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the course name 'Hardware Modeling Using Verilog'.

Introduction

- In a complex digital system, the hardware is typically partitioned into two parts:
 - a) *Data Path*, which consists of the functional units where all computations are carried out.
 - Typically consists of registers, multiplexers, bus, adders, multipliers, counters, and other functional blocks.
 - b) *Control Path*, which implements a finite-state machine and provides control signals to the data path in proper sequence.
 - In response to the control signals, various operations are carried out by the data path.
 - Also takes inputs from the data path regarding various status information.

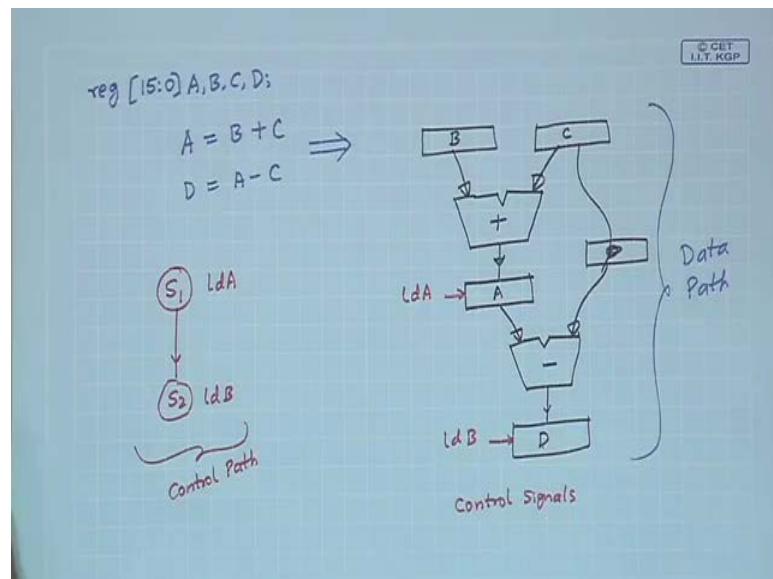
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

So, let us see the basic idea. So, as I have said that for complex digital systems we normally do not design the whole hardware in one piece. They are typically partitioned into two parts; the first part is called the data path. What is the data path? It consists of the functional units, where all the computations are carried out. So, what are there in the

data path. They will consist of typically some registers to store some data, multiplexers, BUS, the adders, subtractors, multipliers, counters and similar functional blocks. So, in a data path, there are a lot of hardware and things which are there, but we are not specifying or telling exactly what to do with those hardware. Let us say I can tell that there are three registers, there is one adder, one subtractor and one counter, but I am not specifying within the data path that exactly how I am going to use them. So for that there will be a second part. There will be a control path, control path is nothing but a finite state machine.

This will be generating or providing some control signals for the data path in a particular sequence. And by doing that the data path will be activating or it will get activated accordingly, and the operations will be carried out as per the intended requirement. The control path can also take some input from the data path to get various status information. Let us take a very simple example.

(Refer Slide Time: 03:23)



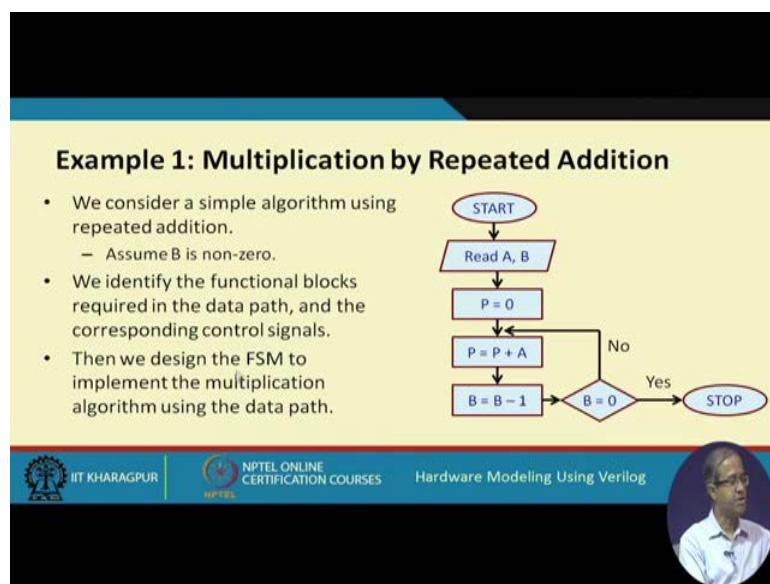
Suppose we have 2 statements, $A = B + C$, followed by $D = A - C$. Now A, B, C, D , let us say these are all registers. Let us say we have defined registers. Let us say these are all 16-bit registers A, B, C and D . Now when we are actually trying to build a hardware for this. First thing is that intuitively speaking you look at this $A = B + C, D = A - C$, what should be my hardware look like. My hardware intuitively you can say that well there will be a register to hold the value B , there will be another register to hold the value of C ,

there will be an adder. This adder will take the inputs from B and C, and it will generate the result in to another register A, this is A, and then will be having a subtractor. It will take A as the first input.

There will be another register D, it will come from here, no sorry, this will be is from C, from C. And the output will be generated in D, right. So, whatever I have shown here this will be my data path. This is what is meant by data path. Some hardware blocks and the way they are interconnected. Now you see in some of these blocks, there will be some controls, like in this A there can be a load control, let us say load A, in D there can be a load B control, right. So, let us assume that this is a fixed circuit. So, these load A and load B these are called control signals.

Now, this is a very simple example. So, in terms of the FSM, I will say that well I have 2 states, S1 followed by S2. In state S1, I am simply activating load A, why? You see if I activate load A, then B and C, whatever is getting added that will get stored in A. In S2, I am activating load B, which means whatever is in A and whatever is in C, they will be subtracted and it will be loaded in D. So, this is my data path and this will be my control path. Control path of course, will be some implementation of this, they were implement. So, this is what I mean by data path and control path.

(Refer Slide Time: 06:38)



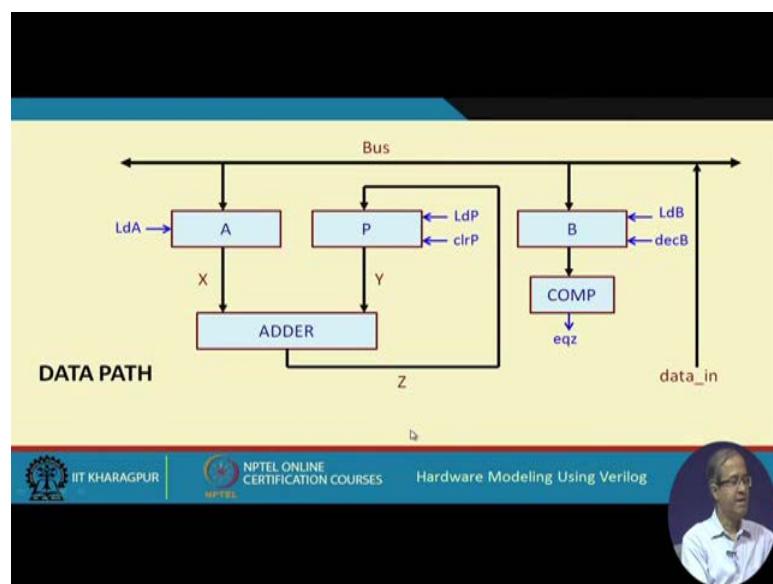
Let us see through some examples. The first example we take in this regard is a very simple example, where we are trying to multiply 2 integers by repeated addition. So, the

algorithm is like this we are let us say reading the 2 numbers A and B, the product which initializes in to 0. And we are repeatedly, we are adding A to the product in this loop, and decrementing B every time, till the time B reaches 0. So, if B is 5, this loop will be repeated 5 times and 5 times will be adding A to 0. So, it will be 5 in to A. So, the product is done. So, here of course, we have made an assumption in this flow chart that we assume B is nonzero, because if B starts with a 0, this algorithm will not work, because we are first decrementing and then checking, ok.

So, for this example let us try to illustrate how we can design the data path and control path. So, in the first step we shall be identifying the data path, and also now in the data path what are the control signals that need to be activated. Then we shall be designing the finite state machine, which will correspond to the control path, right. So, for this example let us try to see, what are the requirements for the data path. Like one thing I can immediately see that I need 3 registers A, B and P.

I need an adder, the B register should also be a down counter because I have to decrement it by 1. There has to be some kind of a comparator which checks for 0, right. These are the few things I need. So, from this requirement we can directly come up with a data path.

(Refer Slide Time: 08:41)



So, this is a data path which we have arrived at from this flow chart specification, let us see this one by one. First we have to read A, B from outside. So, the values of A and B, I

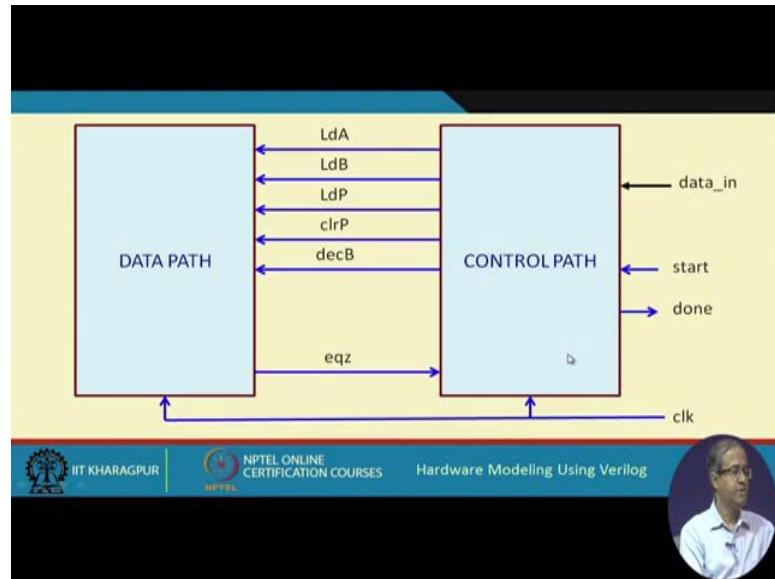
am assuming these are supplied from outside through an input called `data_in`. This is my A register; this is my B register. So, from this `data_in`, the data can go to either to input of A or to the input of B. There are 2 control signals `LdA` and `LdB`, which can be used to load this data either in A or to B. So, let us say first we apply the value for A, activate `LdA`, gets loaded; then apply the second data, then activate `LdB`, it gets loaded in B, right. Now let us see the other requirements. So, we are initializing P to 0, right.

So, P is another register which has a special control called `clear, clrP`. So, if this control signal is activated P will be initialized to 0, right. And another thing, in every step of the iteration, I am, we are doing $P = P + A$, P and A are added result is going back to P. So, you see P and A we are adding them, output of the adder is going back to P and going back to P means, when the value will be loaded. It will be loaded only when this `LdP` signal is activated. So, there will be another control signal for loading this P, and regarding this B register, so B has to be decremented.

So, B is actually a counter which has another control input `decB`. So, if it is activated it will be decremented by 1, and we have to check B for 0. So, there will be a comparator circuit, which will be checking whether B is 0 or not. So, how you can check something for 0? So, it will be nothing but a NOR gate, multi input NOR gate where all the inputs of the NOR gate you connect to the different bits of B. So, the output 1 means the number is 0.

So, this is a signal which is generated by the data path, and this will be required as input by the control path. So, you see this is my data path. So, from outside it is taking the `data_in` and the control signals are `LdA, LdB, LdP, clrP` and `decB`.

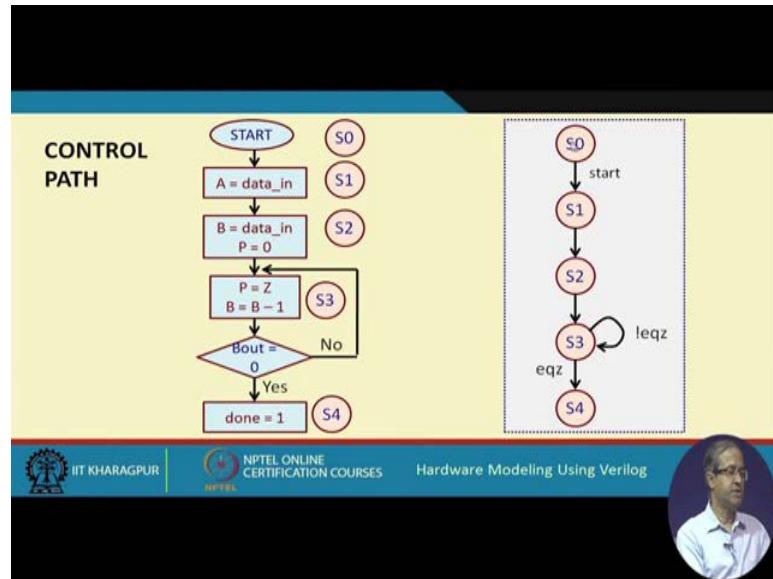
(Refer Slide Time: 11:43)



And it is generating a status signal equal to Z. So, the overall the picture will be like this I have my data path, which will be having this 5 control signals as I mentioned. And it is generating one status signal. Now I will have to generate or design a control path. The control path will be generating this control signals, and it will be taking the status as input. So, clock will be connected to both.

Well, data_in I am showing your, actual data is also coming to data path because in data path also data_in is there. And there are 2 additional signals in the control path, start means, now you start the multiplication. And after the multiplications finished, done will be activated so that from outside you will be knowing that the multiplication is done and the result is available. Now result where is available, it will be available in P, right, product, fine.

(Refer Slide Time: 12:52)



Now, let us see how we can design the control path. That same flow chart, we are showing it in a slightly different way, step wise we are trying to break. See we have to read the values of A and B right, in the first step we are saying `data_in` we are loading in A; second step `data_in`, you are loading in B and parallelly we can clear P because these are not related, you can do it together. And what is Z, just go back to the diagram? Z is actually the output of the adder. So, the output of the adder is going to P and you are decrementing B. So, as long as Bout, what is Bout? B out is the output of this B, Bout, as long as this is not 0 you are repeating this. And when it is 0, you are done, you activate the signal `done` equal to 1.

So, you see here you are defining the different steps and also identify some states side by side. So, the initial state I am calling it S0. So, whenever start is activated, I go to S1, then S2; then I go to S3, and I remain in S3 till Bout is equal to 0. So, when `Bout = 0` only then I go to state 4, S4. So, this state transition diagram for this example will look like this, this is my FSM. I start with S0, if start is 1, I go to S1. So, whenever start is activated then only I start, I go to S1. Then from S1 when the next clock comes, I go to S2; from S2 whenever the next clock comes, I go to S3. But in S3, I check this equal to Z signal. If it is not equal to 0, I remain in S3 at every clock, I just do a, you see $P = Z$ means what $A + B$ is Z. We are just adding, storing in P, next time again adding and storing in P, again adding & storing in P, this goes on. And as soon as equal to Z is active, I go to S4 and stop, right.

(Refer Slide Time: 15:27)

```
module MUL_datapath (eqz, LdA, LdB, LdP, clrP, decB, data_in, clk);
    input LdA, LdB, LdP, clrP, decB, clk;
    input [15:0] data_in;
    output eqz;
    wire [15:0] X, Y, Z, Bout, Bus;

    PIPO1 A (X, Bus, LdA, clk);
    PIPO2 P (Y, Z, LdP, clrP, clk);
    CNTR B (Bout, Bus, LdB, decB, clk);
    ADD AD (Z, X, Y);
    EQZ COMP (eqz, Bout);
endmodule
```

THE DATA PATH

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

So, let us look at the Verilog coding of the designs. Just let us have a look once more here. This is my data path you remember this, there is one register like this with only load. There is one register with load and clear, one register with load and decrement, this is actually a counter. So, here at the top level you are instantiating the modules. So, in the multiplier data path you see you can correlate with that previous example, previous diagram; that these are the signals or the ports that this multiplier data path requires. There is only one output equal to Z, others are all inputs. These are the control signals, and this is the clock. And I am assuming data_in is a 16-bit data. And all these X, Y, Z, B_out, Bus, what are these? Let us go back to the diagram once more. This is X output of A, I am calling as X; output of P, I am calling it Y; this is Z and this I am calling as Bus, fine.

So, we are instantiating the registers one by one, this is A, this is P, this is B. So, there are two kinds of parallel in parallel out register, one is a register without a clear, I am calling it PIPO1, A is the instantiated name and these are the parameters, output is X, input is Bus, the control is LdA (load control) and clk. PIPO2 is a parallel in parallel out register with load and clear control. These you need for the P register. So, Y is the output, Z is the input to this register and this is load (LdP), this is clear (clrP) and clk. B is a counter, at the output of the counter I call it Bout, input is Bus, you can load (LdB) it, you can decrement (decB) it and clk.

And of course, you have an adder where the inputs are X and Y, output is Z. And you have a comparator, we call it EQZ, this is the name. This is the output it takes Bout as the input and generates the output. So, whatever I have mentioned here, this is exactly the data path diagram that I have shown. From the data path diagram whatever components are there I directly instantiate those blocks in my top level data path module. Now I will have to specify all these modules. Now here for simplicity I am considering all of these at the behavior level, but in reality some of them you can also code in structural or gate level mode. PIPO1, PIPO1 has a output, input, load and clock.

(Refer Slide Time: 18:31)

```

module PIPO1 (dout, din, ld, clk);
    input [15:0] din;
    input ld, clk;
    output reg [15:0] dout;
    always @(posedge clk)
        if (ld) dout <= din;
    endmodule

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

module PIPO2 (dout, din, ld,
                clr, clk);
    input [15:0] din;
    input ld, clr, clk;
    output reg [15:0] dout;
    always @(posedge clk)
        if (clr) dout <= 16'b0;
        else if (ld) dout <= din;
    endmodule

module EQZ (eqz, data);
    input [15:0] data;
    output eqz;
    assign eqz = (data == 0);
endmodule

```

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

So, input is 16-bit, output is also 16-bit. So, it very simple, always @(posedge clk), if load is active, the input gets loaded, right. Let us look at this was PIPO2 first, PIPO2 is an extension of this where you also have a clear input in addition to load. So, here there will be an additional input clear. So, again, always @(posedge clk), if clear, the counter is cleared, then dout <= 16'b0, 16 bit 0, else if load, same dout <= din.

You say here is because we are using edge triggered means event we are using non blocking assignment. Now the adder, this is just a behavioral specification, out, in1, in2 these are all 16-bit quantities, so, always @(*), this out = in1 + in2. Then the comparator this equate, the input is the 16-bit data, output is a eqz. So, you just write like this, assign eqz = (data == 0); so you can either do this or you can also use the reduction operator NOR, basically you are using a NOR operation, right.

So, you can write this assign eqz equal to reduction NOR on data in, data. So, this is also a way to do it, $\text{data} == 0$ is a conditional, it will generate true or false that is 0 or 1; 0 means NOT 0, 1 means equal to 0. So, accordingly this output eqz will be affected. So, this completes our specification for the data path.

(Refer Slide Time: 20:39)

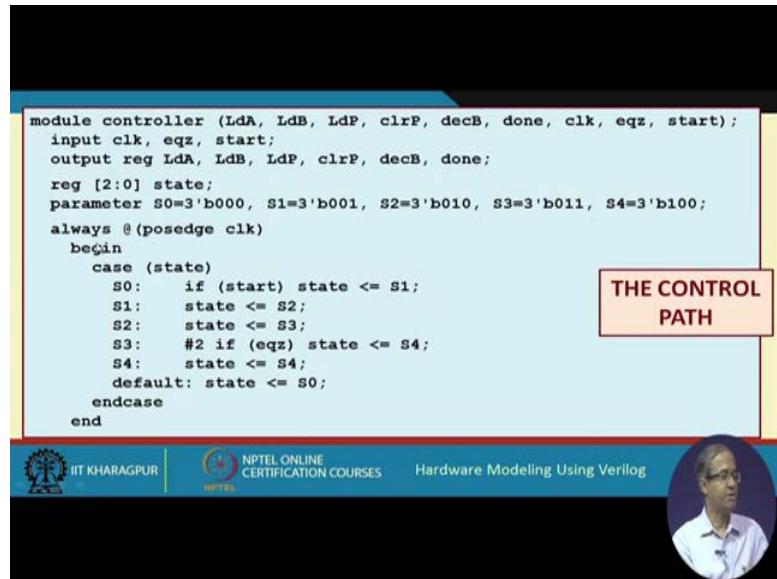
The screenshot shows a presentation slide with a yellow background. In the center, there is a red-bordered box containing Verilog code for a module named CNTR. The code defines inputs dout, din, ld, dec, and clk, and an output reg dout. It uses an always block with a posedge clk to check if ld is active (ld <= din) or dec is active (dec). If ld is active, it loads din into dout. If dec is active, it decrements dout by 1. The endmodule keyword concludes the code.

module CNTR (dout, din, ld, dec, clk);
 input [15:0] din;
 input ld, dec, clk;
 output reg [15:0] dout;
 always @ (posedge clk)
 if (ld) dout <= din;
 else if (dec) dout <= dout - 1;
endmodule

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the text "Hardware Modeling Using Verilog". To the right of the footer, there is a circular portrait of a man.

And of course, one thing is left the counter, the counter is output, input, load, decrement and clock, same way declaration. So, whenever there is a posedge clock, if load is active, you are loading it, else if decrement is active, you are decrementing the counter by 1. So, you have defined all the blocks of the data path and you have instantiated them in the top level module; so one PIPO1, one PIPO2, a counter, an adder and a comparator. Now you see this was my finite state machine, you remember this.

(Refer Slide Time: 21:23)



```
module controller (LdA, LdB, LdP, clrP, decB, done, clk, eqz, start);
    input clk, eqz, start;
    output reg LdA, LdB, LdP, clrP, decB, done;
    reg [2:0] state;
    parameter S0=3'b000, S1=3'b001, S2=3'b010, S3=3'b011, S4=3'b100;
    always @ (posedge clk)
    begin
        case (state)
            S0: if (start) state <= S1;
            S1: state <= S2;
            S2: state <= S3;
            S3: #2 if (eqz) state <= S4;
            S4: state <= S4;
            default: state <= S0;
        endcase
    end
endmodule
```

THE CONTROL PATH

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



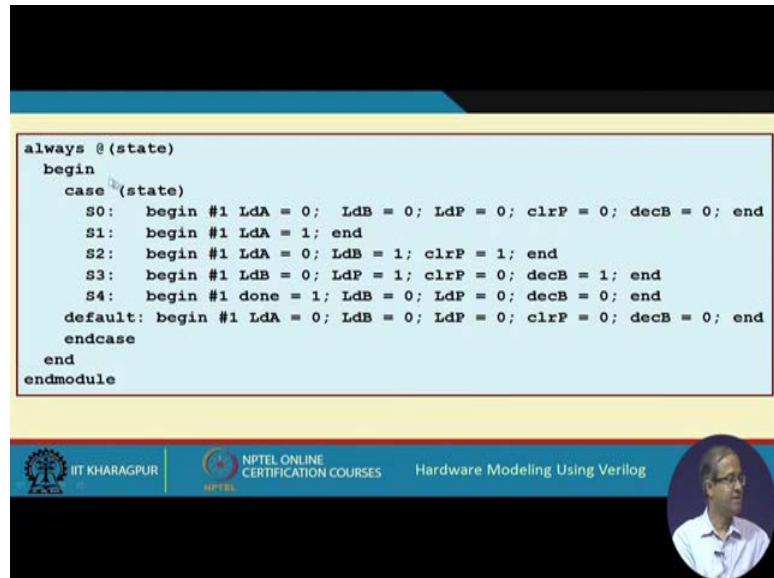
So now we will be coding the FSM, the controller. So, you have seen already earlier how to code an FSM, we had just taken examples of both Moore and Mealy type FSM. So, here the FSM is very simple, let us see. So, here the controller will be generating these signals LdA, LdB, LdP, clrP, decB and of course done. So, these are the output signals of the controller. And the inputs to the controller are of course clk, eqz signal which is coming from the data path, and the start, which is coming from outside.

Now, in this example there were 5 states. So, you need 3-bits to encode this state. So, I am using a parameter statement to define the states and give them names S0, S1, S2, S3 and S4. Well, the use of parameter makes it much easier for this code to understand otherwise you have to write 000 here, 001 here and so on. Now this always block is exactly the state transition diagram which you saw earlier, it says there is a case statement. So, if my present state is S0, so, you remain in S0 until start is activated. So, if start is true, then state becomes S1, now if it is in S1, if there is a clock, you will always go to S2.

Now if you are in S2, and a clock, you will always go to S3. Now if you are in S3, then we check if equal to 0 is activated or not. So, if it is true, that means, you are done, then you go to state S4, or else you remain in S3, no change. And once you are in S4, you do not change, you remain in S4 and by default if it starts with any undefined, it will be initialized to S0.

Now this delay we have given just for the purpose of synchronization and to get the correct output from simulation, because this little delay is required whenever you are going to be final state, right. Now next part of this controller will be to generate this signals LdA, LdB, LdP, clrB, decB.

(Refer Slide Time: 23:59)



```

always @ (state)
begin
    case (state)
        S0: begin #1 LdA = 0; LdB = 0; LdP = 0; clrP = 0; decB = 0; end
        S1: begin #1 LdA = 1; end
        S2: begin #1 LdB = 0; LdP = 1; clrP = 1; end
        S3: begin #1 LdB = 0; LdP = 1; clrP = 0; decB = 1; end
        S4: begin #1 done = 1; LdB = 0; LdP = 0; decB = 0; end
        default: begin #1 LdA = 0; LdB = 0; LdP = 0; clrP = 0; decB = 0; end
    endcase
end
endmodule

```

The slide also includes the IIT Kharagpur logo, NPTEL Online Certification Courses logo, and a portrait of a professor.

This is in a block where we are using blocking assignments. So, again in the S0 state, we are not activating any signals. So, all the signals are 0s. In state S1, you are loading the value of A, right, A equal to data A, so, LdA is activated. In S2, you are loading B, so, this LdA is deactivated and LdB is made 1; and in parallel we are initializing P = 0, so, clrP is also initialized to 1. In S3, we are actually doing the computation. So, you need not have to load B anymore.

So, LdP will have to be done in this loop, clrP is not required anymore, so, decB. So, P = X + Z and B = B - 1; these two things you are doing here. And S3 and here you will be remaining in S3 until eqz is true. So, this steps will be repeated, clock after clock. And finally, when you are in state S4, you activate done = 1, and again deactivate all the signals. So, if it is default, again you deactivate all the signals to 0, right. So, this is how you can specify the FSM, this is exactly like the FSM design which you saw earlier, given a state transition diagram, how to map it to a Verilog behavioral finite state machine description.

(Refer Slide Time: 25:44)

The screenshot shows a Verilog test bench titled "THE TEST BENCH". The code defines a module MUL_test with various signals and initializations. It includes a MUL_datapath instantiation and a controller CON instantiation. The test bench uses initial blocks to set up data_in, clk, start, and done signals. An always block generates a continuous clock. A monitor block is used to log time, DP.Y, and done. The simulation results show data_in values at specific time steps: 0, 6, 35, 45, 55, 65, 75, 85, and 88. The video player interface at the bottom indicates the current slide number is 417.

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

    MUL_datapath DP (eqz, LdA, LdB, LdP, clrP, decB, data_in, clk);
    controller CON (LdA, LdB, LdP, clrP, decB, done, clk, eqz, start);

    initial
        begin
            clk = 1'b0;
            #3 start = 1'b1;
            #500 $finish;
        end

    always #5 clk = ~clk;

    initial
        begin
            #17 data_in = 17;
            #10 data_in = 5;
        end

    initial
        begin
            $monitor ($time, " @d @b", DP.Y, done);
            $dumpfile ("mul.vcd");
            $dumpvars (0, MUL_test);
        end

endmodule
```

THE TEST BENCH

0	x x
6	x 0
35	0 0
45	17 0
55	34 0
65	51 0
75	68 0
85	85 0
88	85 1

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

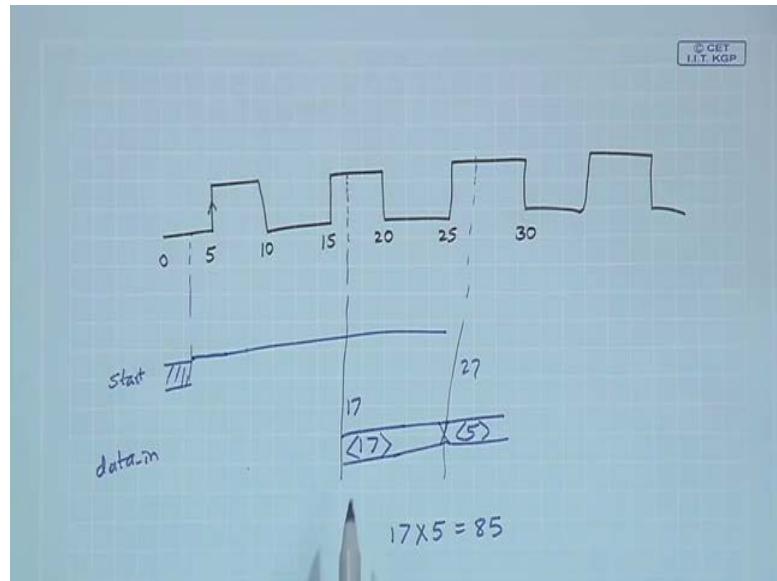
NPTEL

417

So, here we have done the same thing. Now the test bench we have written a very simple test bench to verify our design. So, you see what we have done, we called it MUL_test; data_in, this is coming from outside. So, this will be generated by the test bench somewhere we have initialized data_in, clk and start and done is the output signal. So, we have instantiated the data path and the controller with all the signals are specified, right.

So, in the initial block we are just initializing the clock to 0, and in this always block we are generating continuous clock with a delay of 5, clock equal to not clock. And at time 3, we are setting start to 1; that means, we say that we are starting the multiplication here, and from the next clock you should start. And we are repeating till time 500 where we finish. Now in this initial block, you see we are applying the data at time 17 and 5. So, you just see one thing, this is how we are generating the clock.

(Refer Slide Time: 27:04)



So, we start with time 0, this is 5, 10, 15, 20, 25, 30 and so on.

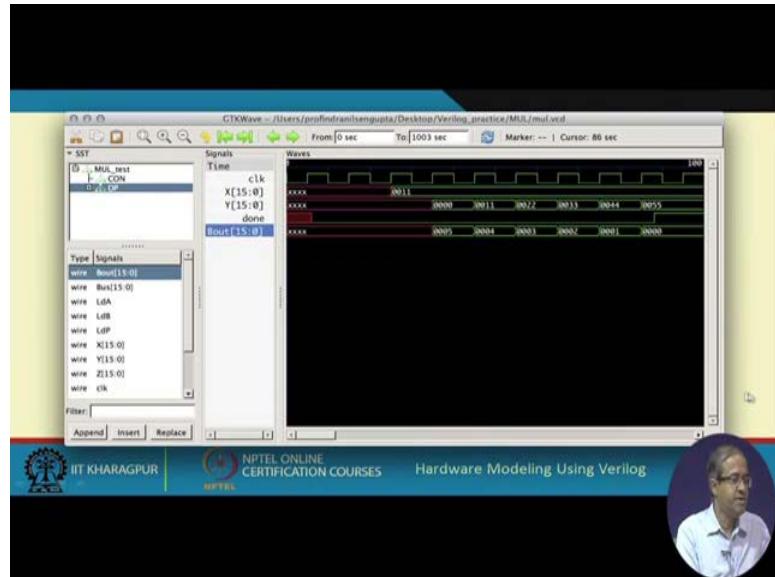
Now, if you think of the start signal, start you are activating at time 3, start. So, initially start was undefined. So, it was tri-state. So, start is activated at time 3. So, multiplication will start from here. So, the next edge that will be obtained is here. So, start will remain like this. So, what I doing data_in, the first data we are providing at time 17. At time 17 we are giving the first data, which is 17, the data is 17, also. And after a gap of time 10, that means, 27, at time 27 we are giving a the next data which is 5. So, we are trying to multiply 17 and 5; 17×5 should be 85.

So, we are actually giving it here. So, the load also will be activated here, after that let us see what happens. And here in the initial block, we are monitoring the value of Y, and what is Y? Y, if we just go back once. Y is the output of the product; that means the final value. You see this output of P is not directly available as a signal in this model, right. But you can access some signal from the other model from DP, you say Y is not available in the parameter, but you can write DP dot Y and access that Y and print the value, this is possible, ok.

So, we are printing that value and also the status of done. And you are dumping them in to a file. So, if you actually simulate this, this is the kind of output you are getting. So, you see the result finally, that you are getting is actually 85, these are the numbers in, because here you are printing in decimal, right, %d, value of Y, you are printing decimal.

So, it initially it was 17 and 5. So, 17, then 34, then 50. So, 17, it was added 0 plus 17 plus 17 plus 17 plus 17 plus 17 at the end done is 1. So, it is done, ok.

(Refer Slide Time: 30:12)



So, if you look at the timing diagram, the same thing is shown here. You see just exactly what I have said that this first data 17 in hexadecimal 0011, this is available in time 17; 10 and the second data was available here. So, the loading I am not shown here. So, I have shown the just the time steps, the output of the B, the counter.

So, it was initially it was 5, then it goes to 4, 3, 2, 1 and then 0. And the just output of the Y, it gets just added, it was initially 0, then 11 means 17, 11 in hexadecimal means 17; then 22 means 34, 33 means 51, 44 means 68. 55 means 85, and here develops done becomes 1; that means, it is done. So, in this example what we have seen is that given any reasonably complex design, we can separate out the data part containing the hardware blocks, and the control part which will be activating the hardware blocks in a proper sequence. This is the standard way of designing complex systems.

And, we shall be looking at some more examples also over the next couple of lectures.

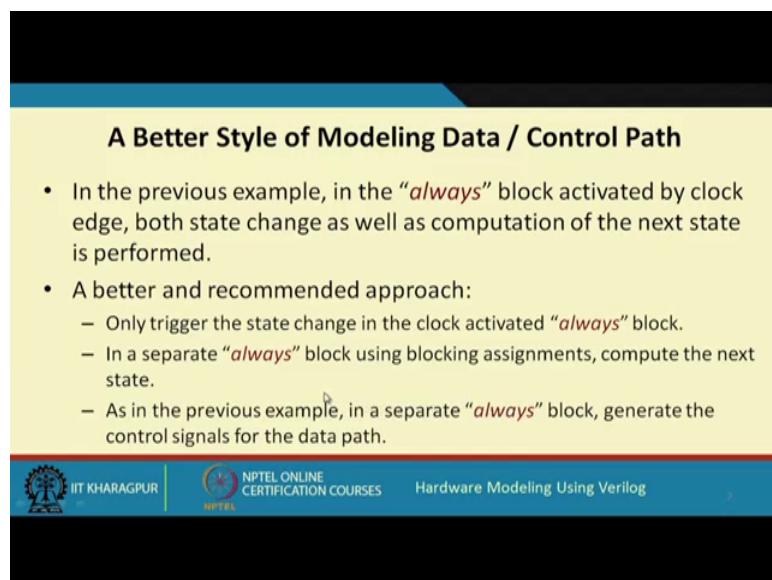
Thank you.

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

Lecture - 26
Datapath And Controller Design (Part 2)

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

(Refer Slide Time: 00:56)



A Better Style of Modeling Data / Control Path

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

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

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

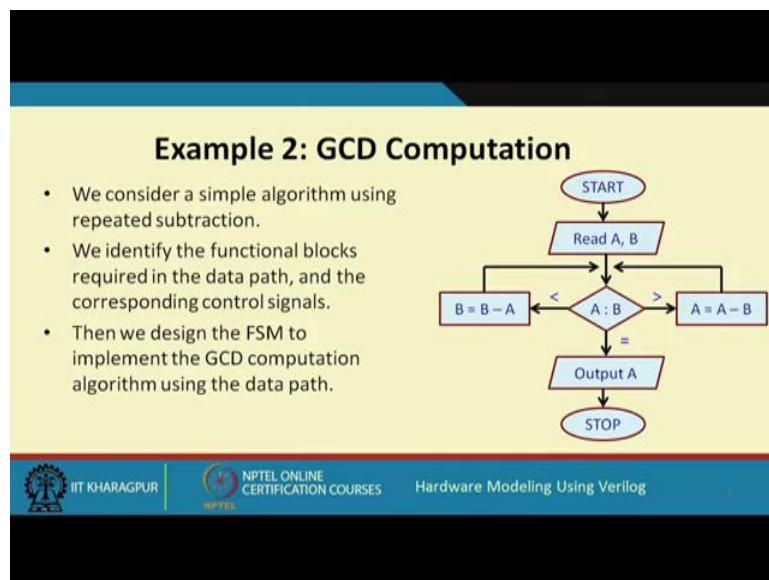
And in another always block which was based on blocking statements, there we were generating the control signals for the data path. Now here we shall talk about an alternate

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

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

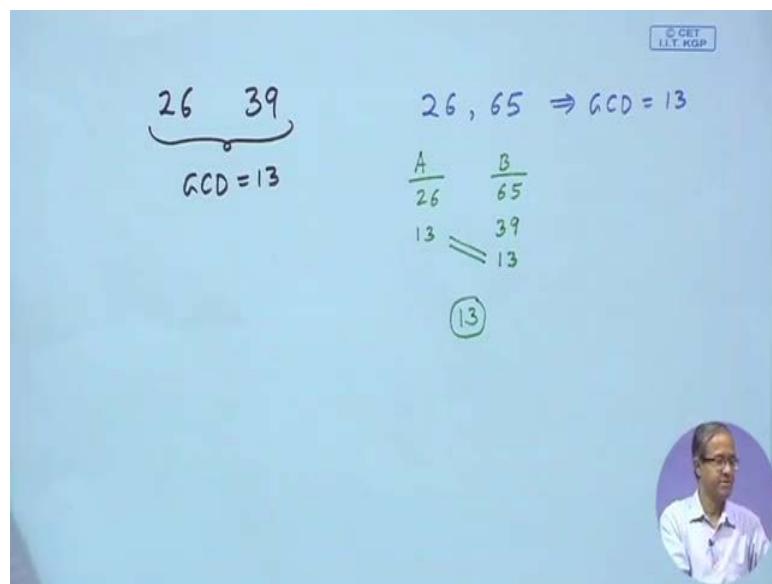
So, here you compute the next state as well as you generate the control signals for the data path. So, in this lecture we shall be taking another example, namely that of computing the GCD of 2 numbers, Greatest Common Divisor. And we shall first be following the previous approach for which what we did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

(Refer Slide Time: 03:34)



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

(Refer Slide Time: 03:56)



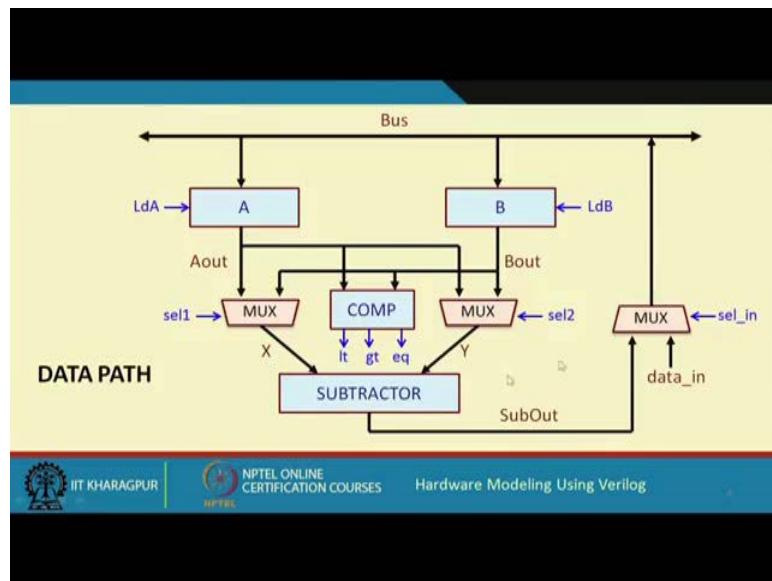
For example, If you have one number as 26, one number as 39, and if you compute the GCD of them, GCD will be 13, because 13 is the largest number which divides 26 as well as 39. Now the method we follow is the method of repeated subtraction. So, the algorithm is depicted in this flowchart you can see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

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

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

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

(Refer Slide Time: 07:27)

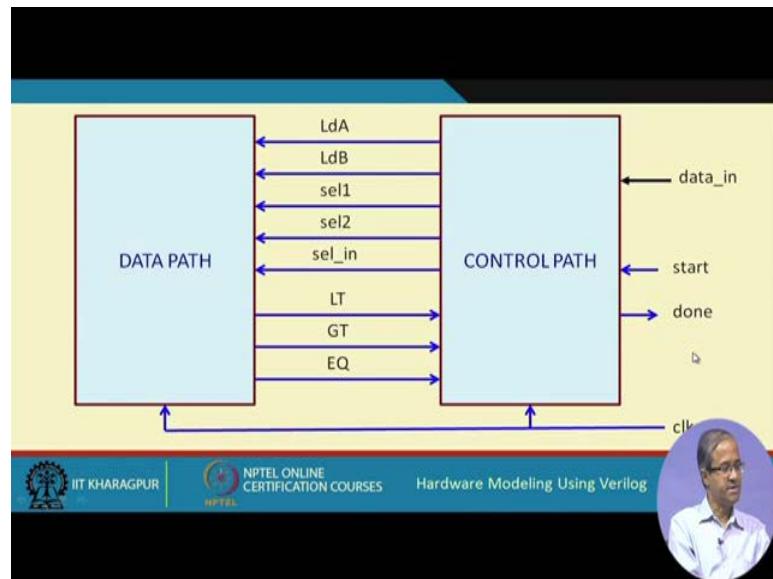


So, let us see the overall data path as I have said. So, we will need 2 registers A and B, to store our initial numbers. Then as I said we need a subtractor. The first input of the subtractor can be either A or B, that is why we have used a multiplexer with inputs coming from A and also from B; for the other input also there is another multiplexer input coming from A as well as B. There are select inputs of this MUX's, sel1 and sel2, they will select which one of A and B are selected. If sel1 is 0 and sel2 is 1, then here A will be selected; from here B will be selected.

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

And this selecting control signal of the multiplexer will select which of the inputs are selected. And during the normal computation, after subtraction the result will have to be stored back in to either A or B. For that I will have to select this input of the multiplexer this SubOut will be coming to the Bus and will be selecting again either LdA or LdB. So, this is how the data path functional components look like, and you can also see the control signals. LdA, LdB, sel1, sel2, sel_in and these are the status signal which are generated by the data path lt, gt and eq. So, the outputs of the multiplexer we are calling as X and Y, fine.

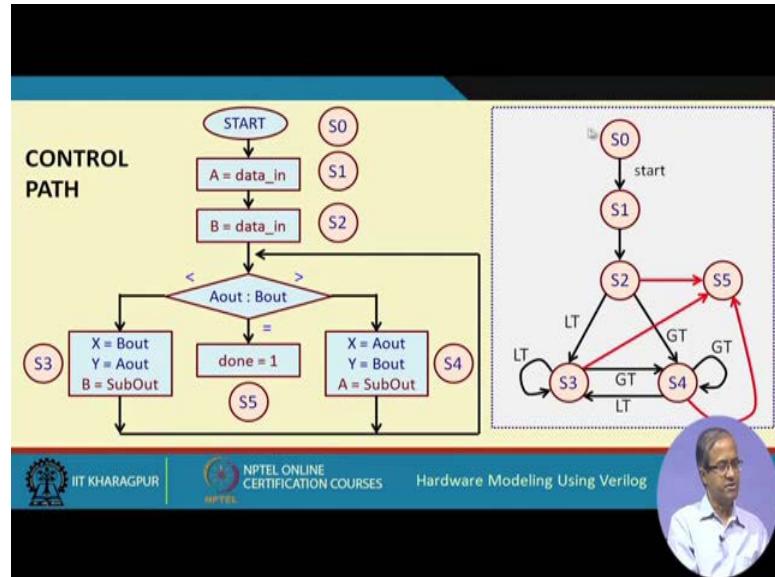
(Refer Slide Time: 09:36)



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

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

(Refer Slide Time: 10:20)



Now, let us come to the control path. Now that same flow chart, we are just showing in a slightly different way, stepwise from the beginning. Let us say we load the data_in signal which is coming from outside in to A first; then data_in to B first. Next we are calling them as states S0, initial state; this is S1, this is S2. Then we compare the outputs of the A or B register which we have called as Aout or Bout.

If it is less then Bout goes to X and Aout goes to Y, what does this mean? You see Bout goes to X and Aout to goes to Y; that means, I am selecting these multiplexers appropriately. Similarly, if it is greater, the other inputs are selected Aout goes to X, Bout to Y and the result goes back to B or to A. And if they are equal, the result is already in either A or B and you are activating done; this state is S3, S4 and S5. In terms of the state transition, you can depict this like this, starting from S0. So, whenever start is activated we go to S1; then to S2; then from S2 depending on the condition, 3 conditions are there, if it is less than, we go to state S3; if it is greater than, to S4; if it is equal, we go to, you see this red arrows indicate the equal to conditions, equal to, it goes to S5.

Similarly, when you are in S3, see after S3 you again go back, like this. So, you can again come to S3 or you can come to S4 or you can come to S5. So, from S3 again if it still remains less than, you remain in S3; if it is greater than, you go to S4; if it is equal,

you finally, go to S5. Similarly, for S4, if it is greater than, you remain here; if it is less than, you go to S3; if it is equal, you go to S5.

(Refer Slide Time: 12:32)

```

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

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

```

THE DATA PATH

So, this FSM we have to implement. So, here we are showing the FSM implementation using the approach which we showed with the previous example, first. So, we are first defining the data path where these are the signals gt, lt, eq are the output signals. And all the others, these 5 are the select signals and of course, data_in and clk are there. These are the control signals. And LdA, LdB, the multiplexer select and clk at the input signals, data_in I am assuming that is a 16-bit data and these are outputs. And this intermediate signals Aout, Bout, X, Y, Bus and the output of the subtractor, these are all assumed to be 16-bit buses of type wire.

Now, just in this schematic we have seen we use 2 registers A and B, 3 multiplexers, 1 subtractor and a comparator. So, we have just instantiated that many functional blocks here. Parallel in parallel out is a register, we have already defined, I will show it. So, here Aout is the output, Bus is the input, load and clock are the control signals. Similarly, for PIPO B; for the multiplexer this X is the output, Aout Bout are the inputs and sel1 is the select. Similarly, for the second multiplexer; for the third multiplexer Bus is the output, SubOut and data_in are the inputs and sel_in is the select. And there is a subtractor where SubOut is the output, X and Y are the inputs and there is a comparator finally, lt, gt, eq are the outputs and these are the 2 input numbers, right.

(Refer Slide Time: 14:30)

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

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

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

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

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

So, you see for the register, the PIPO, data_out, data_in, load, clk, it is very simple, data_in is the input, load, clock are single bit control signals. So, always at posedge of the clock, if the load signal is active then data_in goes to data_out, it is loaded. Subtractor is a combinational circuit, very simple, this in1, in2, out are all 16-bit numbers; so that whenever anything changes, always at star, out equal to in1 minus in2.

Comparator is also very simple, data1 and date2 are two 16-bit numbers, and these are the outputs. We have 3 assign statements, assign lt equal to data1 less than data2, which means, if this condition is true, that condition will be stored in lt, condition true means it is stored as 1. So, 1 will be stored in lt. If it is false, 0 will be stored. Similarly, for this one, if this condition is true, 1 will be stored otherwise 0; similarly, equal, right. MUX also is very similar, this is actually a set of 16 2 to 1 multiplexers because we are multiplexing 2 inputs in0 and in1, each of them have 16-bits. So, actually these are vectors out, in1, in2, depending on the select line, we are either selecting in0 or in1, ok.

(Refer Slide Time: 16:40)

The slide features a block of Verilog code for a controller module. The code defines a state vector 'state' of width 3 bits, with parameters for states S0 through S5. It includes two 'always' blocks: one triggered by the clock that implements a state transition logic based on 'start', 'lt', 'gt', and 'eq' inputs; and another that checks the current state against the parameter values to determine the next state. A red box labeled 'THE CONTROL PATH' is overlaid on the right side of the code area. In the bottom right corner, there is a circular portrait of a man.

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

Now the controller, the controller again it will be generating all the control signals for the data path done, clk and lt, gt, eq and start are the signals which it is taking as input, these are inputs. And the signals which is generating for the data path, these are outputs, and because they are using a procedural block, we have declared them of type reg. Now in this example because there are 6 states, we need a 3-bit state vector. So, we define it as a 3-bit state vector called state, and using parameter we call this state are S0, S1, S2, S3, and S4 and S5. Now in the first always block triggered by the clock we are just implementing that flow chart whatever I had shown. So, initially we are in state S0, if start is active, this state becomes S1.

Well, if you are in state S1 irrespective of anything else next state will be S2. Well if we are in S2 then if it is equal go to S5; if less than, S3; greater than, S4. Same thing is done for this states S3 and S4. And whenever you are in state S5; that means you are done, you remain in state S5. Well and initially if this state is not initialized. So, there is a default case which initializes the state to state S0. This is for the state calculation. Then there is another always block which checks this state.

(Refer Slide Time: 18:18)



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

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

Well, in S0, well, so, while you are going from S0 to S1, you will have to load A. So, that is why LdA is active, you are selecting the other input of the multiplexer sel_in, so that data_in gets in to the Bus, while LdB and done are 0. And when you are in S1 going to S2, you have to load B. So, LdB is 1, again sel1 is 1, and here when you are in S2, S3, or S4, it is the same, you check the conditions.

If it is equal, you activate the done signal, else if it is less than, then you select B in the first one and A in the second multiplexer, load B, result will go back to B. And else you select A in the first multiplexer, subtractor, the multiplexer, B in the second multiplexer, you activate A, result will go back to A. Similarly, S3 this is same, and in the same way S4 is also same, ok.

(Refer Slide Time: 19:36)

```
module multiplexer;
    input [1:0] sel_in;
    input [1:0] sel1, sel2;
    output [1:0] ldA, ldB;
    output done;

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

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

And finally, in S5 when you are done, you activate the signal, done equal to 1 and deactivate all the select lines and loads. So, that nothing else will proceed after that. And for default, we are just setting LdA = 0, LdB = 0. So, you see in this example the FSM we are actually generating all the control signals, as per the flow chart whatever you require.

Well, whenever you want to select a multiplexer, one input of the multiplexer, we are activating sel1 and the sel2 control signals accordingly. When you have to load the result in to A or B we are activating either LdA or LdB. So, you can just compare that flow chart with this FSM description and you can see the correlation, they are the same, fine.

(Refer Slide Time: 20:32)

The screenshot shows a Verilog test bench code. The title 'THE TEST BENCH' is displayed in a purple box. The code includes declarations for modules GCD_test, GCD_datapath DP, and controller CON. It sets initial values for data_in, clk, start, and done. The simulation starts at time 0, activates start at time 3, and continues until time 1000. A clock generation block toggles the clk signal every 5 units of time.

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

    reg [15:0] A, B;

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

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

    always #5 clk = ~clk;

```

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

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

(Refer Slide Time: 21:04)

The screenshot shows a Verilog test bench code with a simulation waveform overlay. The waveform displays the output Aout over time, showing the progression of the GCD calculation. The test bench monitors the output and dumps the results to a VCD file. The simulation results show the inputs and the resulting output value.

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

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

endmodule
```

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

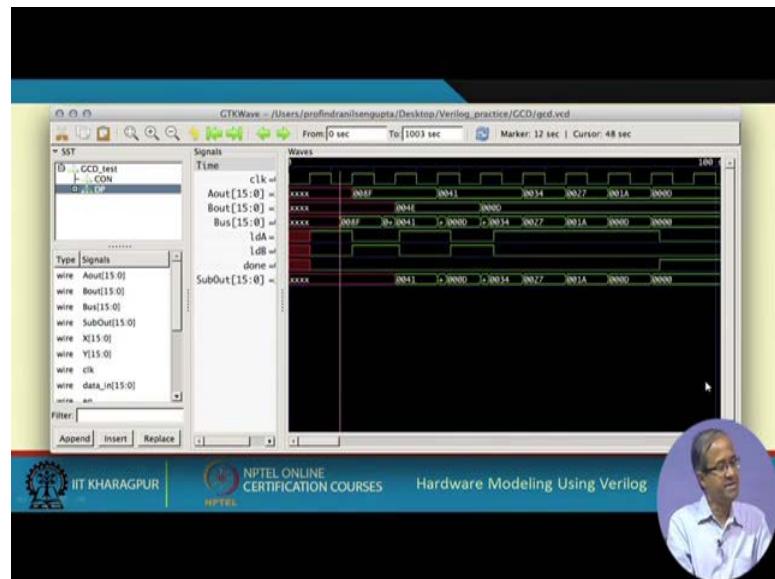
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

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

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

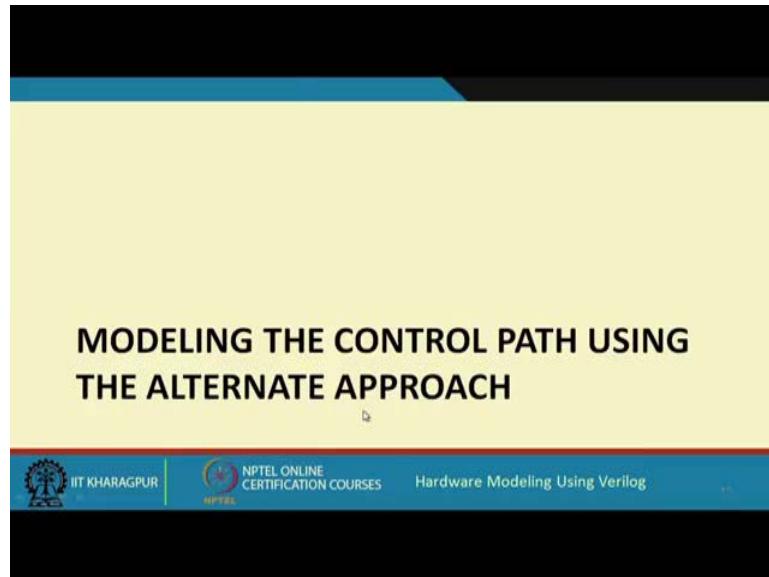
So, if you just simulate this, the result comes like this. So, the three things that are printed are time; the value of A, output of A; and done. So, A changes and finally, it stops at 13, which is the final result when done also becomes 1, right. And just a simulation snap shot also is shown here.

(Refer Slide Time: 22:14)



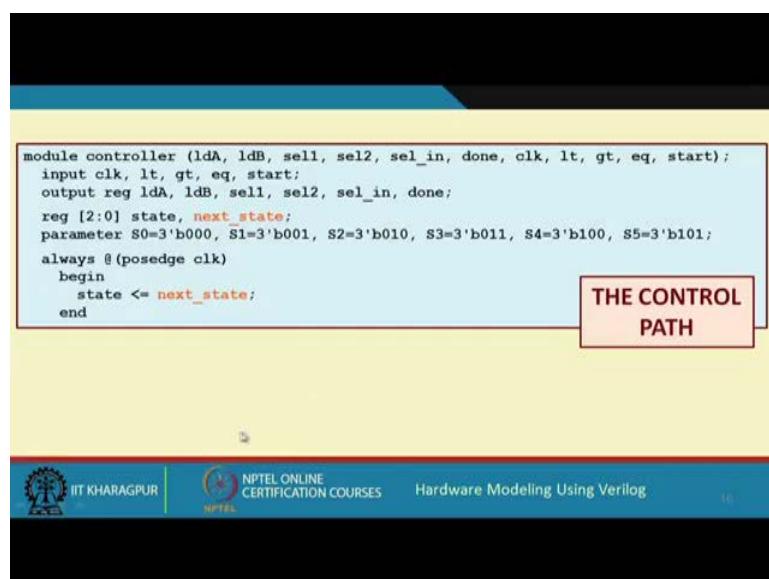
So, here also you can see the different signal values clock, Aout, Bout, Bus, LdA, LdB. So, you can see the values of Aout and Bout are changing. You see initially the values are 143 and 78. So, 143 means 8f , 78 means 4e. So, 8f minus 4e it becomes 41; 4e minus 41 it becomes 0d; 41 minus 0d becomes 34; 34 minus 0d is 27; 27 minus 0d is 1A; 1A minus 0d is 0d, they are equal. Now you stop, done gets activated here when they are equal. So, the result is 000d which is 13, right, fine.

(Refer Slide Time: 23:03)



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

(Refer Slide Time: 23:55)



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

(Refer Slide Time: 24:19)

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

Now, here it only says, state <= next_state. And the next_state is calculated in the second always block.

So, here earlier these red things were not there. The remaining control signals were there, now we have made a little modification here, we have also added the statements which compute or calculate the next_state. So, whenever in S2 and it is equal, your next_state will be S5; when you are in S2, it is less than, your next_state will be S; if it is greater than, next_state will be S4.

(Refer Slide Time: 24:55)

```
  S4:    if (eq) begin done = 1; next_state = S5; end
         else if (lt) begin
                     sel1 = 1; sel2 = 0; sel_in = 0; next_state = S3;
                     #1 ldA = 0; ldB = 1;
                     end
         else if (gt) begin
                     sel1 = 0; sel2 = 1; sel_in = 0; next_state = S4;
                     #1 ldA = 1; ldB = 0;
                     end
      S5:   begin
                 done = 1; sel1 = 0; sel2 = 0; ldA = 0;
                 ldB = 0; next_state = S5;
                 end
      default: begin ldA = 0; ldB = 0; next_state = S0; end
endcase
end

endmodule
```

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

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

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

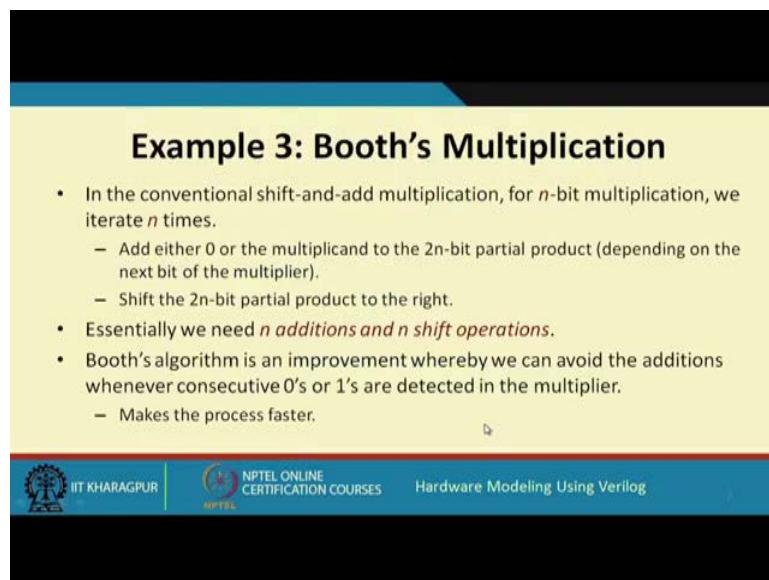
Thank you.

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

Lecture - 27
Datapath And Controller Design (Part 3)

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

(Refer Slide Time: 00:38)



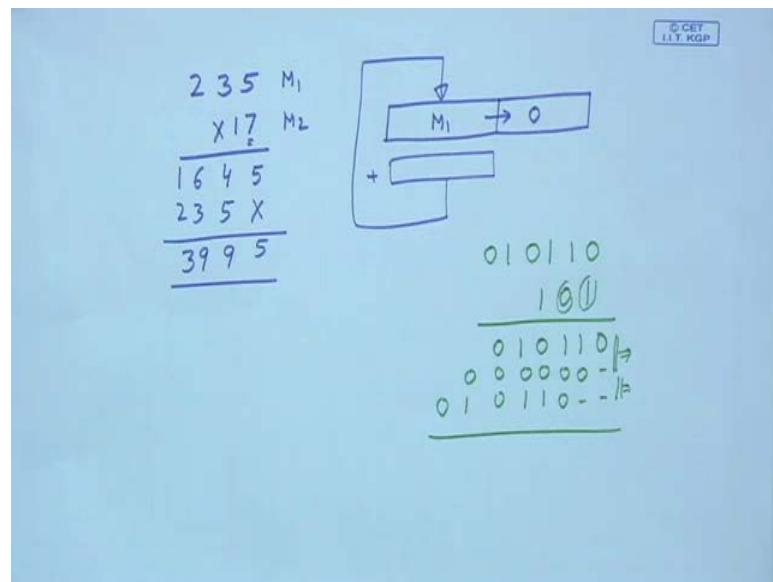
Example 3: Booth's Multiplication

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

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

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

(Refer Slide Time: 01:10)



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

We keep the partial result in a pair of registers, well let us say this is your M, let us say this is your M₁ and M₂. Let us say suppose we load M₁ here and load this with 0. And depending on the next digit or next bit of M₂, we add something to M₁. And after adding these result goes back to M₁. Then we shift M₁ to the right one place. Instead of shifting left and adding, we add in the same position, but the partial product will shift right, it is same thing for example, 1 6 4 5 if you shift right one place and add 235, that is also the same thing, right. So, this is how it works.

Now, in a normal shift and add multiplication this as example I showed for decimal, here you work in binary where the bits of the number are considered. Suppose when we add let us say 010110 with let us say 101, we do the same thing like here I am showing this process, when you multiply 1 with this, so, it remains. When you multiply 0, so, everything is 0, there is a gap when you multiply 1 again, there are 2 gaps 011010 then we add all of them up. This is how we do, shift and add multiplication. So, what you do?

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

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

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

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

(Refer Slide Time: 05:50)

Basic Idea Behind Booth's Algorithm

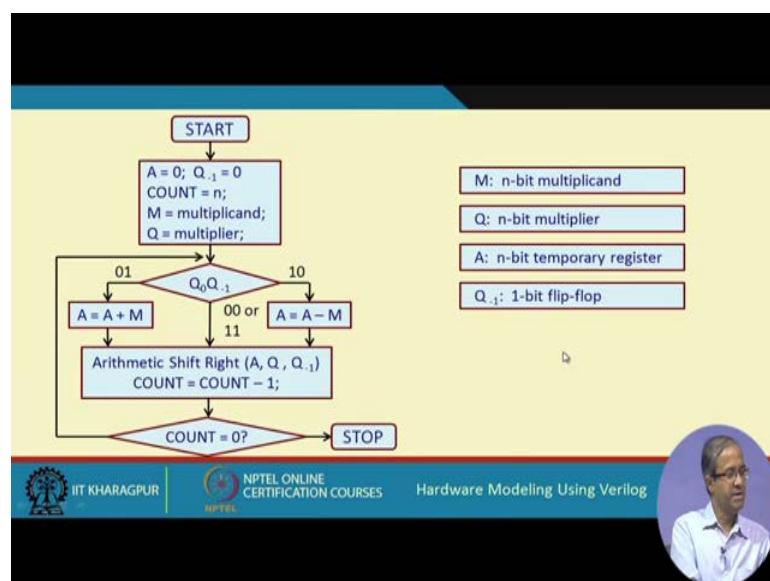
- We inspect two bits of the multiplier (Q_i, Q_{i-1}) at a time.
 - If the bits are same (00 or 11), we only shift the partial product.
 - If the bits are 01, we do an addition and then shift.
 - If the bits are 10, we do a subtraction and then shift.
- Q_{-1} is assumed to be equal to 0.
- Significantly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



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

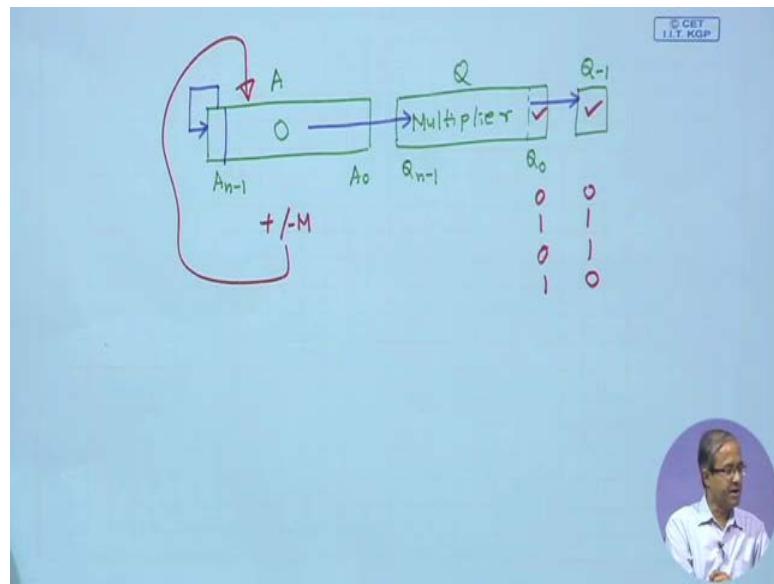
So, the basic idea behind the algorithm is that we inspect 2-bits of the multiplier at a time, i th and $i-1$ th bit. So, the rule is very simple. It says if the 2-bits are either 00 or 11, there is no need to do any addition or subtraction, then you do only shifting. But if you find that these 2-bits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

(Refer Slide Time: 06:49)



And because we are skipping through 00 and 11, this will reduce the number of addition, subtractions in the process. This is the flow chart of the Booth's multiplier. So, you see here we have a register A initialize to 0, and Q which contains the multiplier. So, it is something like this.

(Refer Slide Time: 07:08)



There are 2 registers we are saying this is A and this is Q. Initially A is initialized to 0 and Q is loaded with the multiplier. And we check this bit and there is another flip-flop we say call it Q_{-1} . Because this Q will start from Q_0 up to Q_{n-1} , if it is n bit register. And A will start from A_0 up to A_{n-1} . So, let us say this is Q_{-1} ; so will be checking 2-bits at a time like this. So, what we do? And there is a register count, which will contain the number of bits that mean how many times you have to repeat, that is n. And M contains multiplicand and this Q contains multiplier. So, so in every step we check Q_0 and Q_{-1} , we check this Q_0 and Q_{-1} whether they are 00, 11, 01 or 10.

So, we check the conditions. So, if it is 00 or 11, we skip the addition or subtraction step. If it is 01, we do $A = A + M$; if it is 10, we do $A = A - M$. So, with this A, we either add or subtract M, and the result we store back in to A, right. And then we do a right shift, we do arithmetic right shift of A, Q and Q_{-1} altogether. This means I right shift here in to multiplier, will get right shifted in to Q_{-1} . And arithmetic right shift means the most significant bit of A, which is the sign bit that will be shifted back.

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

(Refer Slide Time: 09:53)

A	Q	Q ₋₁	
0 0 0 0 0	0 1 1 0	1 0	Initialization
0 1 0 1 0	0 1 1 0 1	0 0	$A = A - M$ Step 1
0 0 1 0 1	0 0 1 1	0 1	shift
1 1 0 1 1	0 0 1 1 0	1 0	$A = A + M$ Step 2
1 1 1 0 1	1 0 0 1	1 0	shift
0 0 1 1 1	1 0 0 1 1	0 0	$A = A - M$ Step 3
0 0 0 1 1	1 1 0 0	1 1	shift
0 0 0 0 1	1 1 1 1	0 1	Shift Step 4
1 0 1 1 1	1 1 1 0 0	1 0	$A = A + M$ Step 5
1 1 0 1 1	1 1 1 1 0	0 0	Shift

Example 1: (-10) x (13)
 Assume 5-bit numbers.
 M: (10110)₂
 -M: (01010)₂
 Q: (01101)₂
 Product \triangleq -130
 $= (110111110)_2$

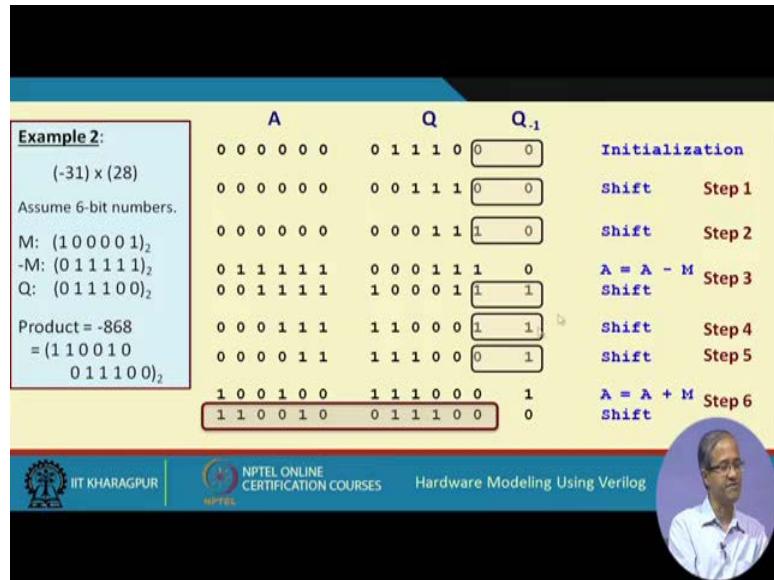
IIT Kharagpur | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

Let us say we multiply -10 with 13, we assume these are 5-bit numbers. M, - 10 is 10110 in 2's complement representation. Well, just for convenience I am also showing -M, -M will mean +10; +10 in binaries 01010. And Q the multiplier is 13. So, we load A with 0, Q with the multiplier and Q₋₁ is 0. Then we check these 2-bits Q₀ and Q₋₁. So, it is 10 means we have to subtract M. Well, just for convenience subtracting M means adding minus M. So, it will be easier to understand that is why I have shown -M. So, we subtract M means we add 01010, -M we add, So A becomes 01010. Then we do a right shift, whole right shift, this 0 gets shifted and everything gets shifted right one place and this 1 gets shifted here.

So, this process will repeat, next we say this is 01, which means we have to add, to this A will have to add M. So, if you 00101 + 10110, if you do, you will see, you will get the result 11011, this you can check if you do it. Then you shift in the same place, this is arithmetic shift, so, this 1 gets shifted. Now the 2-bits are 10, so, it is again subtraction. So, we again do a subtraction; that means, we add -M to this 01010, add to this, you get to 00111, then do a right shift. Now we have 11; 11 means no need to do add or subtract

just shift. Simply shift, simply shift now you get 01; that means, addition. So, it is 5-bits, 5 steps are over and your final result -130, which is in 2's complement is this, you see the same result is obtained. This is our final result, ok.

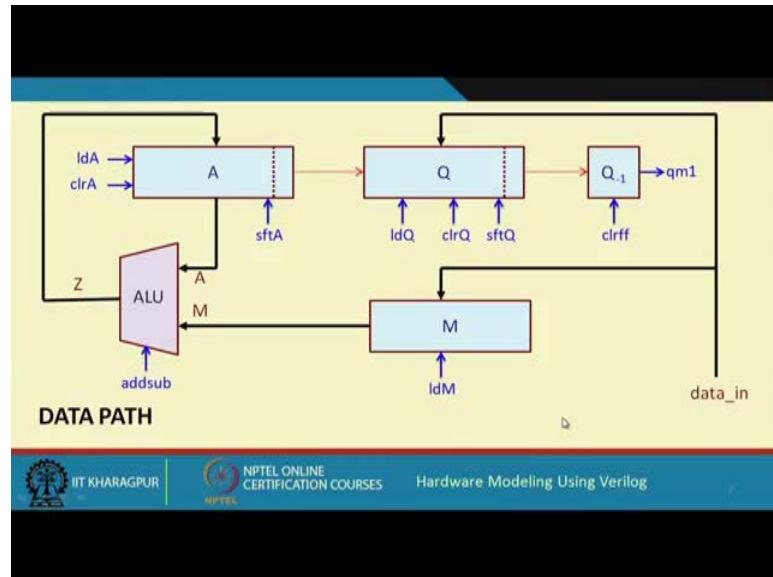
(Refer Slide Time: 12:09)



Let us take another example. Suppose we are multiplying -31 with 28 and in 6-bit representation let us say number have 6-bits, multiplier Q, 28 is in 6-bits, A is also 6-bits. So, M, -31 and also -M is shown. So, you see 00 just shifting, no addition subtraction only shift right. So, again 00 shift again now it is 10, subtract. So, -M you add to this, it becomes this, then shift; then again 11 just only shift; so again 11 just shift again. Now 01, so, you will have to add to this, you add M, this will be the result and then shift and this will be your final result. So, you see here you need only 2 additions or subtractions, rest of the time you are only shifting.

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

(Refer Slide Time: 13:33)

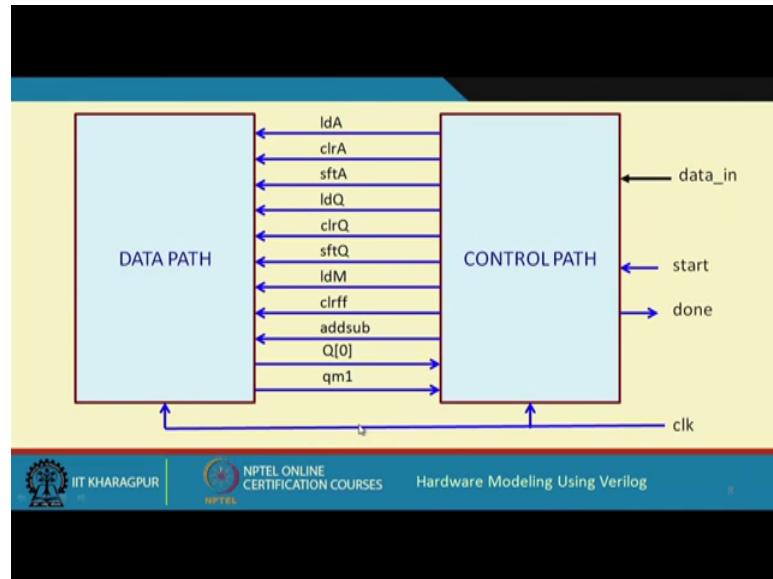


So now let us come to the data path, now I have understood what we are doing. First thing is that A, Q and Q₋₁ must be connected as a shift register. So, A is an n-bit register, Q is an n-bit register, Q₋₁ is a flip-flop. So, an M contains the multiplicand. So, from outside we will have to load the multiplicand,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

Well of course, we do not require this clrQ, but because we are using 2 registers of the same type, this A needs to be cleared. So, there is a clear control signal, and there is a load control signal also because the output of the addition or subtraction will get stored back in to A for that there is a ldA. And there is a shift control signal sftA, sftQ, this is the shift register mode control signal. And for the flip-flop there is a signal to clear it reset, clrff and the output of the flip-flop we are calling it qm1.

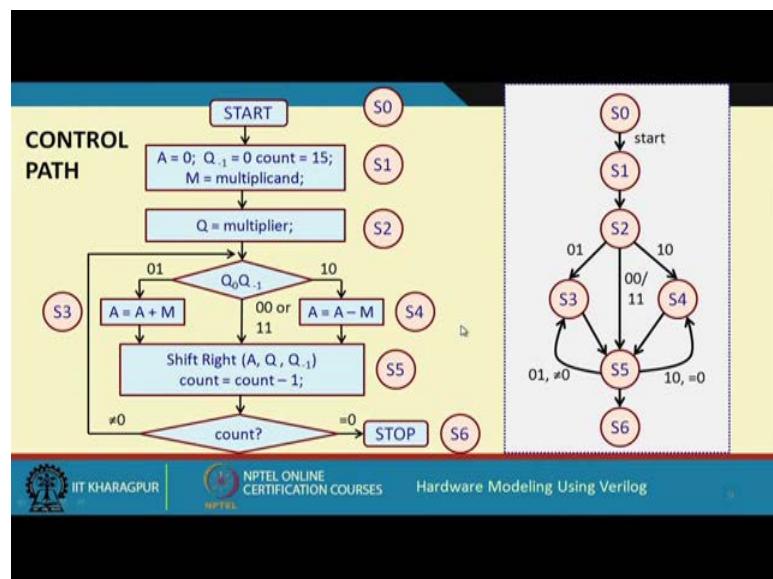
And this ALU takes A and M as the 2 inputs, and there is a control signal, addsub. So, if it is 0, it will add; if it is 1, it will subtract; the output let us call it Z.

(Refer Slide Time: 15:09)



So, it is very simple. So, data path again there are so many control signals as we illuminate it, and from the data path $Q[0]$ and $qm1$ are coming this $Q[0]$, this 0 and $qm1$. Because these 2-bits, we have to check to decide that whether to add or subtract or only shift, right. And the rest signals are the same. Now let us come to the control path. So, again the same flow chart I am showing and here I am identifying the states.

(Refer Slide Time: 15:34)



Now, this loading of the data I have divided in two states, multiplicand M , multiplier Q , you see $count = 15$, you can do together because $count$ here although I have not shown

here, there will be another register called count, which will be loaded with 15 and to be decremented, right, but that can be done in parallel. Because from data_in, you are loading either M or Q, you cannot do these 2 things in together in parallel. That is why loading M and loading Q are in 2 different states. Then depending on this $Q_0\ Q_{-1}$, if it is 0, I do addition, I call it S3; 10, subtraction, I call it S4 or otherwise shifting, I call it S5. At the end when you are done, count = 0, I call it S6.

So, the state transition diagram will look like this if you see compare side by side. So, from S0 whenever we give the start signal, it goes to S1, then to S2. And from S2 depending on the result of the comparison, I can either go to S3 or S5 or S4 depending on these 2-bits. Now once I am in S3 well I can either go back, you see from here I go to S5, sorry, from S3 I go to S5 there is only one path. S3 to S5 and also S4 to S5, S3 to S5, S4 to S5. And from S5 there is a comparison, I either go to S6 or I go back in this comparison and can go to S3 or to S4. So, from S5 I can go to S3 or to S4. So, the condition for going to S3 is that $Q_0\ Q_{-1}$ is 01 and this count is not 0. And we go to S4 if it is 10 and the count is not 0, sorry, this will also not 0.

And if it is 0 then it will come to S6, right, fine.

(Refer Slide Time: 17:55)

```

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

  assign eqz = ~&count;

  shiftreg AR (A, Z, A[15], clk, ldA, clrA, sftA);
  shiftreg QR (Q, data_in, A[0], clk, ldQ, clrQ, sftQ);
  dff QM1 (Q[0], qm1, clk, clrrff);
  PIP0 MR (data_in, M, clk, ldM);
  ALU AS (Z, A, M, addsub);
  counter CR (count, decr, ldcnt, clk);
endmodule

```

THE DATA PATH

So now we can straight away write down the data path just from this diagram which is shown the same thing, just see here these are the parameters. The exact the signals that we have identified their ldA, ldQ, ldM, clrA, clrQ, clrrff, these are all input signals;

data_in which is coming from outside to load M and Q, this is an input signal 16-bits. And qm1, eqz, these are output signal which will go in to the controller. And A, M, Q, Z, a temporary signals of type wire. And count, I need to initialize the count to 15, right, 15 and go down to 0. So, 15 I can store in 4-bits, 5-bits. So, I use a counter of appropriate size. So, 15 I could have used a 3-bit, 3 to 0, a 4-bit counter also. So, there is no problem.

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

(Refer Slide Time: 20:08)

```

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

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

module PIPD (data_out,data_in, clk, load);
  input [15:0] data_in;
  input load, clk;
  output reg [15:0] data_out;

  always @(posedge clk)
    if (load) data_out <= data_in;
endmodule

module diff (d, q, clk, clr);
  input d, clk, clr;
  output reg q;

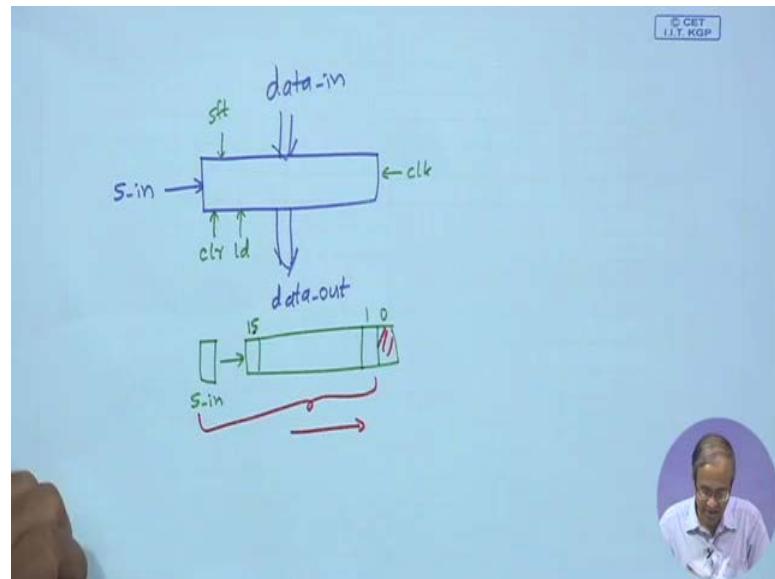
  always @(posedge clk)
    if (clr) q <= 0;
    else q <= d;
endmodule

```

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

register, then you can load this register with data_in, ld signal. Then there is also another signal sft, so, when you want to shift it.

(Refer Slide Time: 21:08)

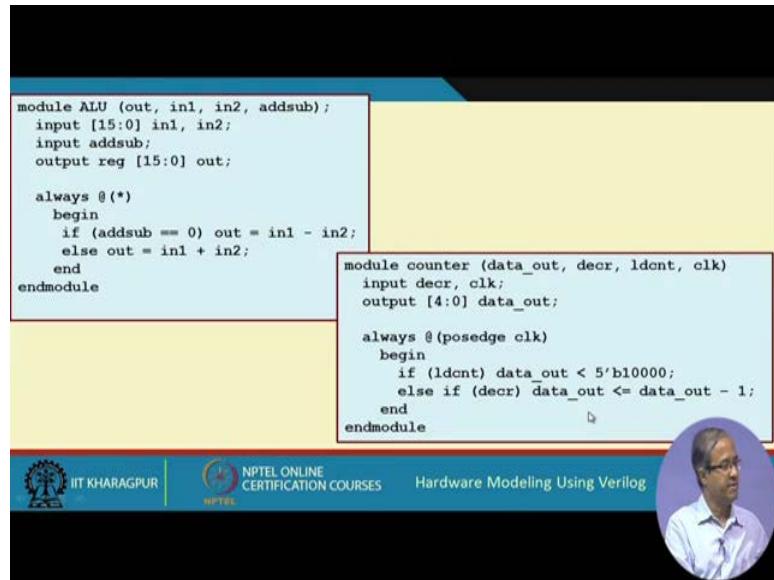


And of course, there will be a clk. These are all the signals, I means at the boundary of a of this shift register. So, this is a shift register we have designed here, data_in is a 16-bit number, data_out also 16-bit and this is our description. So, whenever clk comes this is synchronous clr and ld, we check if clr is 1; if clr is 1, data_out becomes 0. So, initialize it to 0 else if ld is active then whatever there is data_in that goes to data, it is loaded. Else if sft control is active then this s_in and data_out[15:1], this entire thing gets shifted left by one position in to data_out, right.

So, this is shift right. So, this is how we are implementing shift right; data_out[15:1] you see, data_out, data_out is a 16-bit quantity, right. This is your bit0 this is bit1 and this is bit15. And there is something which is coming from outside this is your s_in, now you want to shift it right. So, what we expect is that, this bit should go out and whatever is here, this should be shifted right one position and go here. So, s_in and bit number 15 to 1, whole together should be assigned here. That is exactly what you wrote, s_in and data_out[15:1] is assigned to data_out. This creates the shifting. Similar the PIPO register is very simple parallel in parallel out.

So, it has a data_out, data_in, clk and load. So, here whenever clk is coming, if load is active, data_in goes to date_out. And then we have the flip-flop, we already seen a flip-flop description earlier.

(Refer Slide Time: 23:26)



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

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

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

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

IIT Kharagpur | NPTEL Online Certification Courses | Hardware Modeling Using Verilog

So, it has a clr input, if clr is active, then q becomes 0 output else q becomes equal to the input d. Then ALU, out, the 2 input and there is a control signal, if addsub is equal to 0, you do subtraction; if it is 1, you do addition. And there is a counter so, if there is a ldcnt signal, so, initially if you activate ldcnt; so, it will be initialized to, data_out is initialized to 10000, else if decrement is active data_out is decremented by 1. So, you load it with 16, 10000 mean 16, you decrement it one by one and you check whether the result is 0 or not.

(Refer Slide Time: 24:18)



THE CONTROL PATH

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

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

So, here again the parameters or the arguments are the same as whatever the controller is generating or taking as inputs q0, qm1. There are 7 states S0 to S6. So, the state transitions are actually specified as per that diagram. From S0 if start is active go to S1; from S1 go to S2; now in S2 you check whether these 2-bits q0 and qm1 are 01, if it is 01 go to S3; if it is 10 go to S4 or otherwise go to S5 means 00 or 11. So, which is S3 you straight away go to S5; S4 also goes to S5. So, in S5 also you check if bits are 00 and not equal to 0 yet then go to S3.

(Refer Slide Time: 25:44)

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

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

If it is 10 and not equal to 0 go to S4, but if it was equal to 0 you are done, you go to S6 and once in S6 you remain in S6. Similarly, in the other block this is the blocking assignments, here you have an always with state, whenever state changes you just activate this. So, here all the control signals will be generated appropriately, just you can check this.

So, here you are loading M, $ldM = 1$, loading count, the counter also you are loading, the flip-flop you are clearing, A also you are clearing, you are making $A=0$, you are activating all the signals together. Then in S2 you are loading Q. Now in S3, you are activating $addsub = 1$ which means addition. S4, $addsub = 0$, which means subtraction. In this way you just follow the flow chart and see that whether we have activated the signals properly or not, ok.

(Refer Slide Time: 26:46)

The slide content is as follows:

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

At the bottom of the slide, there is a footer bar with the following logos and text:

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

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

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

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

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

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

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

Thank you.

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

Lecture - 28
Synthesizable Verilog

So, in the lectures so far we have seen various ways to model both combinational and sequential circuits. So, if you recall we talked about and also learnt how to model. So, called behavioral specifications of some digital system block and also we learnt how to model some designs in a structured way, structured design. So, broadly speaking designs can be categorized into behavioral and structural and one thing.

So far whatever results that we discussed we showed they were based on simulation only, but in reality whenever you are trying to design a hardware block either on a FPGA or on a ASIC, then you will have to use some kind of synthesis tool. Simulation can only be a first step to carry out initial verification of a design, but when you are doing synthesis, you may see that a lot means other problems are cropping up. There are some errors which I showing, which were not reflected during the simulation phase.

So, in this lecture we shall be talking about some of the features of the Verilog languages that are meant to be used for synthesis, the other features which we have discussed in the class, but they are often not accepted by a synthesis tool, if you use those constructs the synthesis tool will not be able to generate the final hardware circuit or the net list, ok.

(Refer Slide Time: 02:18)

About the Verilog Language

- The language Verilog has a large number of features, most of which are supported by the simulation tools.
- Unfortunately, several of the language constructs are not supported by synthesis tools.
 - The language subset that can be synthesized is known as "*Synthesizable Verilog*" subset.
- Here we shall state the language features not supported by most of the synthesis tools.
 - Best be avoided if the objective is to map the design to hardware.

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

So, the title of today's talk is synthesizable Verilog. So, whatever I had just talked about just now. So, the Verilog language provides you with the number of facilities and features and this features are mostly supported by the simulation tool. For example, the simulation tool that we have been using a part of this course the Iverilog and of course, the waveform viewer gtk wave, they are all known to support most of the Verilog constructs that we have been discussing throughout the class, right. But as I had said there are some language constructs which are not accepted by the synthesis tools, this subset of the language Verilog which are actually accepted by the synthesis tools are referred to as synthesizable Verilog subset.

Now, in this lecture we shall be looking at some of the language features which are not supported and some of the recommended styles of modeling for synthesis of circuits both combinational and sequential.

(Refer Slide Time: 03:36)

Synthesis Rules for Combinational Logic

- The output of a combinational logic circuit at time t should depend only upon the inputs applied at time t .
- Rules to be followed:
 - Avoid technology dependent modeling (i.e. implement functionality, not timing).
 - There must not be any feedback in the combinational circuit.
 - For “*if...else*” or “*case*” constructs, the output of the combinational function must be specified for all possible input cases.
 - If the rules are not followed, the circuit may be synthesized as sequential.

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

So, talking about the synthesis rules for combinational logic, you recall what do mean by a combinational logic, a combinational logic is a hardware circuit which does not have any kind of enough fan outs or storage devices built into it.

So, the output of the circuit will only depend on the currently applied inputs, well of course the circuit, the gates will have some delays, the output will be available just after the circuit delays. There is no notion of clock and other kind of synchronization that comes in case of combinational circuits

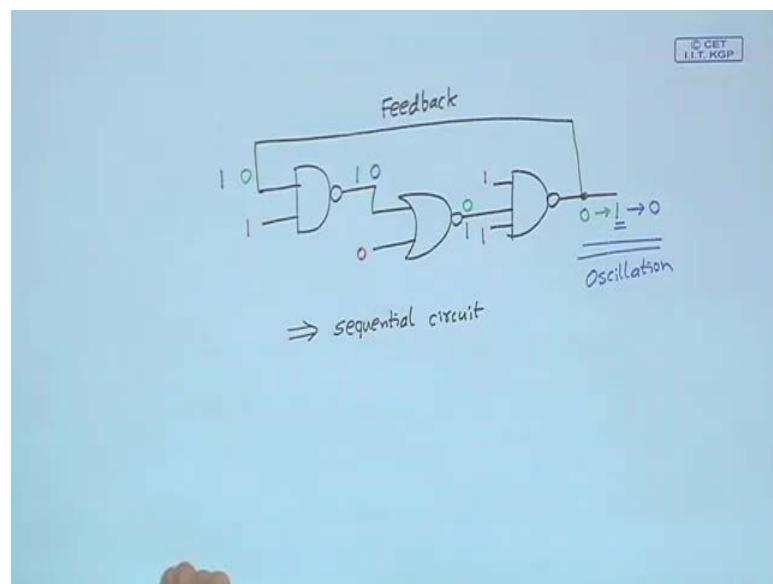
So, you can say that the output of a combinational circuit at some point in time let us say t , should depend only upon the inputs that I have applied at that time, of course there will be a delay of the circuit that we are not mentioning here, well. So, we were trying to model combinational logic, there are some rules that need to be followed, first thing is that since you are talking about combinational logic do not use, I means, any kind of technology dependent modeling, means specifically some details about timings, see the timing detail using that hash (#) command.

So, whatever you give during simulation that is just meant for carrying out the simulation and interpreting the waveforms that you are viewing after the process of simulation is over. But when you are synthesizing you are final delays will be dependent on the actual hardware. So, where your mapping it can be a ASIC or a FPGA. So, you do not have any control about the delay, you cannot set the delays 1 or 2 or 3 nanosecond or

picosecond, you cannot say that, that will be entirely depend on the hardware where you are finally going to map your design tool, ok.

So, this kind of technology dependent modeling are not allowed when you are trying to synthesize combinational circuits. Secondly, of course, in a combinational circuit there are no feedback. So, your net list or your circuit must not have any feedback, feedback means some connection from the output of a circuit to the input of a circuit.

(Refer Slide Time: 06:20)



Let us take an example, suppose I have a circuit like this, there are other inputs, suppose what I say is that this output line is connected to one of the inputs, this is what you mean by feedback. Well feedback is not allowed in a combinational circuit for obvious reasons because when your feedback, your circuit may turn in to a sequential circuit, right.

So, in this example let us say we have applied a 1 here, we have applied a 0 here and a 1 and 1 here. So, now, what will happen, what will be the behavior of the circuit? Let suppose that initially the output of this NAND gate is 0. So, 0 is being feedback, this is 0; 0 and 1, NAND will be 1; 1 and 0, NOR will be 0; 0, 1 and 1, NAND gate, so, the output will be changing to 1, right.

Now, when the output is changing to 1, this 1 will again be feedback, now the new input will be 1; 1 and 1 will be now 0; 0 and 0 will be now 1; 1, 1, 1 will be again 0. So, there

will be something called oscillation in the output, it will continuously go from 0 to 1, again to 0, again to 1, again to 0. So, the output will never stabilize.

So, if your object is to design a combinational circuit, you must avoid feedback connections like this, there must not be any kind of feedback in your net list or circuit, right. And the third point I mentioned repeatedly earlier that whenever you are having a some kind of multi way branching. So, either using if-else or using a case construct, while we are checking for some condition and then you are deciding on something some assignments.

So, here the output of the function whatever you are assigning to must be specified for all possible input cases, like for instance, if you are checking for a condition which is a 2-bit variable then you must specify what the output will be for all 4 combinations of those 2-bit variables for 00, 01, 10 and also 11.

So, we discussed earlier that if you forget to specify one of the input combinations then the synthesis tool will be generating a latch for the output. So, even if you are trying to generate a sequential circuit. So, whatever will get generated will be, sorry, means actually you are trying to generate a combinational circuit, but whatever will be generated that will be a sequential circuit with a storage element. So, you must avoid this kind of incomplete specification in multi wave branching.

(Refer Slide Time: 09:56)

Styles for Synthesizable Combinational Logic

- The possible styles for modeling combinational logic are as follows:
 - Netlist of Verilog built-in primitives like gate instances (AND, OR, NAND, etc.).
 - Combinational UDP (*not all synthesis tools support this*). ↴
 - Continuous assignments.
 - Functions.
 - Behavioral statements.
 - Tasks without event or delay control.
 - Interconnected modules of one or more of the above.

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

So, as I said if you do not follow these rules, the circuit may become sequential. Here are some styles that you need to follow for the synthesis of combinational logic, some of this we have already seen, some of this we have not talked about earlier, we shall just look at them through some examples like the possible styles maybe. Firstly, we have seen several examples. So, you can instantiate the basic primitives of gates AND, OR, NAND, XOR and so on and you can create an net list like that. Secondly, you can use user defined primitive for combinational circuit, you can define a truth table, right. Most of the synthesis tool support combinational UDPs, but you need to check there. There are some synthesis tools which will not accept any kind of UDPs at all even if it is a combinational UDP with a truth table, ok

So, that is mentioned, not all synthesis tools will support this feature then continuous assignment using assign statements this of course, you can use for modeling combinational logic and functions are something which I have not talked about earlier. These functions are something where you can use this continuous assignment statements in conjunction with the functions, we will see. We will see that there are some advantages or the code becomes much easier and more structured.

Then of course, we can use the always block for modeling combinational circuit, we have seen many examples here, behavioral statements and just like functions, there is another thing called tasks. So, we can use tasks as well, but we cannot specify any kind of delays and you can use a net list of the modules created using one or more of these. You can combine any number of them, you can create a some kind of interconnection of them by using instantiation.

(Refer Slide Time: 12:15)

(a) As a Gate Netlist

```
module example (x1, x2, x3, x4, y);  
    input x1, x2, x3, x4;  
    output y;  
    wire w1, w2, w3;  
    or (w1, x1, x2);  
    or (w2, x3, x4);  
    xor (w3, x3, x4);  
    nand (y, w1, w2, w3);  
endmodule
```

Shall be synthesized in terms of gates from some target technology library.
The gate netlist is often optimized during synthesis.

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

Let us look at these examples, some of this one by one. So, as a gate net list, so, we have seen many examples earlier, this is a very simple example, there are 5 parameters, these 4 are the inputs and y is the output and this w1, w2, w3 are some intermediate wires. So, here you are specifying that there are 4 gates or; x1, x2 is input; w1 is the output; this is second or, xor and nand. So, in this way you can create any kind of net list.

Now, the point to note is that here we are directly specifying the gate types, but this synthesis tool when it is synthesizing, it will be generating the final hardware, right. So, the final hardware will be generated based on some target library. Let say for example, the target library does not contain any exclusive or gate. So, the xor gate which you have specified here, they will have to be generated using different kinds of gates, maybe a combinational nand gates or and or not gates and so on. So, during the synthesis process whatever gates you have specified, there can be some changes as well because you will have to use gates only from the target technology library that is called; that means, the target gates which are supported by the hardware. So, during the process the synthesis tool also carries out some kind of optimization, ok.

So, these are done here. So, when you are carrying out this mapping, some of the gate level minimization algorithms can be used.

(Refer Slide Time: 14:16)

The slide has a yellow header bar with the title '(b) Using Continuous Assignment'. Below the title is a Verilog module definition:

```
module carry (cout, a, b, c);
    input a, b, c;
    output cout;
    assign cout = (a & b) | (b & c) | (c & a);
endmodule
```

To the right of the code, there are two explanatory boxes:

- A red box states: "Shall be mapped to gates or cells from some target technology library."
- A blue box states: "The Boolean equations are optimized during synthesis."

At the bottom of the slide, there is footer information: IIT KHARAGPUR logo, NPTEL ONLINE CERTIFICATION COURSES logo, and the course title "Hardware Modeling Using Verilog".

Next, continuous assignment is something which also we have seen through many examples, here we can use an assign statement, to assign some Boolean expression to a variable which is of type wire. Now, this is a very simple example, a module which computes the carry output (cout) of a full adder, so it is ab or bc or ca. right.

So, here also we have not specified any gates, here we are specifying just the function. So, the synthesis tool again will take the functional specification as the input and will be trying to generate some gates or cells from the target technology library after some kind of minimization or a optimization, this is what will be done if we use the assignment statement.

(Refer Slide Time: 15:16)

The screenshot shows a slide titled '(c) Using Procedural Blocking Assignment'. It contains a Verilog module definition for a 2-to-1 multiplexer. The module has three inputs: `in0`, `in1`, and `sel`, and one output `f`. The logic is implemented using an `always` block with a procedural assignment. A callout box highlights the requirement to include all inputs in the event control expression to avoid inferred latches.

```
module mux2to1 (f, in0, in1, sel);
    input in0, in1, sel;
    output reg f;

    always @(in0 or in1 or sel)
        if (sel) f = in1;
        else      f = in0;
endmodule
```

Inputs to the behavior (*here* `in0, in1, sel`) must be included in the event control expression; otherwise, a latch will be inferred at the output.

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

Similarly, you can model combinational circuits using procedural block assignment where the event is not an edge triggered event. This is an example of a simple, example of a 2 line to 1 line multiplexer, where `in0` and `in1` are the inputs, `sel` is the select line and `f` is the output. So, these 3 are the inputs and `f` is the output, which is also `reg` because you are assigning `f` within a procedure block.

So, here we are saying, always whenever any of the inputs are changing, if the select line is 1 then `in1` is selected to `f` otherwise `in0` is selected to `f`. So, the point to notice that when you are specifying a combinational logic like this your activity list must contain all the inputs, in this case the inputs are `in0`, `in1` and `sel`. So, all the inputs of the behavior which are used in this procedural block must be included in the event control because if you do not do it, this synthesis tool will assume that well those are not the inputs, but still you are using it. So, if the inputs are not specified, you do not know, you may have to use a latch to store the output `f`. So, latch will often be inferred by the selection tool if you give incomplete input list specification. So, make sure that you give a complete input list specification in this `always` block, ok.

(Refer Slide Time: 16:53)

(d) Using Functions in Verilog

```
module fulladder (s, cout, a, b, cin);
    input a, b, cin;
    output s, cout;
    assign s = sum(a, b, cin);
    assign cout = carry(a, b, cin);
endmodule
```

```
function sum;
    input x, y, z;
    begin
        sum = x ^ y ^ z;
    end

```

```
function carry;
    input x, y, z;
    begin
        carry = (x&y) | (y&z) | (z|x);
    end

```

A function in Verilog returns a single value.
Can be used to make a code more readable.
Typically used with "assign".
Input arguments appear in same order.



IIT KHARAGPUR



NPTEL ONLINE
CERTIFICATION COURSES

Hardware Modeling Using Verilog



Now, let us see how functions work, we have not talked about functions earlier. See function is very similar to a function in high level language like C; C which means in a language like C, you can have functions. Now, now in a function you can pass a number of variables through the parameters, but you can pass a single result through the name of the function. So, here also whenever you are using a function, it is intended to have a single output, a single bit output. So, it cannot have more than one outputs, this is a restriction of function in Verilog.

So, let us say here we are trying to write a module for a full adder, just using assign statement, but instead of directly writing down the expression, we are calling 2 function sum and carry, these 2 functions we have written separately. Function, the syntax is like this, function followed by the name, sum is the name. Then I will have to specify a list of the inputs and they have to be provided in the same order you are mentioning them here a, b and cin. So, a will be map to x; b will be map y and cin will be map to z. And then here you are giving an assignment statement, for the left hand side is a name of the function and right hand side is any expression based on these input variables. This is the sum function, similarly this is the carry function.

So, you can directly call this functions in the main full adder module, what is the advantage is that your module becomes easily understandable, it is much more well documented. So, instead of writing the expressions here this module will become more

clumsy rather than if you write like this, it will be much clearer. So, as I had said, so function in Verilog will return a single value here sum, here carry and the value will be return against this expression, whenever you are writing an expression on the right hand side. So, whatever is the calculated value of sum that will be assigned to s, similarly for the carry function, the value will be assign to cout. So, this makes the code more readable as I said and these are typically used with assigned as this example illustrates and also I have shown that the input arguments must appear in the same order in the function and the place where you are calling them.

(Refer Slide Time: 19:46)

(e) Using Tasks

The arguments must be specified in the same order as they appear in the task declaration.
More than one output value can be returned.

Now, let us come to tasks, well a task is more general before going on the example. Let us see what a task is, the first thing about task is that using task; task is like a function, but you can pass more than one output value to the calling program or the module. Like you see an example and you see for function you can use it only inside assign as I had said, but a task can be used anywhere; task can be used anywhere inside a Verilog function. You write a task and you can call it from someplace.

So, wherever you have defined the task that will get substituted in the place where your calling with proper argument passing. Let us take this example, so, this is again a full adder example. So, what you are saying, we are using the behavioral modeling, you see always @(a or b or cin), whenever the input change, well instead of writing the sum and carry expressions, we are calling a task.

Now, FA is a task, we have defined FA within bracket the list of parameters, s, cout, a, b, cin. So, in this list of parameters, 3 of them are inputs and 2 of them are outputs, so, s and cout will be returned. Now the way task is defined is like this task name and end task and inside this we will have to specify the outputs and the inputs in the same order as they appear here. Let s and cout are the 2 outputs, you specify them maybe the names will be different here, we call it them sum and carry then a, b, cin, they are the inputs. So, here you call them a, b, c. Now in task you see, you can also specify a delay which you cannot in a function. So, sum expression, carry expression. So, this is how we can use a task in a function.

Now, in our earlier examples which we have seen, we have not given any example that uses function or task, but if you want for a larger designs, you can use functions and tasks as well, fine. Now let us look at the difference between function and a task. So, what are the main difference and where you can use a function and where you can use a task, right.

(Refer Slide Time: 22:33)

Difference between Function and Task	
Function	Task
A function can call another function but not another task.	A task can call other tasks and functions.
A function executes in 0 simulation time.	A task may execute in nonzero simulation time.
A function cannot contain any delay, event, or timing control statement.	A task can contain delay, event, or timing control statements.
A function always return a single value.	A task can pass multiple values through "output" and "inout" type arguments.
A function must have at least one input argument.	A task can have zero or more arguments of type "input", "output", or "inout".



IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Hardware Modeling Using Verilog

So, let us see this table. So, a function, first thing is that, so, a function can call another function, but a function cannot call a task. So, you can have multiple functions and you can have nested calls, a function x can call a function y, y can call a function z, just like in a high level language we do. This is how function calls are made, but tasks are more general. So, a task can either call a function or it can call some other task.

Now, in a function the concept of delay is not there. So, it is assumed that a function will execute in time 0.

But, in the task, in the example that I have shown it also shows there that you can also include some delays, so, it can execute in non-zero simulation time. And a function just usually contains only combinational assignments nothing else, but in a task it can be more general, you can contain delay like the example showed, you can contain some event triggering or some timing control statements as well. The function returns a single value this is another thing and a task it may return more than one value through output arguments, like in the example that I have showed earlier, there were 2 output arguments sum and carry.

Now, another kind of port declaration is there which I have deliberately not talked about, I shall take some examples later these are some kind of bidirectional inputs “inout”. Well inout means you can use that variable as an input and also as an output, but when you map it to the hardware, sometimes some problems are created for inout type variables that is why in the designs that I have shown I have avoided inout, fine. And another thing is that a function must have at least one input argument, but there is no such restriction in task, it can have zero or more arguments and the arguments can be of type either input or output or inout.

(Refer Slide Time: 25:01)

Constructs to Avoid for Combinational Synthesis

- Edge-dependent event control.
- Combinational feedback loops.
- Procedural or continuous assignment containing event or delay control.
- Procedural loops with timing.
- Data dependent loops.
- Sequential user defined primitives (UDPs).
- Other miscellaneous constructs like “*fork ... join*”, “*wait*”, “*disable*”, etc.

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

Now, these are some of the constructs which you must avoid when your objective is to carry out combinational circuit synthesis, like we should not give this edge dependent even control, like you must not give means always at posedge clock or negedge clock, no kind of posedge or negedge kind of event control should be there in a combinational circuit description.

Then second thing I already mentioned there must not be any feedback loops. So, when you specify net list using instantiations make sure that you do not include any feedback loops. Third thing is that you can have procedural assignment or continuous assignment, but there must not be any delay specification or event control, because in synthesis those are not considered.

Similarly, you can have a loop with timing, there can be some delay specified, such delays are not allowed. Then data dependent loops, data dependent loop means some kind of a loop where the number of times you are going to loop, it depends on a variable, that kind of thing is not allowed. The number of times you are looping must be a constant. So, if it is a constant, suppose 3, I specify, I want to execute a for loop 3 times then the synthesis tool will make 3 copies of whatever is the body of the loop, it will make 3 copies of the circuit and it will create a combinational circuit like that, there will be 3 levels which corresponds to the 3 iterations of the loop, right.

Sequential user defined primitives like state table or not permitted by synthesis tools and there are some other miscellaneous constructs which I have not discussed deliberately like “fork join”. These are used to specify concurrency “wait”, well wait is used sometimes to make a statement wait for certain time before it gets activated and of course “disable”, you can disable some signals are commands.

(Refer Slide Time: 27:27)

The slide has a dark blue header and footer. The main content area is yellow. It features a title 'Summary: Synthesizable Verilog Constructs' and a bulleted list of constructs.

Summary: Synthesizable Verilog Constructs

- "*module ... endmodule*"
- Instantiation of a synthesizable module
- "*always*" construct
- "*assign*" statements
- Built-in gate primitives
- User defined primitives – combinational only
- "*parameter*" statement
- "*functions*" and "*tasks*"
- "*for*" loop
- Almost all operators
- Blocking and non-blocking assignments
- "*if ... else*", "*case*", "*casex*" and "*casez*"
- Bits and part select of vectors

At the bottom, there are logos for IIT Kharagpur and NPTEL, followed by the text "NPTEL ONLINE CERTIFICATION COURSES" and "Hardware Modeling Using Verilog".

So, to summaries this synthesizable Verilog constructs, the construct that you must use when you objective is to synthesize the design in to some hardware, there as follows module, endmodule of course. Well you are allowed to carry out instantiation, you can use always constructs, assign statements, built in gate primitives like and or not xor, nand, nor this you can use. But for user defined primitives, you can possibly use only combination specification, but still you will have to check with the synthesis tool your using that whether the tool supports combination UDP or not. Parameters, you can use functions and tasks as I have mentioned as sometime back that can be used; for loop is usually supported by all synthesis tools, the other kind of loops may not be supported.

So, again we will have to specifically check whether the other kind of loops like while or repeat they are supported or not. Regarding operators, almost all the operators are supported by barring a couple of them, a very few blocking and non-blocking assignments are supported and multi way branching using if-else, case, casex, casez. So, you can use bits and with respect to vectors, the part selection you can select some segments from a vector those are supported, ok.

So, if you stick to these rules that mean you will not be using anything outside this then it is often guaranteed that whatever module that you write that can be synthesized into a hardware by the synthesis tool.

(Refer Slide Time: 29:22)

Non-Synthesizable Constructs

- “*initial*” construct
- Delays in assignments and test benches.
- “*time*” construct
- “*real*” data type
- The operators “*==*” and “*!=*”
- “*fork ... join*” constructs
- “*force ... release*” constructs
- Variables in loop control

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

And there are a few things which are not synthesizable definitely like initial construct it is meant only to be used in the test benches, delay specification is in the hash command.

So, you cannot use delays in synthesizable code; time, no concept of time in synthesizable code; real data type is often not supported, only bits and integers they are supported and the operators, this triple equal and not equal to equal to, these two are not supported; fork join; force release; and as I had said in loop control, you cannot loop variable number of time. So, variables in loop control are typically not supported.

So, with this we come to the end of this lecture. So, in this lecture we try to give you just an idea that what are the constructs in Verilog that you have learned so far. Which are useful for synthesis and what are the constructs which are best avoided because most of the synthesis tools do not support that.

Now, in the next lecture we shall be continuing with our discussion, we shall be talking about some recommended practices some dos and dont's. You see Verilog maybe supporting a few things. So, the synthesis tool may be supporting many things, but if you go to an industry who is specialized or whose specializes in design, they will give you some guidelines, the designer will have to stick to those guidelines whenever they are designing any circuit using some language like Verilog, because if they do that then the codes will be well documented, they can be reused and they can be maintainable.

Thank you.

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

Lecture - 29
Some Recommended Practices

So, in this lecture we shall be discussing some of the recommended practices that are typically followed in an industry which works in the area of digital design. So, it is good to learn this, because some of you who may be going to the industry and work in the design area in the future. So, you will be encountering these kinds of guidelines there. So, every industry they have a set of guideline, where they specify that what are the things that you should do, and what are the things you should avoid when you are creating a design, using some hardware description language, ok.

So, let us see some of these recommended practices.

(Refer Slide Time: 01:04)

The slide has a yellow header bar with the title 'Naming Conventions'. Below the header is a white content area containing a bulleted list of naming conventions. At the bottom of the slide is a blue footer bar with the IIT Kharagpur logo, the text 'NPTEL ONLINE CERTIFICATION COURSES', and the course name 'Hardware Modeling Using Verilog'.

- File Naming
 - A file must contain only one design unit, contained in a single "*module ... endmodule*" construct.
 - All Verilog files must have an extension of ".v".
- Naming of variables, signals and other objects
 - Names must be composed of alphanumeric characters or underscores.
 - Names must start with a letter, and not a number or underscore.
 - All names must be unique irrespective of case.
 - Parameters and constant names must be given in UPPER CASE (e.g. *PI*, *DELAY*, etc.).
 - Signals and variable names must be in lower case, and must be meaningful names.
 - A constant name must describe the purpose of the constant (e.g. *reg_a_enable*).

First thing concerns the naming conventions. These concerns file naming. Now see Verilog compiler, simulator or synthesizer whatever you call, they support filenames, many of them they support file names, which can have arbitrary extensions. Not necessarily dot v or dot Verilog or anything, you can have any extension dot anything.

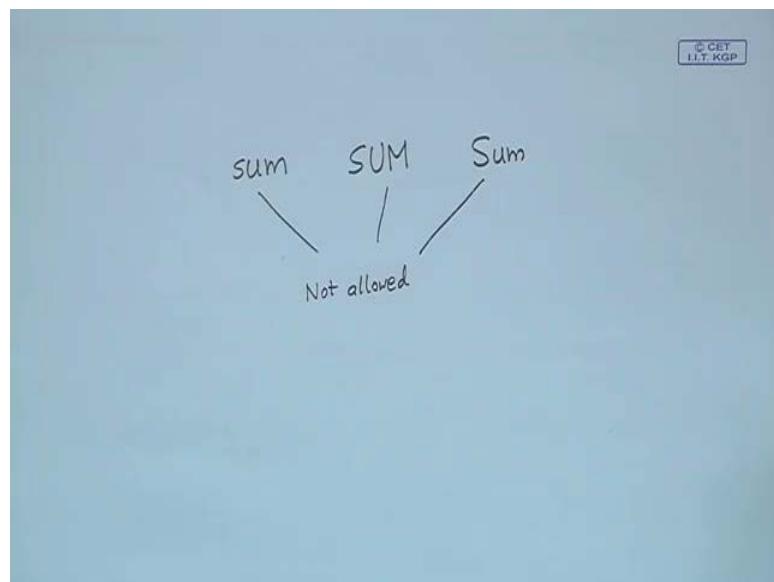
But let us say here we are specifying a set of typical conventions. So, here we are saying that all the files must have an extension of dot v (.v), and we must not include more than one module in a file, this is one rule that you should follow.

A file must contain only one design unit. You see this does not mean that you cannot write more than one modules in a file. Yes, you can do, you can write 10 modules in a file, you can synthesize, you can simulate, but these are some of the design practices, which is practiced in typical industry houses, ok.

Regarding the naming's of the variables signals or any other object, the conventions are as follows, names should be composed of alphanumeric characters or underscore. Alphanumeric means alphabets, numerics and underscores. Some Verilog synthesis tools or simulators, they support other, few other characters also, but this guideline says that you should avoid those, stick to alphanumeric and underscores only, right. And again even if the compiler supports, so, every name must start with a letter, you should not begin a name with an underscore.

Third point says the names must be unique irrespective of case. So, what does this mean. This means Verilog is case sensitive, where uppercase and lowercase letters are considered to be different. So, you must not do something like this.

(Refer Slide Time: 03:40)



Let us say you define one variable as lowercase sum, define another variable as uppercase SUM, and you define another variable which is a combination of this. So, this guideline says that this is not allowed. So, again this is not allowed, it does not mean not allowed by the compiler, it is not allowed by the company, something like this, right.

So, some other conventions, it says parameter and constant names. So, whenever you are defining some constants. So, either by using some defined statement or using parameters we have already seen, the names must be given in uppercase. For example, PI, DELAY, so, that by just seeing a variable name you will understand that whether it refers to a normal variable, reg type or wire type, or it is a constant or a parameter, just by seeing the variable you can understand.

Similarly, signals and variable names, they will be in lowercase. Not only that names should be meaningful, this is quite obvious. Instead of giving names like a, b, c, d, e, f, you should give names like carry, sum, out like that. So, that the names are somewhat meaningful in the context of the design that you are trying to create.

Then it says that a constant name must describe the purpose of the constant, whenever you are defining some constant or a variable name, like see reg a underscore enable something like that this can be very well, because a constant I have said that it will be uppercase. So, it can be anything uppercase or lowercase, but it should be meaningful.

(Refer Slide Time: 05:43)

The slide contains the following list of guidelines:

- Construct names such as modules, functions and tasks must also be meaningful.
- For names composed of several words, underscore must be used (e.g. `load_input`).
- When a signal uses active low polarity, it must use the suffix `_b` (e.g. `clear_b`).
- Signals that are used for clocking that do not have the word `clock` or `clk` already in their names, must use the suffix `_clk` (e.g. `bus_clk`).
- Unrelated signals must not be bundled into buses.

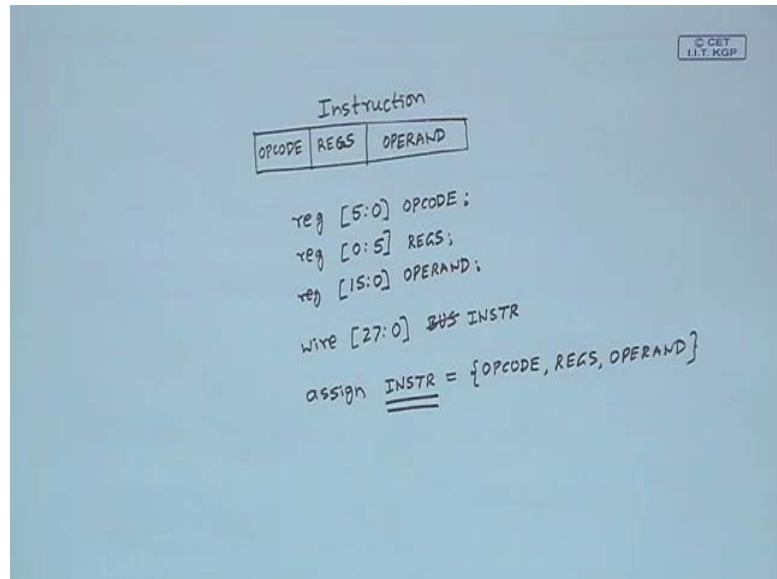
So, this repeats not only the variables, names of the modules, the functions and the tasks they also must be meaningful. So, that just by seeing that module or function or task, you can have a rough idea what it is trying to do, right. So, if a name is a little long, composed of several words, we can use underscores, like this load_input, this convention can be followed.

Now, see signals can be either active high or active low. let us take an example, suppose I have a register, there is a clear input. So, active high means, whenever clear is 1, the register will be cleared. Active low means whenever that clear is 0, the register be cleared. So, in a design I can use many such signals; some of them can be active high, some of them can be active low. Just as a matter of convention in order to identify which of the signals are active low, this convention says that you use a suffix underscore b for the active low polarity signals, like clear_b. This will tell you that clear is a active low signal, b is a short form for below, ok.

Signals that represent clocks, typically we give names like clk or clock, but there can be some other signals also. They represent clocks or they are derived from the clock, using something called clock gating. Like you can take a clock signal, you can take some other signal line you can take an AND gate, and output will be a so called gated clock, that gated clock can also be used to control or activate some storage cells or registers, ok.

So, such signals you should give an explicit name by suffixing it underscore clk, like if the name of the signal is bus, you write bus_clk, indicating that this is also a clock signal. And the last point is quiet obvious, unrelated signals must not be bundled into buses. Like let us say, let us take an example.

(Refer Slide Time: 08:15)



Suppose I have a word that indicates an instruction in a processor. So, in this instruction, I can have an OPCODE. So, I can have some fields that indicate REGS, REGS and some fields that indicate OPERAND. So, I can, what I can do, I can declare them separately like if OPCODE is a 6-bit quantity (reg [5:0] opcode), I can declare them like this. If REGS is a, let us say, this is our 6-bit. See here I am writing different way just to show you that I can do this also, register and this OPERAND, let us say this is a 16-bit quantity let us say.

Now, what I am saying is that, you can specify a bus, let us say this is how much 5, 6 and 6, 12, and 16, 28. So, let us have a 28-bit bus, let us call it bus. So, I can always give a assign statement like this, bus or you can give any other meaningful. Instead of bus let us say let us give a name INSTR, this will be more meaningful. So, assign instruction equal to composition of OPCODE, REGS, OPERAND (assign INSTR = {OPCODE, REGS, OPERAND}). This means 3 different things, and grouping them together and I am referring it to a single entity this INSTR.

Here in this point, means exactly the same thing is mentioned. It says that whenever you are grouping like this, the signals you are grouping must be related. Like here OPCODE, REGS and OPERAND, they are all part of the same instruction; that is why you are grouping them, but unrelated signals you should never group, fine.

(Refer Slide Time: 10:44)

The slide has a dark blue header and a light yellow body. At the top center, it says 'Comments'. Below that is a bulleted list of requirements:

- Comments are required to describe the functionality of a design unit.
 - Each file must contain a file header, that follows some convention.
 - The header must include the name of the file.
 - The header must include the highest level construct contained in the file.
 - Every file header must include the originating section or department, author, and author's email address.
 - Header must include release history whenever such a change is registered.
 - The header must contain a purpose section explaining the functionality of the module.
 - The header must contain information describing the parameters being used in the construct.
 - The number of clock domains and clocking strategy must be documented.
 - Critical timing including external timing relationships must be documented.

At the bottom of the slide, there are three logos: IIT Kharagpur, NPTEL Online Certification Courses, and Hardware Modeling Using Verilog.

Next this is very important in a design comments. So, if you write a Verilog code which is functionally correct, everything is fine, but there are no comments, then after sometime you will see that, you yourself will not be able to understand that code, you forget about others. So, it is mandatory to have every piece of code very well documented, and that is done using comments.

Now, some of the conventions regarding comments are as follows, is at every file you create must contain a header, that contains lot of information, and every company has its own convention. I will give an example later. So, the header among other things must also include the name of the file; such that accidentally the name should not get changed, right.

So, it should also contain the highest level construct contain in a file, because you see, just in Verilog, you can create a design in a hierarchical way. Like we took an example of an adder, a ripple carry adder built using full adder, a full adder built using sum and carry and so on. So, that header should also specify what is the level of hierarchy that module or that file is specifying or referring to, fine.

And of course, it must contain some descriptive note that what the module is actually doing. So, explaining the functionality of the module, and the parameters used in the module, what are the functions of the parameters. And if there are multiple clock domains, you are using multiple clocks, may be a slow clock or fast clock, then the

header must also specify. I mean how many clock domains are used and what kind of clocking strategies used, some may be leading edge triggered, some may be falling edge triggered and some issues regarding critical timing may also be documented.

Because say for example, when you are designing a module, maybe you know that there is 1 signal, which falls in the critical path. If you make that signal generation slower, your whole system will slow down. So, let that remain documented in the header. So, that in the future when someone updates or modifies that module, he will know that well this is a very important signal, I must not make that signal slower; that is a critical signal, fine.

(Refer Slide Time: 13:37)

The slide has a yellow header bar with the title "Coding Style". Below it is a white content area containing a bulleted list of coding style guidelines. At the bottom of the slide, there is a footer bar with three sections: IIT Kharagpur logo, NPTEL Online Certification Courses logo, and the course title "Hardware Modeling Using Verilog". On the right side of the footer bar is a circular portrait of a man.

- Good coding style helps in easy understanding of the code and maintainability.
 - Write code with proper indentation in a tabular format.
 - A constant indentation of *2 to 4 spaces* must be used for code alignment; do not use tab stops (tab stops interpretation varies from system to system).
 - Use spaces and empty lines to increase the readability of code.
 - One line must not contain more than one Verilog statement.
 - Use one line comments using “//”; avoid multi-line comments using “/* ... */”.
 - Keep line length less than 80 characters, so as to avoid line wraps.
 - When declaring ports, declare *one port per line*. Descriptive comment must follow each port listing.

Some of the coding style should also be followed, this will of course, make the code more easily understandable and it will be easier to maintain by others mostly. Firstly, is that you must give indentation in the code; indentation means proper spacing in a proper tabular format. So, again I will give an example, we see earlier all the examples that I have shown their indentation was used. So, I never showed you all the instructions starting from the same column, there will be some spaces, again some spaces, again some spaces.

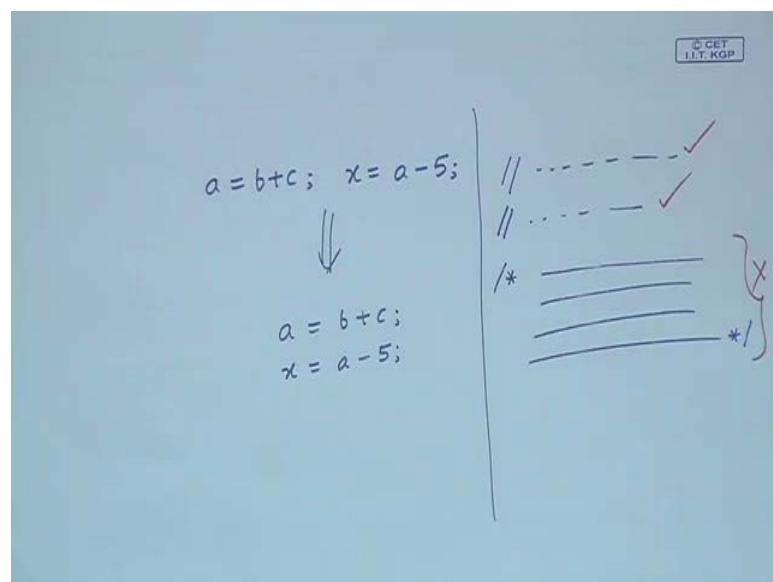
So, just the position of the statement will tell you that under which block that statement is belonging to. So, indentation gives you a very nice visual structure of your whole code, right. Not only indentations, well this indentations are typically, this guideline says

you must give 2 to 4 spaces for indentation, well, and there is also a recommendation, it says do not use tabs for giving indentations.

Well many of us have the habit of giving the tab by pressing the tab, some space is obtained to create the spaces, but the problem with tab is that they are differently handled in different systems. In some of the system a tab may be equivalent to 4 spaces, in some other system it may be equivalent to 6 spaces and so on. So, as you move your code across machines your structure of your code may look different, ok.

And there is another thing here, it says one line must not contain more than one Verilog statements. Now see earlier in many of the examples, because of the lack of space on the slides.

(Refer Slide Time: 15:43)



Well I gave some examples like this let us say, $a = b + c$, $x = a - 5$. So, two statements were given on the same line, but this says that you must not do it, this $a = b + c$ should be in one line and $x = a - 5$ will be in the other line. So, every line must contain one Verilog statement.

The next one is very interesting, it says that well you can give comments in 2 ways; either you can give double slash ('//'), which means your entire line is comment or you can use slash star ('/*') then comments star slash ('*/'). This may be a multiline comment, means say in a code if you give a slash, slash in a line then the whole line will become a

comment. So, again you give a slash slash, this line will become comment, but if you give a slash followed by a star, then this comment will continue across several lines, then you can give a star slash at the end.

So, what this rule says that to avoid the second kind of comment, you better use this comment, this style, right, because at the beginning you will know from the beginning that, well this is a comment line, but here say you are not seeing this, you cannot just identify whether this is a comment line or this is part of your code, right, that confusion maybe there.

So, just use only a one line comment using double slash. And just every line of the code must be within eighty characters, so that the line may not spill to the next line. And again another rule says, whenever you are declaring ports like your declaring a module, whose parameters are a, b, c typically, you specify a, b, c on the same line as in the example that I have shown. But this rule says you must specify them on 3 different lines, a on one line, b on second line, c on third line, maybe the size or the number of lines of your code will be increasing, but your structure will be very easily visible. You can see that there are 3 parameters on 3 lines, a, b and c, it will clearly visible, fine.

(Refer Slide Time: 18:14)

The slide has a yellow header bar with the title "Module Partitioning". Below the title is a bulleted list of guidelines for module partitioning:

- Used for reducing complexity, and also to minimize the chances of error.
 - Procedure, tasks and functions must not access or modify signals / variables not passed as parameter to the module.
 - If we use a gated clock, internally generated clock, or use both edges of a clock, the clock generation circuitry must be kept in a separate module at the top level or at the same logical level in the hierarchy as the block to which the clocks apply.
 - Separate clock domains (e.g. *slow clock* and *fast clock*) must be partitioned into separate blocks.
 - The design should be partitioned so as to minimize the number of interface signals.
 - Do not mix structural and behavioral RTL code within a construct.
 - State machines and asynchronous logic must be partitioned in a separate block.

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the text "NPTEL ONLINE CERTIFICATION COURSES", and the course name "Hardware Modeling Using Verilog".

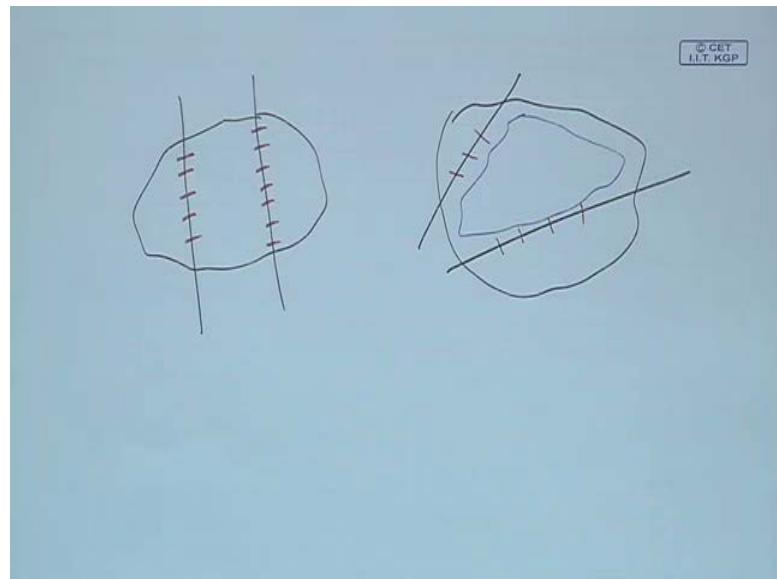
Then of course, module partitioning is very quiet obvious. So, usually for complex modules you can divide or split the module into smaller modules, and instantiate them into a top level module; that is normally the design style we form, this is called module

partitioning, and this helps in reducing the complexity of design, and because you are handling smaller modules at one time the chances of errors will also reduce.

Now, some of the rules regarding module partition says that procedure, task, functions whatever it is, they must not access or modify signals, not passed as parameter to the module. See Verilog allows you to access some variables even outside the scope of that task or function, but here the rule says you must never do that, you only access variables that are passed to the function, and not other variables. And if you are using some gated clock or internally generated clock, or if you are using multiple edges of the clock, then the clock generation circuitry must be separated out in a separate module, this is what this rule says.

Similarly, if you have multiple clock domains like in a design, there is a slow clock and also a fast clock, they must be defined in separate modules, separate blocks, this is important see given a design.

(Refer Slide Time: 20:01)



Say our objective is to partition it to smaller design. Now this same design it is a, I can also partition it like this. Now which of them is better, see you will have to find out how many signals are crossing this partitions. The partition where the number of signals crossing a less that will be considered to be a better partition, because your number of parameters to the modules will be less. So, your design will be much more structured and naturally with this part of your design will contain most of the self contained code,

because they do not have too many interactions with others, right. So, this is one objective of partitioning a larger design. So, the design should be partitioned.

So, as to minimize the number of interface signals. And structural and behavioral code must not be mixed at the same place. Similarly, state machines like FSMs which rely on a clock and asynchronous logic which rely on feedback connections and no clock, they must not again be mixed, they must be put in separate blocks or separate modules, right.

(Refer Slide Time: 21:29)

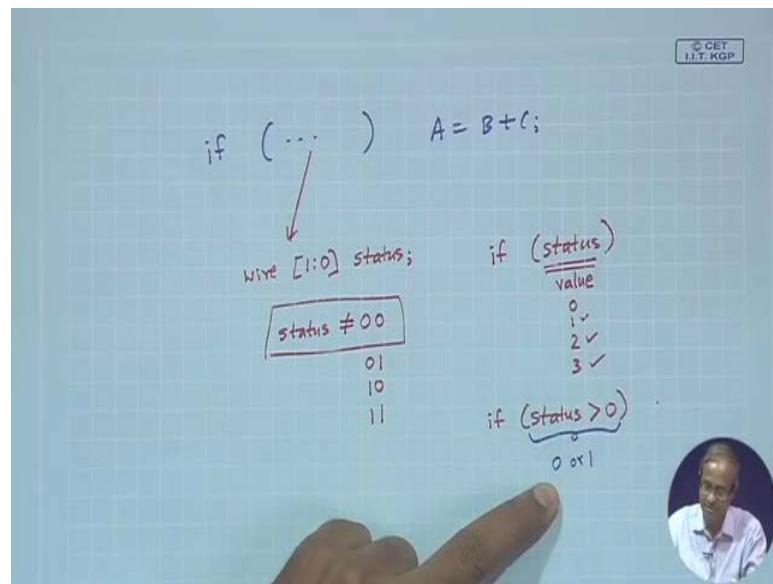
General Coding Techniques

- Some of the general guidelines for coding are as follows.
 - The expression in a condition must be a 1-bit value.
Replace "`if (bus) data_avail = 1`" by "`if (bus > 0) data_avail = 1`".
 - Do not assign signals to "x".
 - Do not infer latches in functions; a function is supposed to synthesize combinational logic.
 - Operand sizes should match.
 - Use parentheses in complex expressions.

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

Well, some of the general coding techniques are as follows. Like when you are giving a condition like for example in the, if then else statement.

(Refer Slide Time: 21:43)



Like if some condition you do something, you say $A = B + C$, but what about this condition. This condition may be, the condition that we are checking, let us say you have declared, let us say 2-bit variable, let us call it status. So, here you are trying to check whether status is not 00 or not, right. So, which means you are checking whether is 01 or 10 or 11.

So, you see there are two ways in which you can write it. You can either write if simply status. So, what does this mean? Status is a 2-bit number, it will be treated as a number with a value. So, in a 2-bit quantity, the value can be either 0 or 1 or 2 or 3 and any non zero value in the if condition is treated as true. So, if it is either 1 or 2 or 3, it will be considered as true, zero will mean false. But what it says that you must not use this kind of multi bit values in status checking, rather you use specifically like this.

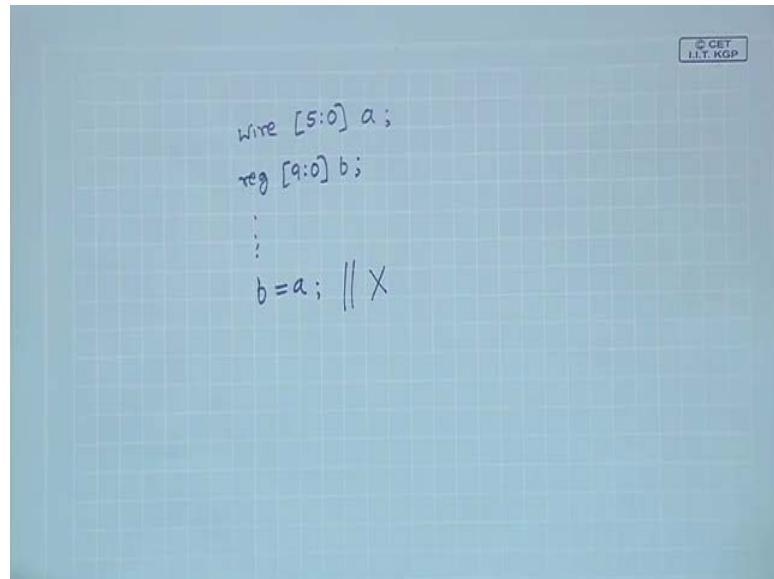
If status greater than zero, then do this. This will also mean the same thing, but here very specifically saying that status greater than, you see status is a 2-bit variable, 0, 1, 2, 3 and greater than is an operator. So, if you consider this expression. This expression can be either true or false, false or true, a single bit value, this expression is a single bit expression, but here status was at 2-bit expression that is why it says the expression in a condition must be a 1-bit value.

So, if bus that same kind of example. So, you avoid this, you rather right bus greater than zero something like this. Do not assign any signal to x, because you see signals are

supposed to be undefined in the beginning, but as computation proceeds you are supposed to initialize them to known values. So, assigning something to x is meaningless. So, you do not do that right. This is of course clear, do not infer latches in functions for multiple branches, you specify the conditions completely. So, a function that is supposed to synthesize combinational logic must not generate latches in synthesis.

Here Operand sizes should match, this is another condition, which is specified here. But of course, Verilog supports unequal Operand assignments also like, let us say suppose I have a variable, let us say 6-bit variable a and I have another reg type variable.

(Refer Slide Time: 25:03)



Let us say this is 10-bit variable b. So, inside a procedural block I can write $b = a$. So, the Verilog compiler will accept it. So, it will just assign the 6-bit to the last 6-bits, then the higher bits will be set to 0, but this guideline says you must not use this kind of assignment. So, the Operands must all be of the same size. So, Operand sizes should match and for complex expressions we use parentheses to clearly specify the presidencies.

(Refer Slide Time: 25:56)

General Guidelines for Synthesis

- Some guidelines for synthesizable blocks are as follows.
 - All “*always*” blocks inferring combinational logic or a latch must have a sensitivity list containing all input signals.
 - Only one clock must be used in an “*always*” block.
 - “*wait*” and “#*delay*” statements must not be used.
 - Conditional expressions must be specified completely; i.e. value must be assigned to a variable under all conditions.
 - The “*initial*” statement must not be used.
 - Expressions must not be used in port connections.
 - Verilog user defined primitives must not be used.
 - Use non-blocking assignments in edge-sensitive constructs.

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

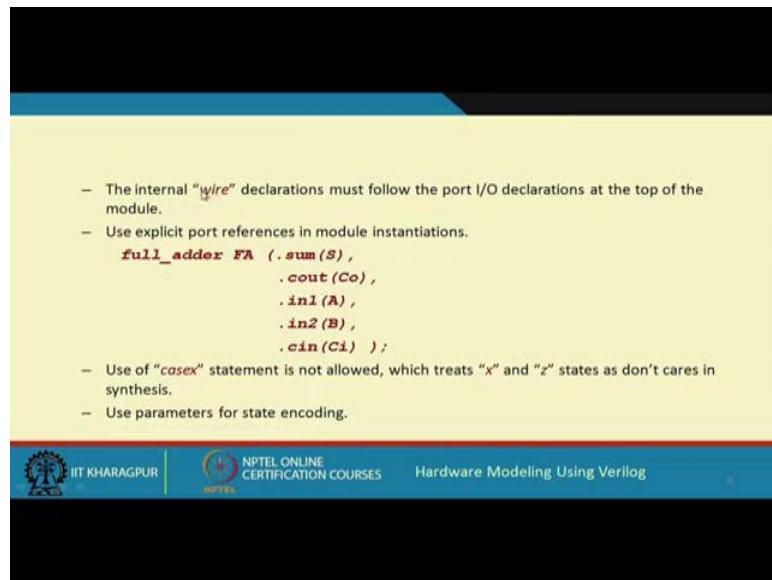
Now, for synthesis already in the last lecture we mentioned a few things, some guidelines have to be followed, like an always block, which is meant to design a combination circuit. A generator combination logic must have a complete sensitivity list. So, all the inputs to that combinational functional block must be there in this list, and you should not use more than 1 clock inside a single always.

This wait and delay statements must not be delay, we have already seen delays used to specify time delay for simulation, and wait is a statement like this, wait within brackets some expression this means the expression is evaluated, and it is true or false. If it is false, then execution is suspended, whatever statements are after that those will be suspended, means until it is true; that means, this expression will be continually evaluated, whenever it is true then only that block will start executing, but again this is something to do with simulation, not with synthesis, so you should avoid this.

Conditional expressions, case, if-else must be specified completely otherwise some latches might be generated; initial statements you cannot use. So, whenever you are instantiating some modules, you must not give expression in terms of ports; like you cannot write in place of some arguments, some $x + y$ or something like that. You can use only single variables for the parameters or arguments when you are instantiating modules, ok.

And this user defined primitives must not be used, even if the synthesis tool might be supporting at least the combination UDPs, but these guidelines says you must not use this. And for edge sensitive constructs, posedge and negedge, you use only non blocking assignments.

(Refer Slide Time: 28:20)



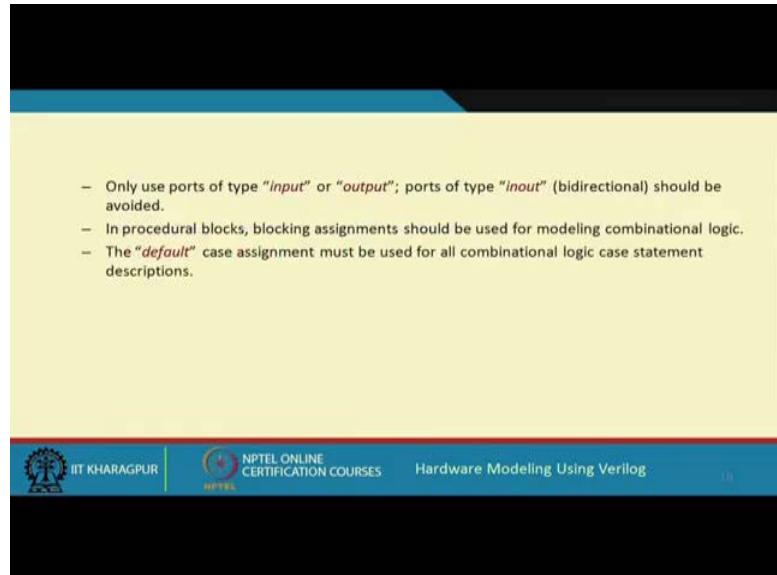
So, the internal wire declarations must follow the IO port declaration which we already do. So, whenever we declare a module, we first define the input and output signals, then we define the wires. So, exactly that is mentioned here that the wire declarations must follow the IO port declarations.

And this is important, it says use explicit port references in module instantiation. So, I said there are two ways to instantiate; one within bracket, to specify the variables in the same order. This is the alternative, you mention the exact name of the variables that was present in the module full adder, and this are the variable names in the current module.

The advantage of this notation is that, even if you change the order of these lines, still instantiation will be correct, right. So, it is said that you use this style, do not use the other style, and one parameter per line, like this. And it says you avoid casex, because in casex statement this states x and z are treated as don't cares. So, which is not good for synthesis; synthesis tool might be generating some wrong hardware. And for state encoding you use parameters, which we have already seen. We had used the state S0, S1,

S2, instead of calling them as 000,001,010, it is always means easy to specify something by names rather than by numbers, fine.

(Refer Slide Time: 30:07)



And it says we use ports only of input and output type, the port inout which have not discussed in our lectures deliberately, bidirectional port should be avoided. So, the inside procedural blocks, blocking assignment should be used for modeling combinational logic only, and for combinational logic again you must use the default statement in case, these are some of the guidelines.

So, just a very simple example which shows you the structure of a program, you see the header can be more elaborate. So, this is a very simple example.

(Refer Slide Time: 30:42)

An Example

```
// Copyright (c) 2017 IIT Kharagpur
// -----
// FILE NAME: counter.v
// TYPE: module
// DEPARTMENT: Computer Sceience and Engineering
// AUTHOR: Prof. Indranil Sengupta
// AUTHOR'S EMAIL: indranil@cse.iitkgp.ernet.in
// -----
// Release history
// VERSION DATE AUTHOR DESCRIPTION
// 1.0 10/08/2016 indranil Initial version
// 2.0 12/07/2017 subir Updated version with clear
// 2.1 16/08/2017 indranil Asynchronous clear
// -----
// KEYWORDS: binary counter, asynchronous clear
// -----
// PURPOSE: 16-bit binary counter
```

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

You see there are so many lines were using for the header copyright, some copyright information or message may be there, then file name, what is the name of this file, what is this type, is a module, Department, Authors, email, because if there is some query, you can contact the author, release history. There are multiple versions of these. So, all the details must be there, what was the version name, which dates the version is created, who was the author and what was the modification. Some of the main keywords and some purpose description, here I have given it in a very brief way, 16-bit binary counter.

You see indentation.

(Refer Slide Time: 31:34)

```
module counter (
    data,
    clear,
    clock
);

output reg [15:0] data; // The 16-bit count value
input clear;           // Asynchronous clear
input clock;          // Counter clock

// 16-bit binary counter with asynchronous clear
always @(posedge clock or negedge clear)
    if (!clear)
        data <= 16'b0000000000000000; // Clear counter
    else
        data <= data + 1;

endmodule
```

This is the module description where instead of putting them on the same line, we have specified them on separate lines. This is the output, these are inputs and with all the variable parameter declaration. There is a comment which specifies the purpose of that parameter. Then with every block, some explanation is there, right, and here also wherever relevant we had comments, clear counter.

So, this is just an example. So, with this, we actually come to the end of this lecture. Actually the example that I have showed you, it is just for you to have a glimpse of the feeling. So, you may think that well my task is to write the module, to create the design. So, why I shall waste my time in writing the header, creating the comments, but well this is extremely important, if you write a correct module without any comments it is as good as useless. No one will be using your module in the future, because in the industry mainly, the main emphasis today is on design re-use.

Suppose, I design some complex subsystem today, tomorrow someone else may be trying to use my design to create a more complex design. So, that person will be trying to re-use my design. So, unless my design is very well documented, this will not be possible.

So, with this we come to the end of this lecture.

Thank you.

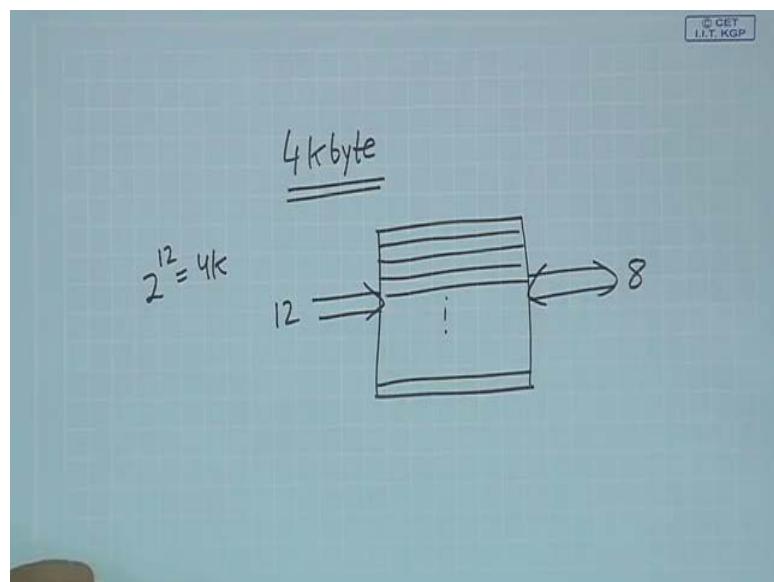
Hardware Modeling using Verilog
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 30
Modeling Memory

So, far we have seen how we can model various sequential and combinational circuits using Verilog language. And you may know the kind of variables that we have used for most of these examples, where either single bit variables or it was a vector. Well in some very specific cases we use 2-dimensional matrices or arrays. Now, in this lecture and the next, we shall be primarily focusing on how to model some specific kinds of 2-dimensional data elements. In this lecture we shall be looking at how to model memory elements.

Now, conceptually speaking, a memory element can be regarded as a 2-dimensional array of storage cells.

(Refer Slide Time: 01:17)



For example, let us say if I talk about a, let us say 4 kilobyte memory chip or a memory cell, what does this mean. So, it is a box, inside which there is storage space for 4 kilobytes of data. Byte means there are 8 data lines available to me, and 4k means there are 2^{12} is 4k. So, there are 12 address lines available to access one of the memory words, and the size of the memory word is 8-bits, fine.

So, how you can declare such a memory cell? I mean as an array, and how we can use it in a Verilog code, we shall be seeing through this lecture.

(Refer Slide Time: 02:14)

How to Model Memory?

- Memory is typically included by instantiating a pre-designed module from a design library.
- Alternatively, we can model memories using two-dimensional arrays.
 - Array of register variables (behavioral model).
 - Mainly used for simulation purposes.
 - Even used for the synthesis of small-size memories.

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

So, the question is how to model memory. Now the first thing I would like to say is that. Well, when you are doing a professional design which will be finally targeted to a hardware. So, when your aim is to synthesize the hardware into an ASIC or a FPGA, then the kind of memory blocks that we use are not exactly what we shall be talking about now. There we will have to pick up some existing memory cells from a library

For instance, when you are using a some kind of FPGA, then there are some predesigned memory cells available in the library, you can pick them up, you can put them in your design and you can create whatever size memory you need, similarly, for a ASIC.

Normally the way memory elements are laid out and fabricated are quite different from the way normal gates are laid out; that is why the layout of a memory is entirely separate and they are typically available as a prefabricated element in the library, fine

So, that is what the first point says, that in a well design where your target is to synthesis, memory will be typically included by instantiating some existing module from a library. Now the examples that we shall be showing today, that as an alternative, we can also model memories using a 2-dimensional array.

Now, you see memory conceptually consist of a number of words. Each word consists of a number of bits. So, each word, we can regard as some kind of a register. So, a memory array can be regarded as an array of registers, where each register has a particular width number of bits, ok.

So, memories can be modeled as an array of register variables. Of course, this is a behavioral model, and we shall be looking into this and this is quite useful for simulation purposes, to check whether the rest of your design is correct or not. And also for memories which are relatively smaller in size, there also you can use this kind of behavioral model, but for larger memories as it said, you have to take some module from a library, design library, fine.

(Refer Slide Time: 04:58)

The slide is titled "Typical Example". It contains two snippets of Verilog code. The first snippet shows a basic declaration of a memory model:

```
module memory_model ( ..... )
...
reg [7:0] mem [0:1023];
...
endmodule
```

The second snippet shows a more detailed declaration with initial values assigned to specific memory locations:

```
module memory_model ( ..... )
reg [7:0] mem [0:1023];
initial begin
mem[0] = 8'b01001101;
mem[4] = 8'b00000000;
end
endmodule
```

To the right of the code, a red box contains two bullet points:

- Each memory word is of type [7:0], i.e. 8 bits.
- The memory words can be accessed as mem[0], mem[1], ..., mem[1023].

The footer of the slide includes the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the title "Hardware Modeling Using Verilog". There is also a small circular portrait of a man in the bottom right corner.

So, a very simple example; so a memory module can be declared as follows: so it is a register array; reg [7:0] mem [0:1023] means, there are 1024 memory words, and each of the word is an 8-bit vector. So, each memory word will be 8-bits, and to access the individual memory words, there are 1024 of them, we will have to just access them using the array notation mem[0], mem[1], up to mem[1023], right

So, just one very simple example how we can initialize some memory content, some particular addresses in memory in some data, so this is the same declaration. Well in an initial block just an example. So, I have written mem[0] equal to; this will actually be a bit. So, this is an 8-bit number 01001100. This is stored in memory address 0, and

mem[4] means this all 0 pattern, this is stored in memory location 4. So, this is how we can initialize a memory word inside a Verilog module. Similarly, you can access it as follows; mem[0], mem[1], mem[2] on the right hand side, same way.

(Refer Slide Time: 06:49)

How to Initialize Memory?

- By reading memory data patterns from a specified disk file.
 - Used for simulation.
 - Used in test benches.
- Two Verilog functions can be used:
 \$readmemb (filename, memname, startaddr, stopaddr)
 (Data is read in binary format)
 \$readmemh (filename, memname, startaddr, stopaddr)
 (Data is read in hexadecimal format)
 - If "startaddr" and "stopaddr" are omitted, the entire memory is read.

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

Talking about initializing memory in general, there are some other methods available that are quite useful for simulation, like you can read some data patterns from a file in the disk. Like suppose I have a large memory, whether or a large number of memory locations, just writing the way I just showed, just assigning values to the different addresses, that way your code will become very long. If there are thousand memory locations, there will be thousand lines of the code

But as an alternative what you can do. You can store the data that you want to load in the memory in a file beforehand, and then there is a function called, using that function, you can load that entire contents from that file into that defined 2-dimensional memory array. So, there are two functions available, and these functions are useful for simulation, particularly in the test benches. These two functions are read memory binary and read memory hexadecimal.

So, in the first function, the data will be stored in the files in binary format, in 0, 1 format; while in the second one they will be stored in hexadecimal format. So, you see there are 4 parameters; first one is the name of the file from where you have to read; then the name of the array memory where you want to store; and start address and stop

address, they indicate that well I mean you may not be wanting to initialize the whole of the memory, maybe only a part of the memory. So, the starting and the ending address, you can specify as a third and fourth arguments. Now if you emit this third and fourth argument, then the entire memory will be read, right.

(Refer Slide Time: 08:46)

Example 1: Initializing a memory from file

```
module memory_model ( ..... );
    reg [7:0] mem[0:1023];
    initial
        begin
            $readmemh ("mem.dat", mem);
        end
    endmodule
```

```
module memory_model ( ..... );
    reg [7:0] mem[0:1023];
    initial
        begin
            $readmemb ("mem.dat", mem,
                        200, 50);
        end
    endmodule
```

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

So, just a couple of examples; so here we had saying read memory in hexadecimal from this file into this variable mem; mem is the memory. Since we have not specified start and end, so, here we will store the entire contents of the 1024 locations from the file.

Here you see, there is another example where we have specified start and end, but start is greater than end, start is 200, end is 50. So, here memory will be initialized by reading from the file, starting from address 200 onward; 200 then 199 then 198, 197, down up to 50. But you can specify the other way round also 50, 200 you can also write. In that case it will start from address 50 and go up to 200, right. So, this is how we can initialize a memory, define memory from data stored in a file, ok.

(Refer Slide Time: 09:56)

Example 2: Single-port RAM with synchronous read/write

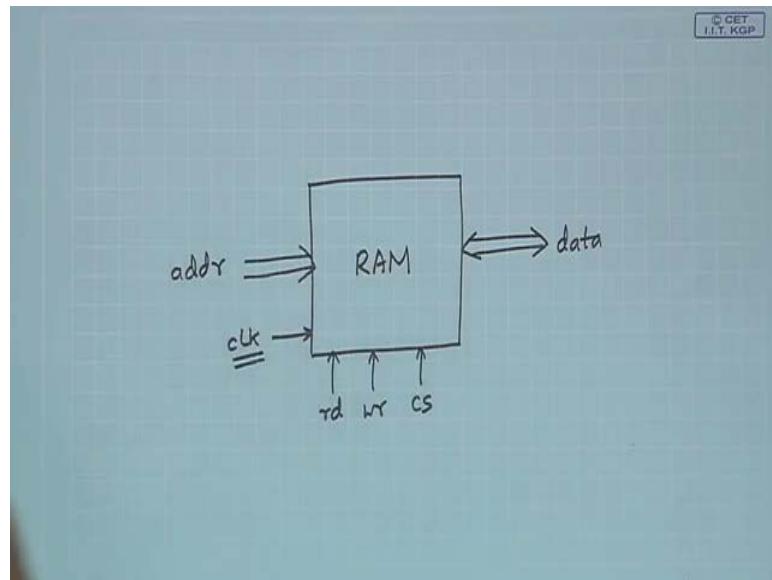
```
module ram_1 (addr, data, clk, rd, wr, cs);
    input [9:0] addr;      input clk, rd, wr, cs;
    inout [7:0] data;
    reg [7:0] mem [1023:0];   reg [7:0] d_out;

    assign data = (cs && rd) ? d_out : 8'bz;
    always @(posedge clk)
        if (cs && wr && !rd) mem[addr] = data;
    always @(posedge clk)
        if (cs && rd && !wr) d_out = mem[addr];
endmodule
```

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

Here there is some examples. This is an example, module declaration where we have used a single port ram with synchronous read, write. So, what does this mean? This means we are trying to design a random access memory which can be read and written both.

(Refer Slide Time: 10:13)



So here I am assuming that there are some address lines, which we call as `addr`; there are some data lines which we are assuming to be bidirectional; that means, I can either write into the ram or read from the ram over the same lines.

Now, in addition there are some control signals; like there is a read signal (rd), there is a write signal (wr), there is a chip select signal (cs). And of course, there will be a clock signal (clk) because it is synchronous, it has to be a clock also. Reading and writing has to be in synchronism with the clock.

Now, here the memories, the data bus is bidirectional. It can be read and also it can be written; that is why you see in the declaration this data has been declared as inout. Well this is a data type which you have not seen so far, the main reason is that it is best avoided because inout, the wait is handled by the simulators and the synthesizers is not consistent. It varies quite significantly from one system to the other. So, it is a good suggestion to the designer, you should avoid inout where possible

But in this example I am showing how inout can be used, you see inout is by default of type wire. So, this is my memory. So, again 1 kilobyte, 1024 x 8. And I am using a variable dout where I want to write it. And from dout to the memory I am using an assign statement, this assign data, data bus. So, whenever chip select is there and read is there, so, from dout, the data will come to this data bus, data and if they are not enabled, then data will be tri-stated 8z, ok.

And it is synchronous operation in terms of read, write, posedge clock; if you are selecting the chip write is active, but not read; that means, you are doing to write, data is written into mem[addr] and if it is read and not write, then you are reading into dout.

Now, you see, you cannot directly read into data, because I said this inout by definition is a wire, and inside a procedural block you cannot have a wire variable on the left hand side; that is why I defined the temporary reg variable dout assigned it there, and in an assign statement from dout, I put it back to data, right.

So, this is one way of specifying a single port RAM with synchronous read and write. With slight modification you can make asynchronous read and write. Like here you see say in the earlier design, you are using the positive edge of the clock for doing the reading and writing.

(Refer Slide Time: 13:40)

Example 3: Single-port RAM with asynchronous read/write

```
module ram_2 (addr, data, rd, wr, cs);
    input [9:0] addr;      input rd, wr, cs;
    inout [7:0] data;
    reg [7:0] mem[1023:0];   reg [7:0] d_out;

    assign data = (cs && rd) ? d_out : 8'bz;
    always @(addr or data or rd or wr or cs)
        if (cs && wr && !rd) mem[addr] = data;
    always @(addr or rd or wr or cs)
        if (cs && rd && !wr) d_out = mem[addr];
endmodule
```

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

But for asynchronous, here we are not using clock, the clock signal is not there anymore, only address, data, read, write and chip select. So, the other declarations are similar.

Now, you see here the two always blocks are activated by change in any one of the input variables; either address changes or data changes or read, write, chip select, the rest is the same. So, synchronous and asynchronous, both memory model definitions are very similar. So, in one case we are using the clock edge to synchronize my reading and writing, and in the other case there is no clock. Whenever the signals changes, so, in accordance to that we are doing either reading or writing depending on whether read is active or right is active, fine.

Now, as it said this inout is a not a very good way of using it in a module. We shall see it a little later, but first let us see.

(Refer Slide Time: 14:46)

The slide has a yellow header and footer. The title 'Example 4: A ROM / EPROM' is at the top. The footer contains the IIT Kharagpur logo, NPTEL Online Certification Courses logo, and the course name 'Hardware Modeling Using Verilog'. There is also a small circular portrait of a man.

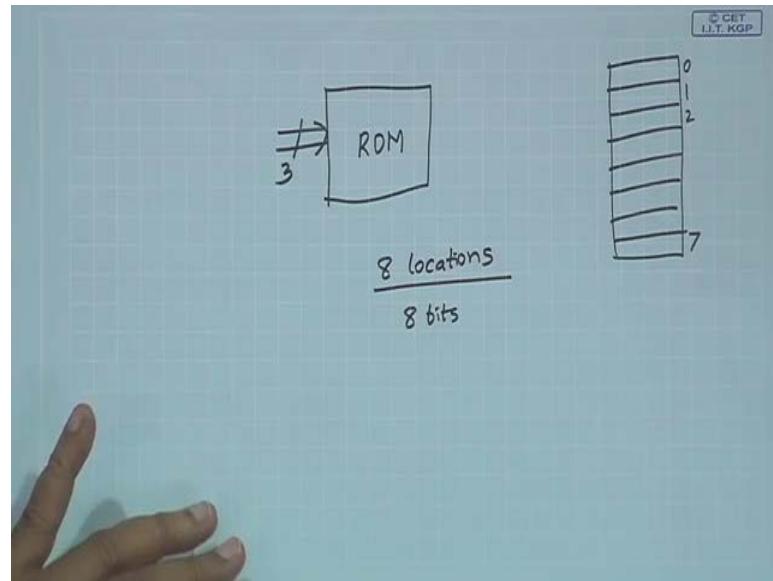
```
module rom (addr, data, rd_en, cs);
    input [2:0] addr;      input rd_en, cs;
    output reg [7:0] data;
    always @(addr or rd_en or cs)
        case (addr)
            0: data = 22;
            1: data = 45;
            .....
            7: data = 12;
        endcase
    endmodule
```

So, how we can model a memory where you are not writing only reading; that means, it can be either be a read only memory, or it can be a erasable programmable read only memory EPROM.

So, in a ROM or EPROM, we are assuming that we are storing some data in the memory, that is fixed. We can only read from there. So, for a ROM when you model a ROM or EPROM, you need not use that kind of an array notation where you are storing and reading like that, rather you can use a case statement. So, here we have used a very small example.

Let us assume that my address is 3-bits. So, here we have taken a very small example.

(Refer Slide Time: 15:37)



Let us say I have A ROM the address lines are 3, 3-bit address. So, inside the ROM there will be 8 locations, and each locations I am assuming they are 8-bits in size. So, what I do, I just specify like a lookup table, there will be 8 locations. So, I store the 8-bit values, and depending on the address. So, one of these will be selected, address 0, 1, 2 up to 7, right. This is done by using a case statement.

So, you see, here there is a always block, this is activated when either address or there is a signal called read enable (rd_en), because from a ROM you can only read you cannot write and chip select. So, when any of them these are active then you start the reading. There is a case (addr). The address can be 0, 1 up to 7 and you have stored some fixed data inside.

So, if you have given address equal to 0, data equal to 22; if it is 1, data equal to 45 and so on. There will be 8 lines like this. So, this is how you typically model a ROM or EPROM, where you do not change the data, data is fixed; that is why you are using a fixed case statement, right, fine.

(Refer Slide Time: 17:12)

An Important Point to Note

- Some simulation or synthesis tools give inconsistent behavior when using the “*inout*” data type.
 - Such “*inout*” bidirectional data should be avoided.
- A better way to design a memory unit is to keep the data input and data output bus signal lines separate.
 - An example memory description with separate data buses is shown on the next slide.

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

Now, the point that I was making, this inout is a data type which is best avoided, because as I said some simulation or synthesis tools can give inconsistent behavior. So, the experience designer says that do not use inout data type inside a Verilog code, but does that mean that I will not use any bidirectional lines in my hardware. You see there are many instances where I do need bidirectional lines, memory is a very good example. So, I will use bidirectional lines, but I will not use inout data type in my Verilog module in my Verilog code, let us see how.

So, what we do. We keep the data input and data output lines separate, and then we will see how you can make it bidirectional. First let us do it in one step; first step says data input and output lines are kept separate.

(Refer Slide Time: 18:17)

The slide is titled "Example 4". It features a block diagram of a RAM module labeled "ram_3". The inputs are "addr", "wr", and "cs". The output is "data_out". A feedback line goes from "data_out" back to the "wr" input. To the right of the diagram is the corresponding Verilog code:

```
module ram_3 (data_out, data_in, addr, wr, cs);
parameter addr_size = 10, word_size = 8,
          memory_size = 1024;
input [addr_size-1:0] addr;
input [word_size-1:0] data_in;
input wr, cs;
output [word_size-1:0] data_out;
reg [word_size-1:0] mem [memory_size-1:0];

assign data_out = mem[addr];
always @(wr or cs)
  if (wr) mem[addr] = data_in;
endmodule
```

The footer of the slide includes the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the title "Hardware Modeling Using Verilog". There is also a small portrait of a man.

So, how does it look like, it is something like this. So, I have a memory a RAM. So, I have my address, data_in is separate, data_out are separate, 2 set of data buses and read, ok.

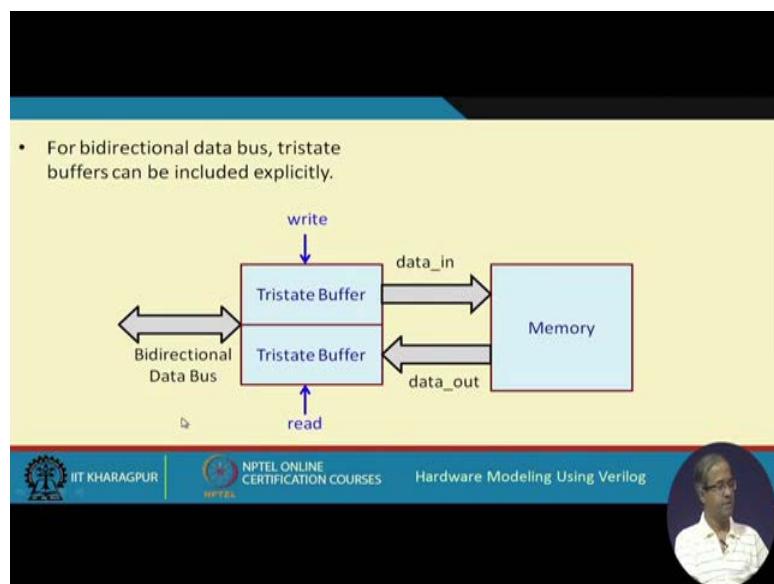
There is no separate read line, what is meant is write is like a read write both; if write signal is high, it means write; If write signal is low, it means read. So, it is like a write slash read bar. So, I am just assuming that this kind of a signal is there.

So, let us see how the module definition will look like. So, here we have used parameters to make our specification general. So, we have assumed addr_size is 10, it is a 10-bit address which means 1k words; word size is 8, means 1 byte per location. And of course, memory size will be 1024, 2^{10} .

So, the parameters are data_out, data_in, these are the arguments addr, wr and cs. So, we have declared the variables; the addr is of size 0 up to addr_size - 10; data_in, data_in its an input. Again 0 up to word_size - 1; word_size is 8. And similarly for the output, data_out is an output that is it is an output. So, here, the size is again word_size - 1, 8 and wr and cs, are single bit inputs. And here we have defined our memory. This is number of bits per word, 0 up to word_size - 1, and this is the total number of memory locations, it is 1024.

Now, the way we have implemented is very simple, writing, we are doing in the always block. So, whenever wr or cs, that means, this changes, then inside it you check if write equal to 1, then only you write `mem[addr] = data_in`. Since `mem` is of type `reg`, there is no problem, you can write it. But for reading the data, we are not using always block. We are using an assign statement, because `data_out` is declared just an output which is wire, it is not `reg`. So, `mem[addr] = data_out`. So, from `data_out`, it will always be reading out, but whenever there is a wr active, `data_in` will be written inside, right. This is the specification of the memory module here.

(Refer Slide Time: 21:13)



Now, let us see how we can make it bidirectional. You see this much we have implemented. We have implemented a memory module with separate `data_out` and `data_in`. Now what you do, we use two sets of tristate buffers. So, one we activate by write, other we activate by read. Now in the previous example there was no separate read signal. So, we can activate it by write bar; so when write is 0. And on the other side the two tristate buffers are connected together, and we have a single bus. So, you enable either this or this, both of them are not enabled together

So, when you are activating write, this is off, and whatever is coming here, this will go through this into the `data_in`, and whenever you are activating read, this is off, this is on. So, whatever is coming out of `data_out` that will be coming by this, ok.

So, now you see in the module you are not declaring anything as inout, but by using tristate buffer, you have implemented a bidirectional bus, sometimes you are moving here, how you are moving here. So, how you can declare, you can declare it very simply like this. Let us say this variable, let us call it bus, here we defined it as tri, this is a tristate; data_out and data_in are of type wires let us say, and we have two assign statements, assign bus equal to, when read is active. When read is active, data_out will go to bus, data_out will go to bus, and otherwise it is tristate, this buffer is tristate.

And in the other assign statement, data_in equal to, depending on write bus, bus will go in or if write is not active then tristate. So, just a normal memory module with separate input and output data, and these kinds of tristate bus and enabling them will make the overall thing a bidirectional data bus enabled, fine.

Now, just a simple test bench we are writing just to test this ram3 module which we just wrote. So, we are just showing a simple test bench.

(Refer Slide Time: 23:41)

```

module RAM_test;
reg [9:0] address;
wire [7:0] data_out;
reg [7:0] data_in;
reg write, select;
integer k, myseed;

ram_3 RAM (data_out, data_in, address, write, select);

initial
begin
  for (k=0; k<=1023; k=k+1)
  begin
    data = (k + k) % 256; read = 0; write = 1; select = 1;
    #2 write = 0; select = 0;
  end
end

```

So these are the two parts of the test bench, let us see. Here we have instantiated the memory data_out, data_in, address, write and select.

So, here this is wire. So, this address is of type reg, data_out which will be coming out of the RAM; this is of wire, data_in will also go inside. So, that has to be of type reg, write

and select you have to activate, chip select and write those are also type reg, and we have defined 2 integers in addition k and myseed for the purpose of writing the test bench.

Now, here you see what we do, in the first initial block we initialize the whole memory. How you do in, a for loop, k equal to 0 up to k equal to 1023, we go in a loop and what you store, we store k plus k modulo 256. So, if the address, if k is 0,0 plus 0 is 0. So, if k is 1, 1 plus 1 is 2. If k is 10, 10 plus 10 is 20. If k is 100, 100 plus 100 is 200. If k is 200, 200 plus 200 is 400 modulo 156, divide by 156 and take the remainder. So, it will be 144 like that data will get stored. This is the initialization part.

So, when you store something in the data, you activate read = 0, write = 1 and select = 1, so that this data will get written into the corresponding memory location. So, this gets written.

(Refer Slide Time: 25:56)

The screenshot shows a Verilog code block and its corresponding simulation output. The code is as follows:

```
repeat (20)
begin
    #2 address = $random(myseed) % 1024;
    write = 0; select = 1;
    $display ("Address: %d, Data: %d", address,
              data);
end
initial myseed = 35;
endmodule
```

The simulation output on the right lists 20 random addresses and their corresponding data values:

Address	Data
0	0
960	128
482	196
693	106
246	236
228	200
148	40
767	254
355	198
259	6
21	42
872	208
758	236
193	130
909	26
632	240
719	158
214	172
67	134
908	24

So, after this, now here you repeat 20 times, just I am randomly selecting 20 addresses, and I am reading them out and seeing what is there. So, I am just calling the random number with the seed, mod 1024. So, an address in the range 1 up to 1023 will be generated, that is my address; write is 0 that means, I want to read; select is 1, I display address, data, so whatever is coming out

So, if I simulate it I see that my output is coming like this. So, repeat 20, there will be a 20 lines generated, this will be the random addresses, that the smaller one you check

address is 21. So, what does address 21 contain, k plus k. So, which is 42; so the others one also 193. So, 193 is how much, 193 multiplied by 2, just let us check, 193 multiplied by 2 is 386.

(Refer Slide Time: 27:02)

The image shows a handwritten mathematical calculation on a grid background. At the top right, there is a small logo with the text "©CET I.I.T. KGP". The calculation itself is as follows:

$$193 \times 2 = 386 \quad \% \quad 256$$
$$\begin{array}{r} - 256 \\ \hline 130 \end{array}$$

The calculation is performed in two steps: first, 193 is multiplied by 2 to get 386; second, 386 is divided by 256 to find the remainder, which is 130.

Now $386 \bmod 256$ means what, divide by 256 and take the remainder. So, basically minus 256 which means 130; so you see in location, here 193 data is 130. So, accordingly you can verify the operation.

Now, here there is one mistake that I see which I have missed out, here I have not put the address. We will have to put another line here. Here you will have to put address = k, this is missing, because where you will be writing. We will be writing it into address k, right, data you are calculating, you are storing it into data, right, and this write and select are activated, this will be data_in actually, data_in, fine. So, you have declared it as data_in. So, data_in will be getting this data address, address will be k and then you write 1, select = 1 it will be written, like this.

So, with this we come to the end of this lecture. So, in this lecture we have basically seen how we can model some memory devices as a 2-dimensional array, as an array of registers and at least from the simulation point of view, wherever we need memory elements in a larger design, you can implement the memory systems like this. So, we shall see some examples later also, where you use these kind of memories in a larger system and you can simulate them and see how it works.

And again I had said for smaller memory systems, even the synthesis tool will be able to synthesize the memory, but it will not be very efficient in terms of hardware. They will be implementing the synthesis tool will be implementing the hardware in terms of flip-flops and arrays of flip-flops. But in actual memory chip, the weight is constructed, it is much more compact, much more efficient in terms of number of transistors required per cell, per storage cell, ok.

Thank you.

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

Lecture - 31
Modeling Register Banks

So, in this lecture we shall see how we can model register banks or register arrays or register files. Now we have seen some examples earlier like we saw how we can design the data path and control path for certain designs like multiplied GCD computation etc. Those designs were pretty small designs, but even there we saw, there was the requirement of several registers. Like for a multiplied you needed register A, Q, M and so on. But you think of more complex systems, you think of a computer a processor. So, inside a processor there will be much larger number of registers. Well, a modern day processors can have 64, 32, 128 registers, very large number of registers.

So, how to implement those registers? Registers you can implement as individual reg - reg0, reg1, reg2 like that, individual variables. Or you can also stored them similar to a memory, model them like a memory and just allow the synthesis tool to synthesize the register bank as an array of hardware registers. Because usually register banks or register files they are not implemented using memory technology, but that implemented using flip-flop which are much fast, in a much faster, static ram technology, ok.

(Refer Slide Time: 02:04)

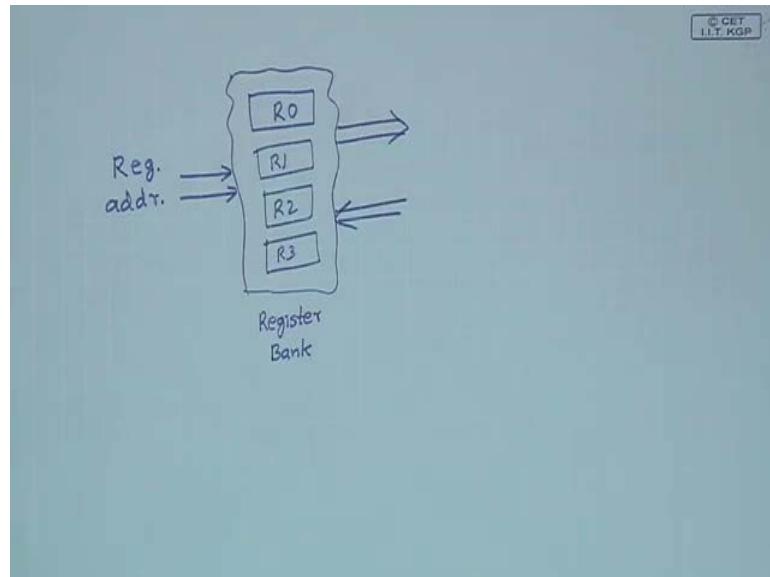
Introduction

- A register bank or register file is a group of registers, any of which can be randomly accessed.
 - Commonly used in computers to store the user-accessible registers.
 - For example, in the MIPS32 processor, there are 32 32-bit registers, referred to as *R0, R1, ..., R31*.
- Can be implemented in Verilog as independent registers, or as an array of registers similar to a memory.
- Registers banks often allow concurrent accesses.
 - MIPS32 allows *2 register reads and 1 register write every clock cycle*.

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

So, the topic of this lecture is modeling register banks. So, the first question is what is a register bank? A register bank which is sometimes also called a register file, it is a group of registers.

(Refer Slide Time: 02:24)



Let us take an example. Suppose I have a register R0, I have a register R1, I have a register R2, I have register R3, let us say 4 registers. So, I can use them individually in a program R0, R1, R2 or R3, but suppose what I say is that well, let me group all the 4 registers together and let me call this a register bank, right.

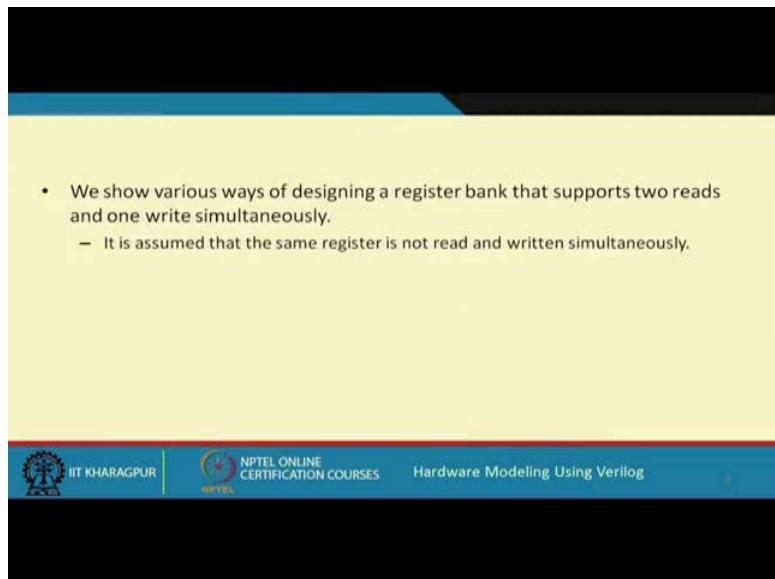
Now, in this register bank from outside I can specify a register address. A 2-bit register address because there are 4 registers in 2-bits, I can specify R0, R1, R2 or R3. So, with this register address I will be selecting one of the registers and then I can either read or I can write into that register. This is the concept of a register bank. So, just to summarize I can use the registers individually as variables and read and write from them individually by the names, or I can treat them as a group just like a memory element and I can read and write from there, fine.

Now, a register bank as I said is typically used in a processor in a computer to store the registers which are typically used for temporary storage of data. Now as an example there is a common processor call MIPS32, these are an example of a reduced instruction set architecture. Here there are 32 registers and each of the registers are of 32-bit size. And these 32 registers are referred to as R0, R1 up to R31.

Now, as I said in Verilog you can either define 32 variables and implement them as independent registers, or you can implement them as an array of registers which will be very similar to modeling memory. Well, not only this a register bank typically supports some other features like concurrent access. As an example for this MIPS32 processor that I was talking about, here the registered bank is so designed that you can carry out two register reads and one register write in every clock cycle. Which means 3 operations can be carried out concurrently on the register.

But of course, suppose you are writing into register R5 and you are at the same time reading from R5, there will be inconsistency.

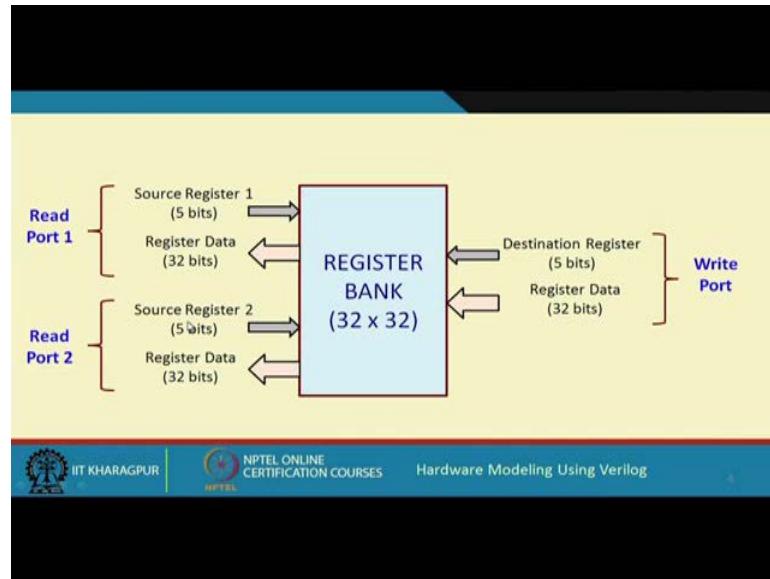
(Refer Slide Time: 05:47)



So, you should not read and write into the same register, but you can read from R5, read from R10 and write into R8 that is possible and these 3 things can go on in parallel, fine.

So, we shall be showing some alternate designs of register bank, which will support this kind of concurrent access that means, two reads and one writes at the same time. And of course, I made this assumption same register is not read and written together because if it is so there will be some conflict.

(Refer Slide Time: 06:13)



So, pictorially our register bank well the MIPS32 register bank, 32 registers each of 32-bits will look like this. There will be 3 access ports, two read and one write. In the read ports what do we have? We have a register number that which register I want to write, because there are 32 registers, you need 5-bits, here.

Similarly, for the second port there are 5-bits. This will actually be source register 2 not 1, this is 1, and this is 2. And so once we have specified the register number we can read them and the data will be available over these 32-bits, and on the other side when you are writing. So, again you have to specify which register you want to write, that is again 5-bits and the actual data you want to write 32-bits. So, there two read ports and one write port, fine.

(Refer Slide Time: 07:27)

```
// 4 x 32 register file
module regbank_v1 (rdData1, rdData2, wrData, sr1, sr2, dr, write, clk);
    input clk, write;
    input [1:0] sr1, sr2, dr; // Source and destination registers
    input [31:0] wrData;
    output reg [31:0] rdData1, rdData2;
    reg [31:0] R0, R1, R2, R3;

    always @(*)
        begin
            case (sr1)
                0: rdData1 = R0;
                1: rdData1 = R1;
                2: rdData1 = R2;
                3: rdData1 = R3;
                default: rdData1 = 32'hxxxxxxxx;
            endcase
        end
endmodule
```

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

Let us take a simple example to start with. Let say we do not have 32 registers, but we have only 4 registers, small number. So, because it is small we can declare the registers individually. This is what I am assuming. So, how is our declaration? Our declarations like this. So, in our register bank there are two read ports and one write port as I said, rdData1, rdData2, wrData. So, what are these 3? This is rdData1, rdData2 and this is wrData. And we have sr1, sr2 and dr, this is here; sr1, sr2 and dr, these are all. So, in this case because there are 4 registers, this will be 2-bits each and of course, write and a clk, because read is by default.

Let us see the variable declarations clk and write will clearly be inputs. And sr1, sr2 and the dr, they will be 2-bits because there are 4 registers. wrData, rdData1, rdData2 all are 32-bit quantities and for rdData, since reading and assigning, they defined as reg, ok.

And the data you want to write that is coming from here and that is declared as input wire. And we define the 4 registers like this R0, R1, R2, R3. So, you will be using a setup always blocks very simple. This is always block for the port1. So, you check always whenever something is changing case (sr1), source register 1 whatever is sr1 is 0 1, 2 or 3 because it is 2-bits; if it is 0 in the value of R0 will go to rdData1; if it is 1 then R1. If it is 2, R2; if it is 3, R3, but if it is not; if it is other than 0, 1, 2, 3 because you see these are Verilog variables. So, the Verilog variables can also contain values which are z or x, not necessarily only 0 and 1, ok.

(Refer Slide Time: 10:18)

```
always @(*) begin
    case (sr2)
        0: rdData2 = R0;
        1: rdData2 = R1;
        2: rdData2 = R2;
        3: rdData2 = R3;
        default: rdData2 = 32'hxxxxxxxx;
    endcase
end
always @ (posedge clk)
begin
    if (write)
        case (dr)
            0: R0 <= wrData;
            1: R1 <= wrData;
            2: R2 <= wrData;
            3: R3 <= wrData;
        endcase
    end
endmodule
```

This way of modeling is feasible if the number of registers is small.

So, if it is anything else then rdData1 will be assigned the undefined value xxxx. This is for the first read port. Similarly, for the second read port, same block, but here case (sr2) depending on sr2; R0 will be assigned to rdData2 or R1 or R2 or R3. This is for the second port. And third one is for write and writing is in synchronism with a clock. So, whenever there is a clock then only we are writing. If write is active, then you see what is your destination register. So, if dr is 0, you write into R0; if dr is 1, you write into R1 and so on.

So, these is a very simple description of a register file or register bank containing 4 registers, where the registers are defined individually independently. Let us see some other ways in which we can define or declare such register bank definitions. So, this as I said this way in numerating the values is feasible if the number of registers is small. Just imagine if there are 32 registers will be too tedious, we have to write 32 lines, here 32 lines here and so on.

(Refer Slide Time: 11:36)

```
// 4 x 32 register file
module regbank_v2 (rdData1, rdData2, wrData, sr1, sr2, dr, write, clk);
    input clk, write;
    input [1:0] sr1, sr2, dr; // Source and destination registers
    input [31:0] wrData;
    output [31:0] rdData1, rdData2;
    reg [31:0] R0, R1, R2, R3;

    assign rdData1 = (sr1 == 0) ? R0 :
                    (sr1 == 1) ? R1 :
                    (sr1 == 2) ? R2 :
                    (sr1 == 3) ? R3 : 0;
    assign rdData2 = (sr2 == 0) ? R0 :
                    (sr2 == 1) ? R1 :
                    (sr2 == 2) ? R2 :
                    (sr2 == 3) ? R3 : 0;

    always @(posedge clk)
    begin
        if (write)
            case (dr)
                0: R0 <= wrData;
                1: R1 <= wrData;
                2: R2 <= wrData;
                3: R3 <= wrData;
            endcase
        end
    endmodule
```

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

Now, let us again keep this number as 4, registers define independently, but we are using an alternate way of implementing the read ports. Like in the earlier case you see, the read ports were implemented using an always block and a case. Here we are implementing them using assign statements. The first part is identical, there is no change here. Just only think because here rdData1, you are not assigning inside in always block. So, we have not given reg here, only output, right.

Here this is like an if then else kind of a statement using the conditional operator. What is, what does mean? It means if $sr1 == 0$; if this condition is true, then $R0$ is assigned. If not, otherwise if $sr1 == 1$, if this is true then $R1$ assigned. Otherwise if it is 2, then $R2$; otherwise if it is 3, then $R3$; if nothing matches, here given 0 or we can give undefined also if you want. Similarly, $rdData$ is identical with $sr2$. And the third block is identical again with a clock for writing. So, this is how you can implement a register file and you can do reading and writing like this.

(Refer Slide Time: 13:13)

```
// 32 x 32 register file
module regbank_v3 (rdData1, rdData2, wrData, sr1, sr2, dr, write, clk);
    input clk, write;
    input [4:0] sr1, sr2, dr; // Source and destination registers
    input [31:0] wrData;
    output [31:0] rdData1, rdData2;

    reg [31:0] regfile[0:31];

    assign rdData1 = regfile[sr1];
    assign rdData2 = regfile[sr2];

    always @ (posedge clk)
        if (write) regfile[dr] <= wrData;
endmodule
```

Let us now come to a complete register file like MIPS 32 by 32. Here let see how we can modeled, here we are using an array like a memory. So, the arguments are the same rdData1, rdData2, wrData, sr1, sr2, dr, write and clock. The difference is here sr1, s2 and dr will be 5-bit quantities, because there are 32 registers and 2^5 is 32. That is why we will be needing 5-bits to represent a particular register, right.

So, this is how we are declaring our register file just like a memory. So, each word contains 32-bits and there are 32 locations, 0 up to 31. So, for reading we can simply give assign statements, rdData1 = regfile[sr1], this sr1 as the index; for the second port regfile[sr2], sr2 as the index. And for write, always @ (posedge clk), if write is active, wrData goes to regfile[dr] simple, right. So, so in a behavioral way is very compact and simple to specify a register file like this.

Now, let us in addition let us add another facility, like you see when you have a register file means often there is a facility using which you can reset the registers from outside. Resetting means let say there will be a reset signal, control signal. So, if we activate reset from outside, all the registers, they will be initialized to 0s, this is what you want.

(Refer Slide Time: 15:21)

```
// 32 x 32 register file with reset facility
module regbank_v4 (rdData1, rdData2, wrData, sr1, sr2, dr, write, reset, clk);
    input clk, write, reset;
    input [4:0] sr1, sr2, dr;      // Source and destination registers
    input [31:0] wrData;
    output [31:0] rdData1, rdData2;
    integer k;

    reg [31:0] regfile[0:31];
    assign rdData1 = regfile [sr1];
    assign rdData2 = regfile [sr2];

    always @(posedge clk)
    begin
        if (reset) begin
            for (k=0; k<32; k=k+1) begin
                regfile[k] <= 0;
            end
        end
        else begin
            if (write)
                regfile[dr] <= wrData;
        end
    end
endmodule
```

The screenshot shows a Verilog code editor with a module named `regbank_v4`. The code defines a 32x32 register file with a reset facility. It includes inputs for clock, write, and reset, and outputs for two register reads. A detailed explanation of the `always` block is overlaid on the right side of the code, highlighting the synchronous initialization logic. The code editor also displays logos for IIT Kharagpur and NPTEL, and the title "Hardware Modeling Using Verilog".

So, in the next version is just a modification of this. Here we are saying this is a 32 by 32 register file, but with reset facility. So, what is the change? The change is, there is a variable additionally included here called `reset`, which is also an input variable and we have declared an integer `k`, rest is the same. The way you read from register file, there is no change, just in the previous one exactly the same, but while writing there is a little change, this is synchronous.

So, whenever there is a clock, you first check whether `reset` is active or not. If `reset` is active then in a for loop, you go from `k` equal to 0 up to `k` equal 31, and initialize all `regfile[k]` is to 0s. So, all the registers are initialized to 0. But if `reset` is not active, you come to the else part, you see if `write` is active or not; if `write` is active, then you write this data into `regfile[dr]`, right. So, this is how it works.

(Refer Slide Time: 16:39)

```
module regfile_test;
reg [4:0] sr1, sr2, dr;
reg[31:0] wrData;
reg write, reset, clk;
wire [31:0] rdData1, rdData2;
integer k;

regbank_v4 REG (rdData1, rdData2, wrData, sr1, sr2, dr, write, reset, clk);

initial clk = 0;
always #5 clk = !clk;

initial
begin
$dumpfile ("regfile.vcd"); $dumpvars (0, regfile_test);
#1 reset = 1; write = 0;
#5 reset = 0;
end

```

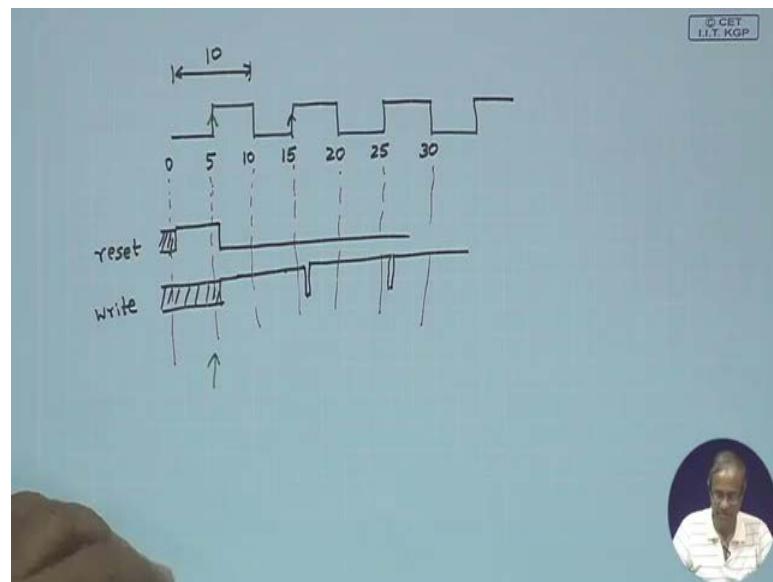
A test bench to verify operation of the register file

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

Now, let us look at a test bench to verify the regbank_v4, which means we are just using the register bank with the reset facility, the last example that we took, this one, right, this one let us write a test bench for it. So, what you have done? Let us see. So, we have instantiated the register bank, we have given a name reg. So, we have not change the names, we can change the names if you want. So, rdData1, rdData2 wire inputs, now here they are output, they were actually coming out. So, in this case they will be wires, but wherever you are writing they will all be reg, reg wrData, sr1, sr2, dr, write, reset, clk this will all be reg, right, and we are defining an integer k.

Let say what we are doing here, we are initializing clock to 0 and always at 5, clk = !clk, this means again we are using a clock signal like this.

(Refer Slide Time: 18:07)



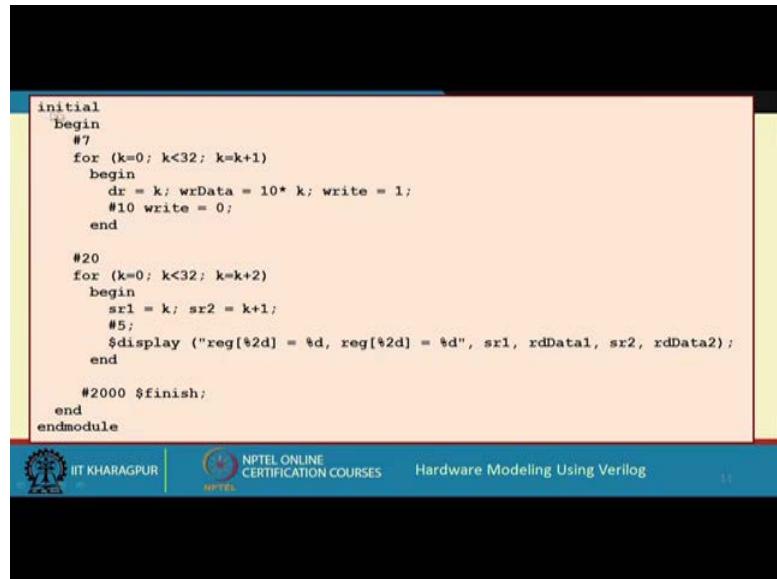
Which start with time 0, set 5 toggle, again after 5 toggle, again after 5, again after 5, like this. So, so after time 5, we are toggling the states. So, our time period will be 10, this will be our time period which is 10.

Now, let us look into the timing of this signal, what we are actually doing. These are the edges, right. Now what we have done? Now in this initial block, we are actually creating the reset signal. So, here of course we have specified a dumpfile and the variables to dump. We are, see we are activating reset at time 1, write is 0. So, that write does not take place and after a gap of 5, we are setting reset back to 0. So, what will happen? This is time 0, right.

So, if you look into the reset signal, reset initially it was undefined. So, it was. So, at time 1, it is undefined let say this is one; at time 1, you are setting reset to 1. And it continues 1 up to time 6, at time 6, you set reset to 0. And after that you do not make it 1 again. So, what does this mean? This means that when the first clock edge comes here, out here, your reset is 1; that means, initially all the registers will be reset to 0s, right. This is what you do here.

And then there is another part of the test bench. You see here, from, there is another initial blocks, so, that again it starts from time 0.

(Refer Slide Time: 20:20)



The screenshot shows a Verilog simulation environment. The main window displays the following Verilog code:

```
initial
begin
  #7
  for (k=0; k<32; k=k+1)
    begin
      dr = k; wrData = 10* k; write = 1;
      #10 write = 0;
    end

  #20
  for (k=0; k<32; k=k+2)
    begin
      srl1 = k; sr2 = k+1;
      #5;
      $display ("reg[%d] = %d, reg[%d] = %d", srl1, rdData1, sr2, rdData2);
    end

  #2000 $finish;
end
endmodule
```

Below the code, the status bar indicates the source file is "Hardware Modeling Using Verilog". The footer features the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the text "NPTEL".

So, after a gap of 7, you see what you do? You go in a for loop 0 up to 31, you put k into destination register ($dr = k$), write data 10 into k ($wrData = 10*k$), you write it, write equal to 1. $10*k$, means register number i, you are storing the value $10*i$. So, register0 will get 0; register1 will get 10; register2 will get 20; 3 will get 30 and so on.

So, let us say we initialize registers like this. So, $wrData = 10*k$. And there is a delay of 10, after delay of 10, I set back $write = 0$ and repeat this again, this is repeated. So, what does this mean? You think of the write signal, what happens, write becomes 1 after a delay of 7 and after a delay of 10, it again becomes 0. So, what does this mean? So, how will the write signal look like. Write will become 1 at times 7, 7 is here; so, this is 5. So, it is somewhere here, before that it was not specified. So, write will be 1 here and it will remain 1 for a period of time 10; that means, it will remain 1 till here, then it will again go 0, right.

So, like this it will go on. So, when the next clock edge comes, it will be high and write will take place. So, again when you go back to the loop; so again this will be 1; again after a time 10, it will be 0; again it will be 1; like this it will go on. So, it will be, writing will be taking place with every clock edge, one by one, ok

So, in this loop the writings will take place. Well, after the writings are done here just as an example, here we are just observing the contents of some of the registers. So, what we are doing, k equal to 0 up to 31, again? Here we are reading 2 registers at a time, and

here $k = k + 2$, as skipping by 2, k equal to 0, 2, 4, 6, 8 like that. And $sr1$ we are initializing k and $sr2 = k + 1$. And we are displaying the values of a $sr1$, $rdData1$, $sr2$, $rdData$, whatever data is stored there. And we are printing it like this reg within bracket register number equal to data, like that, and we finish here.

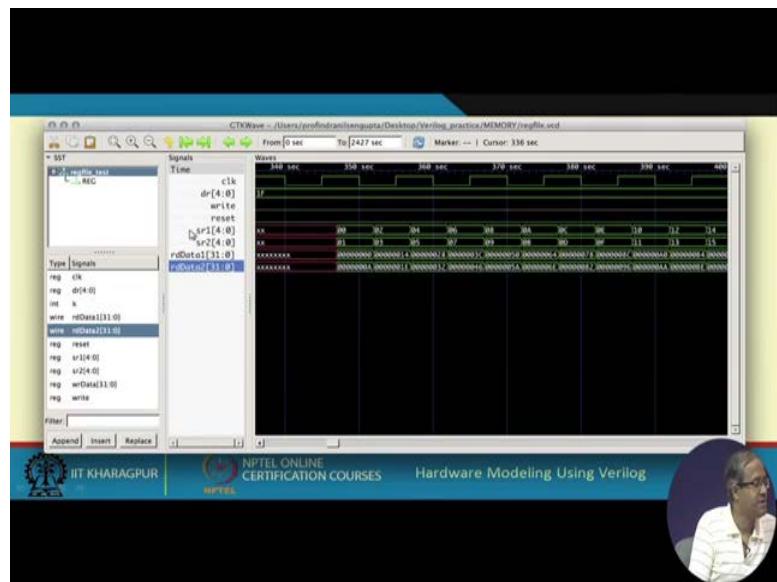
(Refer Slide Time: 23:29)

reg[0]	0	reg[1]	10
reg[2]	20	reg[3]	30
reg[4]	40	reg[5]	50
reg[6]	60	reg[7]	70
reg[8]	80	reg[9]	90
reg[10]	100	reg[11]	110
reg[12]	120	reg[13]	130
reg[14]	140	reg[15]	150
reg[16]	160	reg[17]	170
reg[18]	180	reg[19]	190
reg[20]	200	reg[21]	210
reg[22]	220	reg[23]	230
reg[24]	240	reg[25]	250
reg[26]	260	reg[27]	270
reg[28]	280	reg[29]	290
reg[30]	300	reg[31]	310

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

So, when we just run this. Just if you look at the simulation output like whatever is getting printed by this display, we see something like this. You see what you expected is actually that. So, each register is stored the twice the value, register number 8, 10 times the value, sorry; so 8 contains 80, 13 contains 130, register 23 contains 230; 28 contains 280. So, it is actually 10 times it is stored, ok.

(Refer Slide Time: 24:00)



Now, if you also see the timing diagram. So, I am showing you 2 parts of the timing diagram. This is the first part where reset has been activated, initially reset has been activated, and then you are writing the data 10^*k , you see the register number dr is changing 00, 01, 02, 03. So, the write is continuously active, because it is going low and immediately going high. So, that you are not seeing here, write is continuously active. So, 0, 1, 2 with the clock edge writing is going on, ok.

And towards the end, this is towards the end of the simulation. Here you are, you are reading the values, you see sr1 or sr2; 01 then 2, 3 then 4, 5, 6, 7 like that. And you see the rdData1, rdData2 for 01 the data are 000 and 000a, which means 0 and 10. 2 and 3 are 0014 means 20 and this is 1e means 30; for 4 and 5 it is 28 hexadecimal means 40; 32 means 50. So, like that the data values are actually coming.

So, you see in this way you can model a register bank or register file, and you have seen through the test bench that actually you are able to read and write together at the same time. So, we will see some more examples of register banks later when we talk about more complex systems that involves something called pipelining, but this we shall be discussing later.

So, with this we come to the end of this lecture. So, in this lecture whatever we have basically seen, we have summarized how we can define register banks or register files and how we can have multiple read and write ports, you can read from the register banks,

you can write into the register banks. And how we can write a test bench and simulate the design using the test bench.

So, we shall see later as I had said that will be using some more complex designs where we will be using this kind of register files as a basic building block. So, whatever description we have given something similar to that, that will happen as part of the main model description therein.

So, with this we come to the end of this lecture.

Thank you.

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

Lecture - 32
Basic Pipelining Concepts

So, you recall whatever we have discussed during the course of the last few weeks. We have seen various ways of designing digital circuits and systems, combinational circuits, sequential circuits; very simple circuits, slightly more complex circuits. Now, one thing you should remember whenever we are trying to design high performance systems, where the speed of operations and something called throughput; number of calculations that we are able to do per unit time that becomes important. We often go for a technique called pipelining, so we shall be looking at several examples of pipelining, how we can model pipelines in Verilog and so on.

But in this lecture, let me give you a very brief overview about the basic concepts in pipelining, what it is and how does it give us an advantage in terms of speed up and increasing throughput. So, the topic is Basic Pipelining Concepts.

(Refer Slide Time: 01:33)

What is Pipelining?

- A mechanism for overlapped execution of several input sets by partitioning some computation into a set of k sub-computations (or stages).
 - Very nominal increase in the cost of implementation.
 - Very significant speedup (ideally, k).
- Where are pipelining used in a computer system?
 - **Instruction execution:** Several instructions executed in some sequence.
 - **Arithmetic computation:** Same operation carried out on several data sets.
 - **Memory access:** Several memory accesses to consecutive locations are made.

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

Now, what is pipelining? Pipelining essentially is a method for overlapped execution of several input sets. Like what I mean to say is that, you see whenever you have some kind

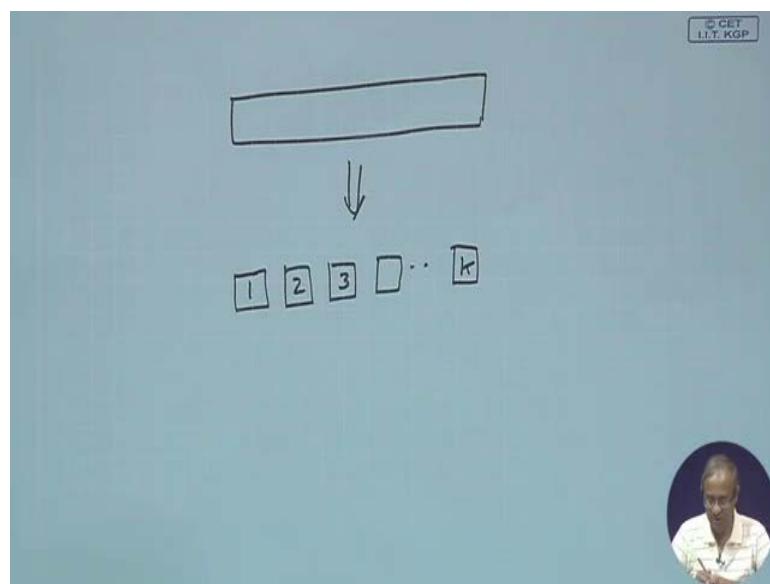
of computation; something you want to compute, there will be some input data you are applying and you are finally getting some result.

Now, your computation maybe such that you have to do the same kind of calculation on a large number of input data, so the data are coming one after the other. So, in that case what we are saying is that the computations that are going on, for the consecutive data items, they are somehow overlapping in execution. Because you see conventionally without pipelining what you will do?

First data comes, we finish our computation, generate the result then only we take the second input. Again we do the computation, generate result then we take the third input, there is no overlap. Now, what I am saying is that before the computation on the first data is complete; we have already started something on the second data. This is something which is called overlapped execution, we shall see how.

So, this overlapping normally we just express in terms of something called sub-computations or stages. Well I shall be explaining through some examples; so, what we are saying is that we have some computation.

(Refer Slide Time: 03:27)



Let us say we represent it like this and as an alternative we divide the computation into smaller pieces called stages. Let us say there are k number of stages; this is how what we are doing the partition. Now, in this kind of a partitioning, the way pipelines are

implemented, the cost of implementation does not increase appreciably; very low increase. But the advantage is that we shall see how it comes, the speed up can be very significant; it will almost be equal to k ; how many times we have divided the computation into the number of partitions.

Now, in a computer system; pipelining can be used to speed up operations in various different places; when instructions are executed, this is called instruction execution. When some computations are going on let us say arithmetic computation addition, subtractions, multiplications there also we can use pipelining to speed up and memory access. When a large number of memory accesses are going on consecutively using pipelining again we can speed up the overall access time of the memory using some efficient technique.

(Refer Slide Time: 04:57)

A Real-life Example

- Suppose you have built a machine M that can wash (W), dry (D), and iron (R) clothes, one cloth at a time.
 - Total time required is T .
- As an alternative, we split the machine into three smaller machines M_W , M_D , and M_R , which can perform the specific task only.
 - Time required by each of the smaller machines is $T/3$ (say).

Let us take a real life example to illustrate how pipelining helps; this is nothing to do with a computer system or a digital system. Let us take this rectangular box indicates some operation that you are doing. What is the operation? This corresponds to machine M ; let us say which can wash, dry and iron clothes one at a time. Suppose, you have manufactured such a machine which will take one cloth at a time, it will automatically wash, dry and iron and it will give you the nicely ironed cloth as the output.

Let us say the total time taken for this W , D and R steps all taken together is T ; capital T . So, if we have N number of clothes, so how much time will it take total to complete for

all the N? For every cloth; we need T, so for N clothes we need N multiplied by T; this we represent as T_1 ; means I have a single partition, one is that 1.

Now, let us say that well you have thought that well instead of a single machine doing everything; because you see any way washing, drying and ironing are 3 different things. Washing involves soap and water, drying involve heater, blowing air and so on and ironing involves pressing and so, on. So, there is nothing which is repeated in the steps; they are as such different. So, you identify that and what you come up with that well, let us now do this.

Let us divide this machine into 3 smaller machines; one which can only wash, second which can only dry, third which can only iron. Well, now you can argue that well because we have divided the functionality and there is very little overlap. The total cost does not increase appreciable; of course, there will be little increase in cost because we have to do a necessary packaging, we have to put it inside a nice cabinet and so on. But whatever they are there inside; the actual electromechanical systems, they are not much different. So, we split the machine into 3 smaller machines M_w , M_d and M_r ; these 3 boxes, which can perform specific tasks.

Now here what happens let us see; now the total time was T. Let us say now; this individual stages; they will take one third time each $T/3$, $T/3$, $T/3$. So, the total time is still T. Now see when a cloth comes, you wash it. After washing is complete; you give it for drying, now you see while drying is going on; this washing machine is free.

Now, the second cloth can be given for washing. Similarly, when the first cloth goes for ironing, this second cloth can come for drying and the third cloth can go for washing. So, I will show you how; if we have a mechanism like this; this is called pipelining. As if there is a pipe; the clothes are flowing through a pipe and there are 3 stages; it first goes to W, then to D, then to R and then it goes out.

So, we will see how this calculation comes; for N clothes the total time taken here will be $(2+N)*T/3$.

(Refer Slide Time: 09:10)

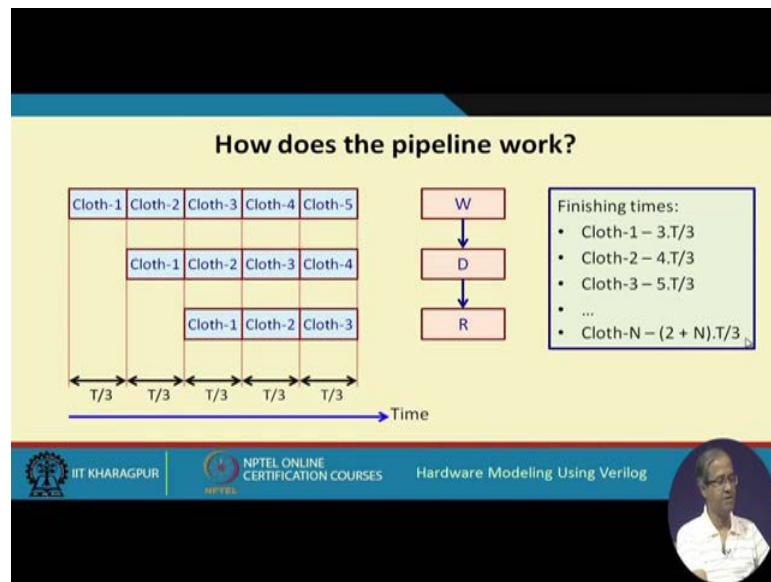
The image shows handwritten mathematical equations on a blue background. At the top left, there are two equations: $T_1 = N \cdot T$ and $T_3 = \frac{(2+N) \cdot T}{3}$. To the right of these equations, there is a vertical double bar followed by the text "Large N" and "≈ 1000". Below the equations, there is a hand-drawn oval containing the equation $T_3 \approx \frac{1}{3} T_1$.

So, in the first case; the time was N multiplied by T and in the second case with 3 stages the time is coming as 2 plus N into T divided by 3 .

Let us assume that the value of N is very large; let us say for example, I am washing 1000 clothes. So, with respect to 1000, you can ignore this 2; 2 is negligible. So, what you can say? You can say that your T_3 is approximately one third of T_1 ; T_1 was NT ; T_3 is approximately NT divided by 3. So, you have gained a 3 time speed up without investing on 3 machines. You see to get a speed up of 3, what you could have done initially that big machine, you could have bought 3 copies.

Which means you would spend money 3 times; 3 times the cost, but now you have not done that; you have divided that big machines into 3 parts. It will involve a marginal increase in cost, but still the performance is improving 3 times; this is the essential idea behind pipelining without increasing the cost appreciably we are getting very appreciable to speed up.

(Refer Slide Time: 10:54)



Now, let us see how this is coming. Let us look at the axis of time; this is washing, drying and ironing. So, cloth-1 comes here; cloth-1 comes for washing. So, it takes $T/3$ time; after $T/3$, cloth-1 comes for drying and the second cloth comes for washing; this requires again $T/3$, then cloth-1 comes for this ironing, cloth-2 for drying, cloth-3 for washing.

Now, after $T/3$, cloth-1 is done, so, cloth-1 you can take out; then cloth-2 comes here; cloth-3 comes here; cloth-4 comes here. After $T/3$, cloth-2 can be taken out, then cloth-3 can take. You see after this pipe is full, after every $T/3$, $T/3$, $T/3$ time you can take out one cloth. So, one cloth per time $T/3$ is being generated as output.

So, for one cloth, the first cloth will take $T/3$, $T/3$, $T/3$; $3 \times T/3$. Second cloth will take one more time 1, 2, 3, 4, $4 \times T/3$; third cloth will take $5 \times T/3$. So, proceeding in this way, the Nth cloth will take $(2 + N)T/3$, that is what we showed in the expression.

(Refer Slide Time: 12:22)

Extending the Concept to Processor Pipeline

- The same concept can be extended to hardware pipelines.
- Suppose we want to attain k times speedup for some computation.
 - Alternative 1: Replicate the hardware k times → cost also goes up k times.
 - Alternative 2: Split the computation into k stages → very nominal cost increase.
- Need for buffering:
 - In the washing example, we need a tray between machines (W & D, and D & R) to keep the cloth temporarily before it is accepted by the next machine.
 - Similarly in hardware pipeline, we need a *latch* between successive stages to hold the intermediate results temporarily.

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



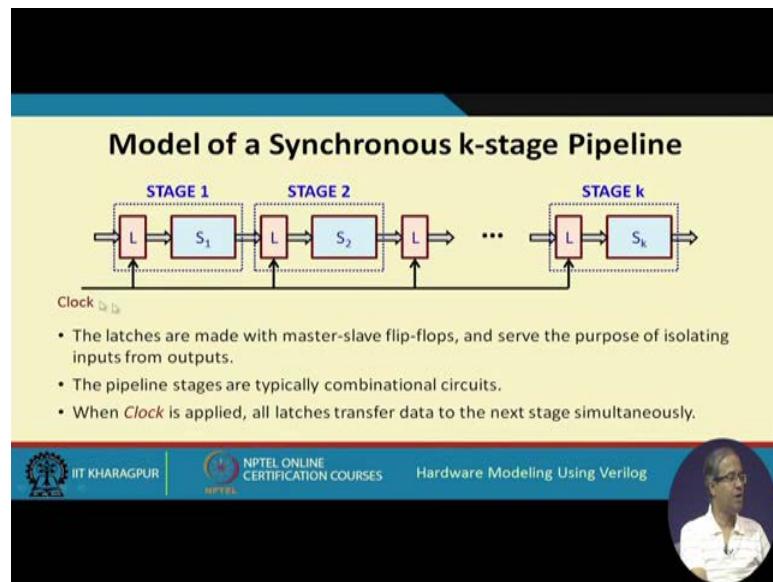
Now, extending this concept to a processor; suppose I am building a computer, I want to speed up instruction execution that is called a processor pipeline. So, instead of executing one instruction at a time; I am dividing the total execution process in two stages and I am overlapping the execution in the same way.

So here again I am just repeating, so, we had two alternatives, one is we can repeat the or replicate the hardware k times, which also will increase a k times, increasing the cost. But, secondly a second alternative we are not replicating rather you are splitting the hardware into smaller pieces; these are called stages, very nominal cost increase. But one thing is required which we have not considered, you see you consider the washing example.

Suppose, there is a washing machine; there is a dryer, suppose washing machine has finished with a cloth, but dryer is not yet ready, it is still drying the last cloth. So, that cloth has to be temporary put in some kind of a tray or a buffer. So, there has to be some trays between the machines which can temporarily stored the clothes before it can be fed to the next machine.

So, this is the need for buffering. So, we need a tray between washing and drying machines and between drying and ironing machines. So, in the same way when we talk about a hardware pipeline, we need a latch or a register between successive stages which can store the latch, store the values, before it can be processed by the next stage.

(Refer Slide Time: 14:28)



This is how a hardware pipeline will look like. So, a total computation I am dividing into k number of stages S_1, S_2 up to S_k ; there are latches between the stages and there is a clock, clock is activating the latches. So, what is the purpose of the latch? You see when the first data comes, the data get stored in this latch and S_1 is working on that.

After S_1 has finished doing the computation; the result it will be storing in this latch. At the same time, the next data will be stored on this latch. So, when S_1 is working on the second data; S_2 is working on the result of the first data. Now, if this latch was not there, then what might have happened is that while S_1 is working on the next data, the output could be changing.

So, the input of S_2 could also be changing, resulting in wrong computation. So, we are using the latch to hold temporarily the previous data, so that while S_2 is processing the data, the data does not change. Similarly, for the other stages and this latches are typically master slave flip-flops, which serve the purpose of isolating inputs from outputs. And the stages S_1, S_2 to S_k , these are typically combinational circuits, we shall see examples later.

So, when clock is applied everything gets shifted to the right by one place. So, S_1 goes to S_2 , S_2 goes to S_3 , S_3 goes to S_4 and so on. So, when clock comes, data in a pipeline will move forward in a lockstep fashion, this is what really happens.

(Refer Slide Time: 16:32)

Structure of the Pipeline

- **Linear Pipeline:** The stages that constitute the pipeline are executed one by one in sequence (say, from left to right).
- **Non-linear Pipeline:** The stages may not execute in a linear sequence (say, a stage may execute more than once for a given data set).

A possible sequence: A, B, C, B, C, A, C, A

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

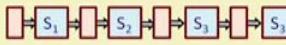
Now, talking about the structure of a pipeline; so, a pipeline can look like this as we have seen, it is called a linear pipeline. The stages are connected in a linear fashion, the output of one goes to the input of the other in exactly in a straight line fashion.

But for more complex systems of course, here we shall not be considering such examples. Pipelines can be non-linear also, meaning that there will be stages, let us say A, B, C but data can flow not necessary from A to B, B to C; data can go from A to C also sometimes, C to B also and C to A also. So, there can be multiple paths, the data can take for example, a possible sequence can be A, B say A, B, C, B, C A, C, A then it comes out from this path. So, this can be a possible sequence of data computation, but these are for more complex cases, forget this for the time being.

(Refer Slide Time: 17:48)

Reservation Table

- The *Reservation Table* is a data structure that represents the utilization pattern of successive stages in a synchronous pipeline.
 - Basically a space-time diagram of the pipeline that shows precedence relationships among pipeline stages.
 - X-axis shows the time steps
 - Y-axis shows the stages
- Number of columns give evaluation time.
- The reservation table for a 4-stage linear pipeline is shown.



A horizontal sequence of four pink rectangular boxes, each labeled S_1 , S_2 , S_3 , and S_4 from left to right. Each box is connected by a small arrow pointing to the next box in the sequence.

	1	2	3	4
S_1	X			
S_2		X		
S_3			X	
S_4				X

IIT KHARAGPUR

NPTEL ONLINE
CERTIFICATION COURSES

Hardware Modeling Using Verilog

Now, you see these are some example pipelines; now how processing is carried out, this is depicted by a data structure which is called a reservation table. What does reservation table show? Reservation table shows the utilization pattern of the stages, like in the various time steps. How are the pipeline stages utilized, in time step 1; which stage you are using? In time step 2 which stage? In time step 3 which stage? and so, on. This is depicted nicely in the form of a tabular fashion in the reservation table.

Now, in the reservation table; the X axis shows the time steps and Y axis shows the stages. The number of columns will give you the total number of stages; that means, the evaluation time. Let us take an example of a fourth stage linear pipeline. Suppose I have a four stage pipeline, at this pink boxes are the latches. The reservation table can be shown like this, these are the stages.

So, in time step 1, you are using X , S_1 ; time step 2, you are using S_2 ; time step 3, S_3 ; time step 4, S_4 . So, it will be a diagonal matrix kind of a thing; X will go like this. For a linear pipeline, the reservation table will always look like this.

(Refer Slide Time: 19:22)

The diagram illustrates a 3-stage dynamic multi-function pipeline. At the top, a block diagram shows three stages, S₁, S₂, and S₃, connected sequentially with feedback connections from S₃ back to S₁ and S₂. Below this, two reservation tables are shown for functions X and Y over 8 time steps.

Function X Reservation Table:

	1	2	3	4	5	6	7	8
S ₁	X					X		X
S ₂		X		X				
S ₃			X		X			X

Function Y Reservation Table:

	1	2	3	4	5	6
S ₁	Y				Y	
S ₂			Y			
S ₃		Y		Y		Y

Logos for IIT Kharagpur and NPTEL are at the bottom left, and "Hardware Modeling Using Verilog" is at the bottom right.

But for a non-linear pipeline, reservation table can be more complex because as we have seen for a non-linear pipeline, there can be feed forward like S₁ to S₃ or feedback S₃ to S₂, S₃ to S₁ such connections.

Let us say, suppose I am computing two functions X and Y. Now in the function X, my pipeline utilization can be like this; first the data comes, it goes to S₁, then S₂, then S₃. From S₃ it again goes to S₂, you see from S₃ to S₂, there is a path; from S₂ again to S₃; S₃ to S₁, S₃ to S₁ there is a path and S₁ to S₃, S₁ to S₃ also there is a path. S₃ to S₁ and finally it finishes, from S₁ there is a path for the data to go out.

And similarly for the other computation Y; say from S₁ to S₃, S₂, S₃, S₁, S₃ then goes out. You see from S₃ also there is a path to go out. So X computation the results are generated here, Y generated here. But here, means it is not very important for you to understand all these things because the examples we shall be taking the reservation tables for the linear pipeline. So, we shall only be considering linear pipelines.

So, in a reservation table multiple cross marks in a row means repeated use of the same stage in different cycles. Like here, S₁ will be used in time step 1 also in time step 5. Contiguous X's in a row, suppose there are 2 X's side by side, which means a stage will be used for more than one cycle. Multiple X's in a column it is not shown in the example which means more than two stages may be active at the same time step.

(Refer Slide Time: 21:28)

Speedup and Efficiency

Some notations:

- τ :: clock period of the pipeline
- t_i :: time delay of the circuitry in stage S_i
- d_L :: delay of a latch

Maximum stage delay $\tau_m = \max \{t_i\}$

Thus, $\tau = \tau_m + d_L$

Pipeline frequency $f = 1 / \tau$

- If one result is expected to come out of the pipeline every clock cycle, f will represent the maximum throughput of the pipeline.

(1)

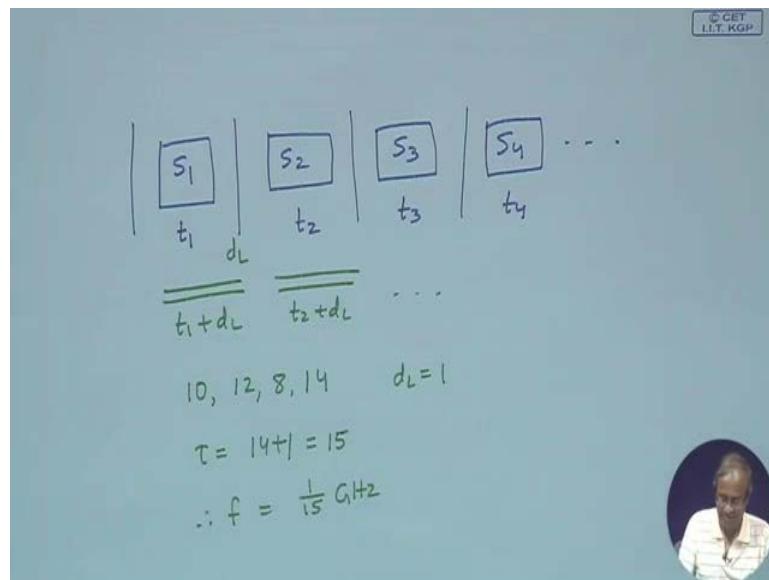


NPTEL ONLINE
CERTIFICATION COURSES

Hardware Modeling Using Verilog

Let us look into some simple calculation; so, for a pipeline what we have said very loosely is that; if there are k number of stages, we expect our speed up to be approximately k , let us see how it comes. So, some notations let τ (τ) denote the clock period of the pipeline, t_i is the time delay for stage S_i .

(Refer Slide Time: 22:02)



Let us say we consider a pipeline like this; we have stage S_1 , S_2 , S_3 and so on and there will be latches in between the stages. So, I am assuming that the time to compute S_1 is t_1 , this is t_2 , this is t_3 , this is t_4 and so on. So, t_i is the time delay of stage S_i and let us say

d_L is the delay of a latch. So, what will be the total delay of a stage? t_1 plus the delay of a latch, so the delay of this will be t_1+d_L ; delay of this will be t_2+d_L and so on.

Now, you see we are having a common clock; so, we will have to look at which stage is the slowest. We find out the maximum of t_i , call it τ_m and $\tau_m + d_L$ (the latch delay) that will be the clock period of the pipeline; clock period cannot be less than this and the pipeline frequency has to be reciprocal of that. Like let us take an example; suppose I have a pipeline with four stages, where the stage delays or let us say 10 nanosecond, 12 nanoseconds, 8 nanoseconds and 14 nanoseconds and the latch delays let us say 1 nanosecond.

So, what will be the maximum clock frequency? Here my τ will be max of the delays which is 14, plus the latch delay, 15. So, my clock frequency can be $1/15$ so many gigahertz because these are in nanoseconds. So, let us move on.

(Refer Slide Time: 24:26)

- The total time to process N data sets is given by

$$T_k = [(k - 1) + N] \cdot \tau$$
 ($k - 1$ τ time required to fill the pipeline
 1 result every τ time after that \rightarrow total $N \cdot \tau$)
- For an equivalent non-pipelined processor (i.e. one stage), the total time is

$$T_1 = N \cdot k \cdot \tau$$
 (ignoring the latch overheads)
- Speedup of the k -stage pipeline over the equivalent non-pipelined processor:

$$S_k = \frac{T_1}{T_k} = \frac{N \cdot k \cdot \tau}{k \cdot \tau + (N - 1) \cdot \tau} = \frac{N \cdot k}{k + (N - 1)}$$
As $N \rightarrow \infty$, $S_k \rightarrow k$

So, what will be the total time that will be required to process N data sets? You see there are k number of stages in the pipe; so, $k - 1$ time steps will be required for the pipe to fill up and after the pipe has filled up; there will be a one output generated every clock. So, $k - 1$ for the pipe to fill up, after that one output every clock, there are N data sets, so for all the results I need N clock cycles after that.

So, you see $k-1+N$; whole into clock period; this will be the total time taken to process; process N data. But, if we assume that we have an equivalent non pipelined processor, where we did not use pipeline then the time taken would be T_1N number of say means $N\tau$, N is the number of data and $k\tau$ we are assuming is the time to process every data.

So, we are ignoring the latch overrides. So, approximately it will be $Nk\tau$. So the speed up will be the time taken on this processor divided by the time taken on the pipeline. So, it will be like this, τ cancels out like this and as I had said as N becomes large. So, you can ignore this k as compared to N and this 1. So, it becomes Nk/N approximately k . So, speed up will be approximately equal to k .

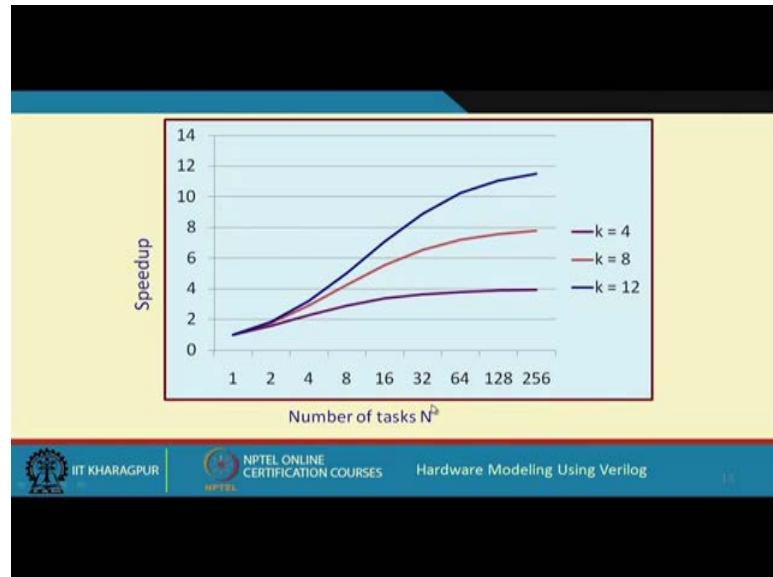
(Refer Slide Time: 26:21)

- Pipeline efficiency:
 - How close is the performance to its ideal value?
$$E_k = \frac{S_k}{k} = \frac{N}{k + (N - 1)}$$
- Pipeline throughput:
 - Number of operations completed per unit time.
$$H_k = \frac{N}{T_k} = \frac{N}{(k + (N - 1)) \cdot \tau}$$

And pipeline efficiency is defined as S_k / k ; you see k is the maximum possible speed up and S_k is the actual speed up. So, you see your actual thing is $(k-1+N)\tau$ and this is N . So, if you just divide this up; this is defined as the pipeline efficiency.

Pipeline efficiency equal to 1 means; you are having an ideal value of k and pipeline throughput says that how many operations are completed per unit time. You are processing N number of data items, you are taking a time T_k . So, N/T_k will be number of operations per unit time; N/T_k . So, these are some simple calculations and this actually shows you the speedup curve; let us say tasks N .

(Refer Slide Time: 27:33)



So, as you plot speedup versus N for different values of k ; you will see that your curve looks like this. For k equal to 4, your speedup increases, but it levels to 4. So, as N increases, it goes approximately to 4. For k equal to 8, it approaches 8; for k equal to 12, it approaches 12. So, this k is the maximum speed up that can be attained in a pipeline.

(Refer Slide Time: 28:09)

Clock Skew / Jitter / Setup time

- The minimum clock period of the pipeline must satisfy the inequality:
$$\tau \geq t_{\text{skew+jitter}} + t_{\text{logic+setup}}$$
- Definitions:
 - Skew*: Maximum delay difference between the arrival of clock signals at the stage latches.
 - Jitter*: Maximum delay difference between the arrival of clock signal at the same latch.
 - Logic delay*: Maximum delay of the slowest stage in the pipeline.
 - Setup time*: Minimum time a signal needs to be stable at the input of a latch before it can be captured.

So, there are some other issues like clock, skew and jitter. You see clock frequency does not only depend on the logic delay and the time you need to set and reset the latch. There are other times called skew, jitter and the setup time.

Skew means the clock signal might get delayed on its way to the different flip-flops and latches. Suppose you generate the clock here, you will have to deliver the clock to different points; the length of the parts can be different. So, there will be different delays that the clock signal will experience before it reaches the different flip-flops or latches. This is what is called skew; maximum delay difference between the arrival of the clocks.

Jitter is well on the same latch; sometimes there will be a variation in delay. Jitter arises due to environmental disturbances like noise in the power supply and so on. Logic delay, we have already seen, the delay in the slowest stage in the pipeline the tau. And setup time, we have mentioned this earlier, setup time is the minimum time that you should make a signal stable at the input of the latch so that the latch can accept it. So, not only the latch delay, you will have to add the latch setup time also to it, these are some small delays that get added.

So, with this we come to the end of this lecture; where we just gave you an overview of what is pipeline? How it works? What are the basic principles behind it? So, now, that we have seen what a pipeline is and how it works. In the next couple of lectures we shall give you some very simple examples of pipelines and illustrate how we can actually quote these designs in Verilog? How we can implement these designs in Verilog and through simulation we shall see that actually it is working in a pipeline.

Thank you.

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

Lecture - 33
Pipeline Modeling (Part 1)

So, in the last lecture, we saw how a basic pipeline works and how it helps in getting a significant degree of speed up over an equivalent non pipelined implementation of a certain piece of hardware. Now, in this lecture, we shall give some illustrative example where you will be able to see given some computation that we want to implement in hardware, how we can implement it in the form of a pipeline, and how we can code the corresponding module in Verilog. So, the title of the lecturer is Pipeline Modelling.

(Refer Slide Time: 01:01)

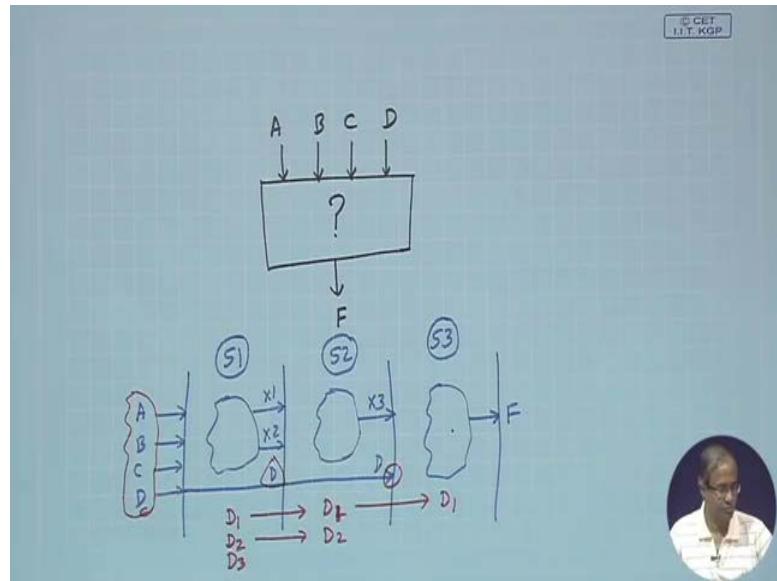
A Simple Example

- We consider the example of a very simple 3-stage pipeline.
 - Four N -bit unsigned integers A, B, C and D as inputs.
 - An N -bit unsigned integer F as output.
 - The following computations are carried out in the stages:
 - a) $S1: \quad x1 = A + B; \quad x2 = C - D;$
 - b) $S2: \quad x3 = x1 + x2;$
 - c) $S3: \quad F = x3 * D;$
 - Point to note:
 - Input D is used in $S1$ as well as $S3$.
 - So the value of D must be forwarded to $S2$ and then to $S3$.

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

So, here we shall be considering a very simple pipeline structure comprising of three stages. The pipeline computes some hypothetical computation, which is defined as follows. Let us suppose there are four numbers A, B, C , and D which are given as input to the computation and finally, we want to get F as output.

(Refer Slide Time: 01:34)



So, what we are saying is that we have a piece of computation. If you treat it as a black box, there are four inputs A, B, C and D and there is one output, the final output that is F. Let us see what is there inside this computation box. So, all these A, B, C and D are multi bit vectors, let us in general call it N. So, we can define a parameter to fix some value of N later. Similarly, F is also an N-bit vector, which is as output. Now, the computation that is done is as follows.

First A and B are added, the result is stored in another temporary variable x1; C - D, D subtracted from C, we store the result in x2; then this x1 and x2 are added to another variable x3, finally, x3 and D are multiplied to get the final result. Now, in order to have this in a pipeline, we observe the dependencies. See, there is an obvious dependency, the first line there are two statements, which are computing x1 and x2, which are used in the next statement. So, the next statement cannot be executed at the same time as you are executing the first two. So, this has to be in the next step.

Similarly, after x3 is computed only then you can compute F. So, quite naturally we have divided this overall computation into three stages, we call them S1, S2 and S3, right. Now let us look at one thing, see three stages S1, S2, S3 we have divided computation into. So, let us represent that like this, this is our stage S1, let us say this is our stage S2, and this is our stage S3. You see there are a few things you need to observe, we shall be showing this using some diagrams later. In the first stage A, B, C and D are the inputs

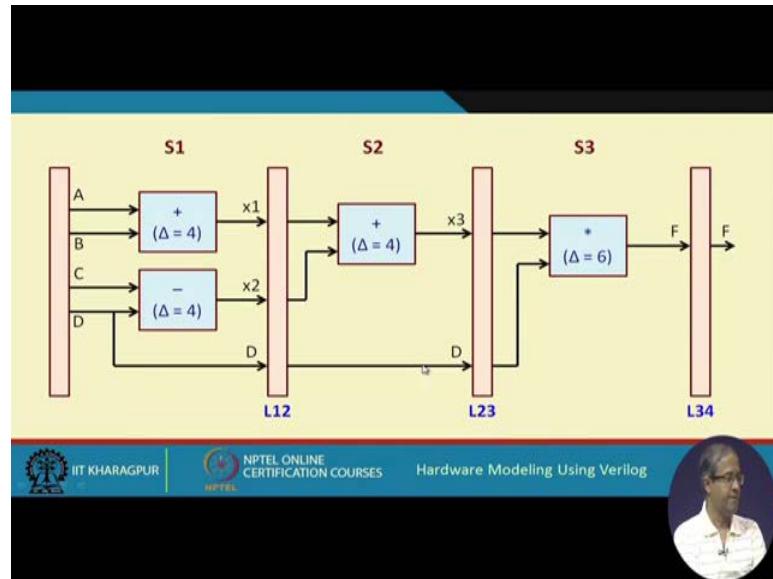
that are coming from outside. So, when you are starting computation in the first stage, you must have the four inputs which are coming in A, B, C and D.

Then you see the first stage is computing the temporary values x1, x2, which will be forwarded all right to the next stage. So, next stage is computing x3. So, from here there will be some computation, this x1 and x2 will be forwarded to the next stage. Look at the third stage, third stage is taking x3. So, what is x3? x3 is computed in stage 2; it takes x1 and x2 as input, it computes another value x3. Now, in S3 whenever you are doing the calculation, you are not only taking S3, you see you are also taking the value of D. So, this value D which was here this also has to be forwarded up to this place. So, that S3 can be computed and finally, and you can get the value F generated.

So, these intermediate buffers or the latches will be used to store this well. For example, D has to be forwarded it. So, you will also have to store D here, you will also have to store D here and then finally, F will be calculating the value. Because you just remember one thing, so when we are giving the first set of data, let us call them D1. So, D1 will first be executing on stage S1, then D1 will be moving to S2, when it is moving to S2. So, S1 will be starting with the next data D2 sorry this is D1. Similarly, D1 will be moving finally to S3, then D2 will be moving to stage two, and the next data will be entering stage one.

Now, the point to notice that when the computation for D1 is entering stage three, so the value of A, B, C, D which were provided for D1 that should be available here, that is why we have forwarded that D value up to this place. So, the same value will get forward because when D1 is doing this computation, this is the D of D1; in the meantime S2 is calculating for D2. So, whatever D is stored here, this is the D value for D2; and whatever D is coming that is the D for D3. So, this is how it goes on. So, here we have an example to illustrate diagrammatically. So, this is the computation that you want to do.

(Refer Slide Time: 07:01)



So, this is the pipeline diagram that we are just assuming, you see in the first stage as I had said we are doing two calculations $A + B$ goes to as x_1 , $C - D$ goes to x_2 . You see in the first stage, the values that are coming are A , B , C and D . So, $A + B$ goes to x_1 , $C - D$ goes to x_2 . So, we are assuming that the delay of both addition and subtraction are 4 units. So, delta is the delay of this block and because D will be needed later as I had said, so the value of D is also forwarded here. In stage S2 we are doing $x_3 = x_1 + x_2$. So, there is another adder, which takes x_1 and x_2 , and it generates x_3 . In the last stage, there is a multiplier, so we assume that the delay is 6 units, it multiplies x_3 and D and generates F as the result.

Now, in this pipeline, what we are saying is that in order that you can execute this operation in a pipeline, because you see what is the essential advantage of a pipeline. When there are a number of data sets on which we want to perform the same kind of computation. Let us say for this example, let us assume, A , B , C , D are vectors, vectors in the sense that it is like an array. There are let say 1000 sets of data which are coming one by one A , B , C , D first value, second value, third value, like that. So, as these values are coming like this computation in a pipeline becomes very efficient.

Now, in order to do it in a pipeline, you will have to insert as I said latches or registers in between the stages, because when the first calculation is over in stage one. And now you move to stage two, these temporary results x_1 , x_2 and also this D , they must be held

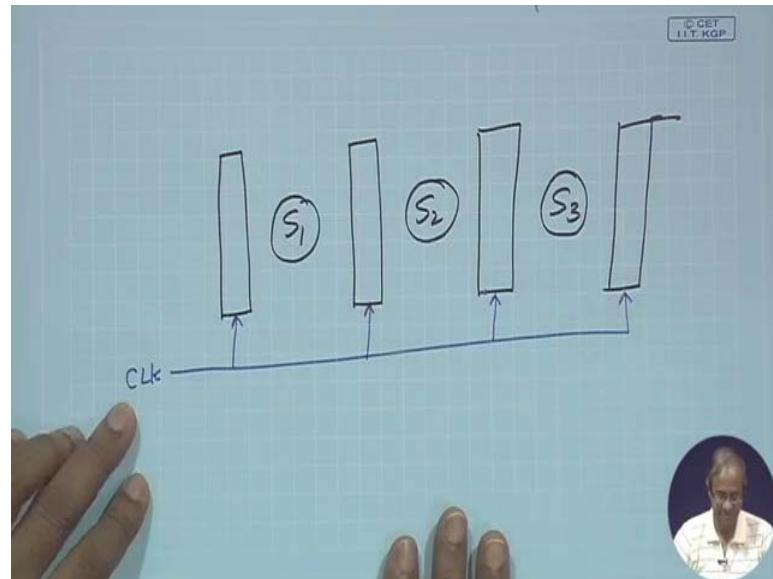
somewhere, so that stage two can start, why they must be held somewhere because the next set of A, B, C, D values are already here. So, those values will make the x1, x2 and D values to change.

So, while S2 is computing in order to ensure that these values do not change, we must have a storage element in between, similarly between every stages you need storage elements like this. So, these storage elements, we refer to as this L12, L23 and L34 and just for convenience, these variables, you see earlier we call them x1, x2 and D like. So, here we are renaming the variables by also mentioning which latch stage, this is L12 between 1 and 2. So, L12_x1, L12_x2, L12_D. Similarly L23_x3, L23_D. You see D is being forwarded. So, the value of D when it reaches here we call it L12_D and when that value reaches here, we call it L23_D; and finally, when we store the value F here, we call it L34_F, because there will be a fourth stage emission after this.

Now, you see let us consider this L12. So, in L12, we need to store three values. So, it is not that x1, x2 and D are three registers that you need to have separately from the latch stage. Rather this latch is nothing but they will be allocated storage here itself for the variables x1, x2, D, they will be allocated storage here itself, x3 and D they will be allocated storage here, and F will be storage allocated here. Like it is something like this, these are the individual register elements, let say or the latches, this latch stores L1 to x1, this stores L1 to x2 and so on.

Now, these three latches taken together we call them as L12. These two taken together we call it L23, and this is L34. So, this is our pipeline diagram, which we want to code in Verilog. Now, when you code in Verilog, we need to remember the name of these variables L12_x1, L12_x2, L12_D because these are the variables we have to generate because all these latches are storage elements that need to be assigned values in synchronism with some clock.

(Refer Slide Time: 12:12)



Now, just one thing we are assuming here for the time being that here there are some latches. I am showing the latches like this; there are three stages. There is a stage S1 here, there is a stage S2 here, there is a stage S3 here. So, what we are assuming is that there is some kind of a clock, which is connected to all the latch stages. like this. Data moves through the pipeline stages in synchronism with the clock. Now, we shall see later that what kind of clocking mechanisms are better for this register stage, we will see it later. For the time being, let us assume that these are common clock, which is feeding all the register stages like this.

(Refer Slide Time: 13:07)

```
module pipe_ex (F, A, B, C, D, clk);
parameter N = 10;
input [N-1:0] A, B, C, D;
input clk;
output [N-1:0] F;
reg [N-1:0] L12_x1, L12_x2, L12_D, L23_x3, L23_D, L34_F;

assign F = L34_F;

always @ (posedge clk)
begin
    L12_x1 <= #4 A + B;
    L12_x2 <= #4 C - D;
    L12_D <= D;           // ** STAGE 1 **
end
endmodule
```

Pipeline Modeling

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

Let see. Now, we want to code this in Verilog, let us see how we do it. So, we define a module, the name of the module is pipe_ex and these are the parameters F is the output, A, B, C, D are the four inputs and of course a clk is there and parameter N defining as 10, so that all the numbers are 10-bit values. So, A, B, C, D are declared as input, 0 up to N-1, clk is also an input, this F is an output of N-bit again. Now, this output F, I have not declared as reg because I am just assigning a value using the assigned statement, but all the temporary values like this L12_x1, L12_x2, L12_D, L23_x3, L23_D, L34_F. Look at the previous diagram, all these values 1, 2, 3, 4, 5 and 6, they are all declared of type register because you will be storing them in the latches, they are all declared of type reg again of size N. And the last value, this L34_F that is nothing, but F, right, so we are simply assigning it to F.

Now, regarding coding as I have said all computation will be done in a synchronous way, always *(posedge clk). So, what will happen when a positive edge of clock comes, for stage S1 whenever a clock comes, A and B has arrived, after delay of 4, this L12_x1 will get the value of the sum, we will have to store it here. Again after value of 4, C - D will be stored here L12_x2; and D will be stored directly here; we are assuming there is no delay. So, in terms of the declaration, we are using non blocking assignment we are writing L12_x1 <= #4 A + B; L12_x2 <= #4 C - D, and this L12_D <= D. And these three statements constitute our stage one of the pipeline.

(Refer Slide Time: 15:51)

```

L23_x3 <= #4 L12_x1 + L12_x2;
L23_D   <= L12_D;           // ** STAGE 2 **

L34_F   <= #6 L23_x3 * L23_D; // ** STAGE 3 **

end

endmodule

```

Then comes the stage two. So, in stage two, what you are doing, just go back once in stage two, we are taking the values of L12_x1 and L12_x2 and we are storing the sum into L23_x3, and D we are moving straight. So, there are two things. This L12_x1 and L12_x2 are added after a delay of 4, it was assigned to L23_x3 and the value of D is simply copied from L12 to L23. And in stage L3, you are doing simple a multiplication with the delay of 6; and the result and multiplication you are doing on L23_x3 and this L23_D, and the result you are storing in L34_F.

So, this is as simple as that you have the pipeline diagram and we have simply coded the pipeline specification in the form of the different stages inside an always blocked, which are triggered with the clock. So, translate in a pipeline schematic diagram into a Verilog code is fairly simple as we have seen. Once we have the diagram you can directly start writing the Verilog code from there, fine.

(Refer Slide Time: 16:51)

```

module pipe_ex (F, A, B, C, D, clk);
parameter N = 10;
input [N-1:0] A, B, C, D;
input clk;
output [N-1:0] F;
reg [N-1:0] L12_x1, L12_x2, L12_D, L23_x3, L23_D, L34_F;

assign F = L34_F;

always @(posedge clk)           // ** STAGE 1 **
begin
    L12_x1 <= #4 A + B;
    L12_x2 <= #4 C - D;
    L12_D <= D;
end

// ** STAGE 2 **

always @(posedge clk)
begin
    L23_x3 <= #6 L12_x1 + L12_x2;
    L23_D <= L12_D;
end

// ** STAGE 3 **

always @(posedge clk)
begin
    L34_F <= #6 L23_x3 * L23_D;
end

```

Alternate way of coding:

- One stage per “always” block.
- Code is more readable.

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

Now, there is another alternate style that we are showing here. So, you can do the same coding in a slightly different way. You see here what we did, we used a single always block right. And inside the always block, the first three statements were stage one, of course, we wrote a comment here just to indicate that these three are stage one, these two are stage two, and this is stage three. But suppose if I do not write the comments then it would be a little confusing to the reader, the person who sees the code to identify that which one is my stage one, which one is my stage two, and which one is stage three. So,

the alternate style what we do, we split or break the always block into three smaller always blocks, one for stage one, one for stage two, and one for stage three, like this. The first part is identical, now this is the always block for stage one, you see there is a separate always block, begin and end, the three statements for stage one are coming here.

Then there is another always block for stage two, and another always block for stage three, because there is a single statement, you do not need begin-end. Now, this style is the equivalent of the previous one, but this code is more readable because you are showing the stages separately and the persons we seeing it will be very easily able to appreciate and understand the stages. And there is another reason that we will see later with another example we shall discuss that there is another need, why we need to break it up into multiple stages.

(Refer Slide Time: 18:39)

The screenshot shows a presentation slide with a yellow header bar. On the right side of the header bar, there is a red-bordered box containing the text "Pipeline Test Bench". Below the header, there is a code editor window displaying Verilog code. The code is as follows:

```
module pipel_test;
parameter N = 10;
wire [N-1:0] F;
reg [N-1:0] A, B, C, D;
reg clk;

pipe_ex MYPIPE (F, A, B, C, D, clk);

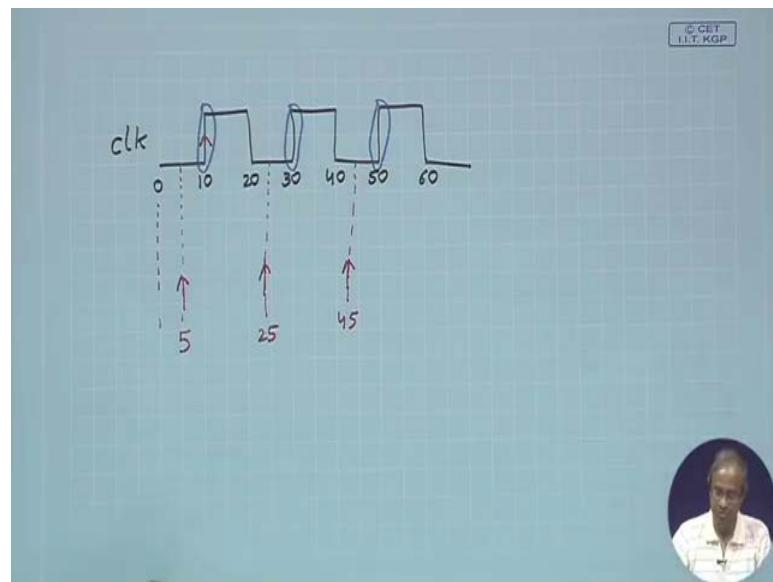
initial clk = 0;

always #10 clk = ~clk;
```

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the text "Hardware Modeling Using Verilog". To the right of the footer bar is a small circular portrait of a man.

Now, let us try to write a test bench for this, now in this test bench, what do we need, we need to supply four values A, B, C and D and after sometime we will be getting the result F available with us. So, here we are instantiating this module, these are the parameters. Now, this A, B, C, D are the inputs to the pipes. So, this will be declared as reg, because here will be assigning values; clock is also reg and F is the output which declared as wire. So, in the similar way these are all 10-bit values. And we are applying a clock, you see the way we are applying clock is 0; and with the delay of 10, we are complimenting it.

(Refer Slide Time: 19:31)



So, the clock will be like this. If this is 0, this will be 10, this will be 20, this will be 30. So, after 10-10 intervals, the clock is toggling, changing state, right, this will be our clock, right.

(Refer Slide Time: 20:01)

initial
begin
#5 A = 10; B = 12; C = 6; D = 3; // F = 75 (4Bh)
#20 A = 10; B = 10; C = 5; D = 3; // F = 66 (42h)
#20 A = 20; B = 11; C = 1; D = 4; // F = 112 (70h)
#20 A = 15; B = 10; C = 8; D = 2; // F = 62 (3Eh)
#20 A = 8; B = 15; C = 5; D = 0; // F = 0 (00h)
#20 A = 10; B = 20; C = 5; D = 3; // F = 66 (42h)
#20 A = 10; B = 10; C = 30; D = 1; // F = 49 (31h)
#20 A = 30; B = 1; C = 2; D = 4; // F = 116 (74h)
end
initial
begin
\$dumpfile ("pipel.vcd");
\$dumpvars (0, pipel_test);
\$monitor ("Time: %d, F = %d", \$time, F);
#300 \$finish;
end
endmodule

The screenshot shows a Verilog code editor with two main sections. The first section is an 'initial' block containing a series of assignments and delays. The second section is another 'initial' block containing a '\$dumpfile' command to dump waveforms, a '\$dumpvars' command to define dumped variables, a '\$monitor' command to monitor the 'Time' and 'F' signals, and a '#300 \$finish' statement. At the bottom of the screen, there are logos for IIT Kharagpur and NPTEL, along with the text 'NPTEL ONLINE CERTIFICATION COURSES' and 'Hardware Modeling Using Verilog'. A small video window in the bottom right corner shows a professor speaking.

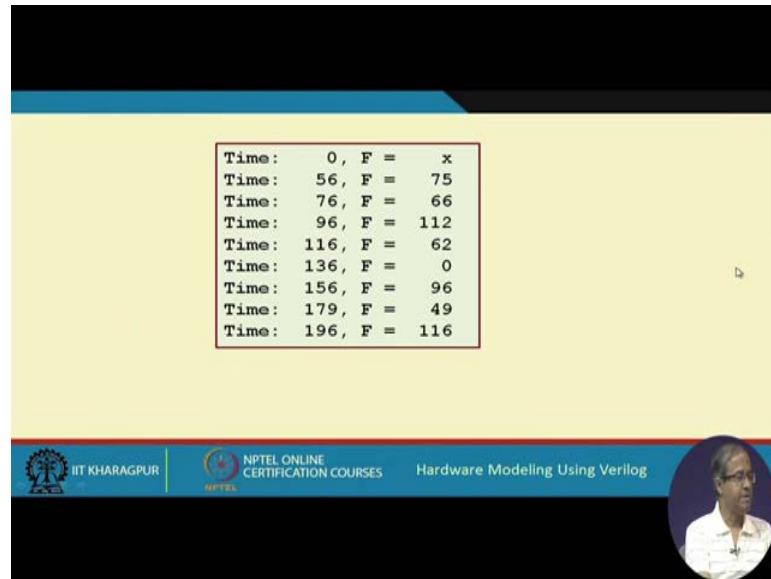
Next what we do, we apply the inputs. Now, let us see what we do, the first set of inputs there is another initial block, so it starts with time 0 again; we give a delay of 5 and apply the first set of inputs. What does this mean? So, you see here is a timing diagram, clock starts with 0. So, time 5 is somewhere, right, this is 5. So, it is here we are applying the

first input. Why, because when the next clock edge comes, this input is already ready. So, we can start processing this input from the next clock edge and after this we are giving 20-20 gaps, because this was 5, next it will be 25, it is here. Next it will be 45, it is here. So, my first input is coming here, second input will come here, third input will come here and so on.

The reason is very clear that if I apply the first input here, then the first clock edge will be able to capture this input. If I apply the second input here, this second clock edge will be able to capture it, similarly, the third clock edge will be able to capture it. So, in this way we are applying the inputs. So, there are a set of 8 sample inputs we have applied various values of A, B, C, D. So, initial delay of 5 and then 20, 20, 20, gaps. So, here also by the side I have shown the expected value of the result, for this thing you see A + B is 22, C - D is 3. So, 22 and 3, what we are doing; you see that computation once more, you are doing the addition then subtraction then you are adding this x_1 and x_2 , then you are multiplying it with D.

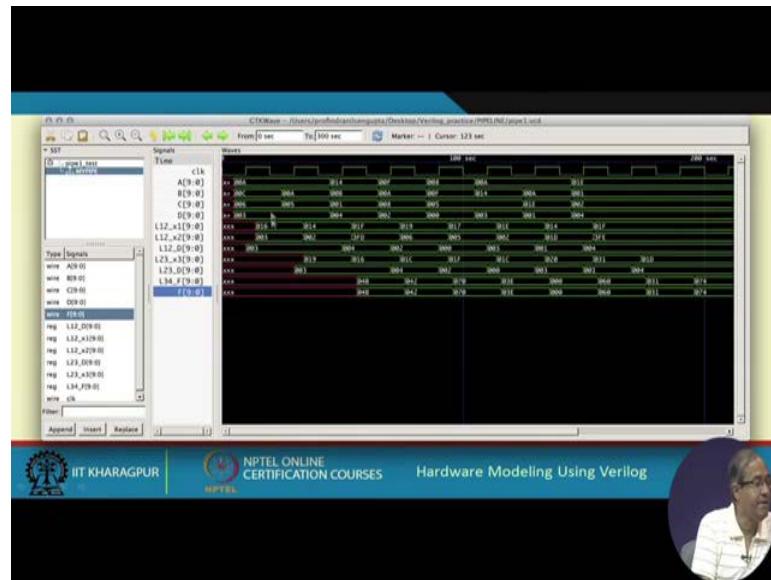
So, 22 and 3, you add them up, it becomes 25, 25 you multiply with D, it become 75, like that. These are the values in decimal and side by side I am also showing the hexadecimal values because in the timing diagram that will be generated by gtkwave, the values will be shown in hexadecimal. Now, in this initial block, we are specifying the dumpfiles and dump variables and will be monitoring the time and the values of F. So, time and F will be printed. So, we will be doing simulation up to certain maximum time let say 300, but here we do not need 300 because we have only; how many 3, 4, 5, 6, 7, 140, by 150 should be over but anyway we are doing up to 300.

(Refer Slide Time: 22:52)



Now, if you run the simulation, the simulation result comes like this. The time is whenever F changes, time is whenever F changes and you see the F value the 75, 66, 102 the same values are coming. So, whatever the expected values, the same values are being generated. This is the final output.

(Refer Slide Time: 23:20)



And if you see the output of the gtkwave the timing diagram here, you can see what is happening. You see the first row shows the clock; this A, B, C and D are being applied; you see 00A,C, 6 and 3; that means, 10, 12, 6 and 3 and the values of x1 and x2 are

getting calculated. $10 + 12$ is 22, it is 16 in hexadecimal, it is 22. $6 - 3$ is 4, this is 3; these two are again added 22 and 3 is 25, it is 19 in hexadecimal; this is L23_x3, and this is multiplied by D. Finally, you get here 48; that means 16×4 is 64, and B, 75. So, this is your final result, and this is assigned to final F. So, you see 4b, 42, 70, 03, these are the actual values are coming 4b, 42, 70, 03, I have shown in hexadecimal, the same values are coming in the simulation output. And it is coming at every clock as is expected in a pipeline.

So, the advantage of the pipeline is that you are not waiting for one computation to finish before applying the next input. So, you are continuously applying input one after the other every clock and when the pipeline is full output will also be generated ones per clock cycle that is why the throughput will be improved to a great extent. This is illustrating by this example, this diagram. This timing diagram as it shown, you can see from here that the output is also generated ones per cycle, first output, second output, third, fourth, fifth, sixth. But initially there is a delay for the pipe to fill up, but once the pipe gets filled up, the outputs are generated one per cycle that is the advantage of pipeline.

(Refer Slide Time: 25:24)

A Warning

- In this example, we have used a single phase clock to load all the inter-stage registers in the pipeline.
- In a real pipeline, this may lead to race condition.
- Possible solution:
 - Use master-phase flip-flops in the registers.
 - Use a two-phase clock to clock the alternate stages.
- We shall illustrate the two-phase clocking approach in the next example.

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

Now, there is an important point to note that in the example that I have shown here we have used a single clock and that clock is used to activate all the latches. So, we have used a single statement always @ (posedge clk). So, it is actually a flip-flop, a register we

are implementing at the positive edge of the clock, we are just activating the latch stage. That same thing is happening for all the stages, right.

Now, the point to notice that this is actually a warning that in a real pipeline, this may lead to a race condition, see what we are saying is that in order that correct operation is achieved in a pipeline, all the register stages or latch stages, if you are supplying a common clock must be master-slave type, there must be a master stage, a slave stage and so that inputs and outputs are perfectly isolated. But if you do not use master-slave flip-flops or latches a single stage then when the input changes, the output also changes which might affect the second stage that is the danger here. Of course, in this example, you could not see that, but in the next example we shall see in our next lecture, there we will do this isolation in a very proper way.

So, what are the possible solution as I said first one is to use a master-phase flip flop, and the second one is to use a two-phase clock to clock the alternate stages. So, we shall be illustrating this in the next example, which will be taking up in our next lecture. So, in this lecture, we have taken a simple example of a computation, and showed how we can map it to a pipeline structure, and how we can code it in Verilog.

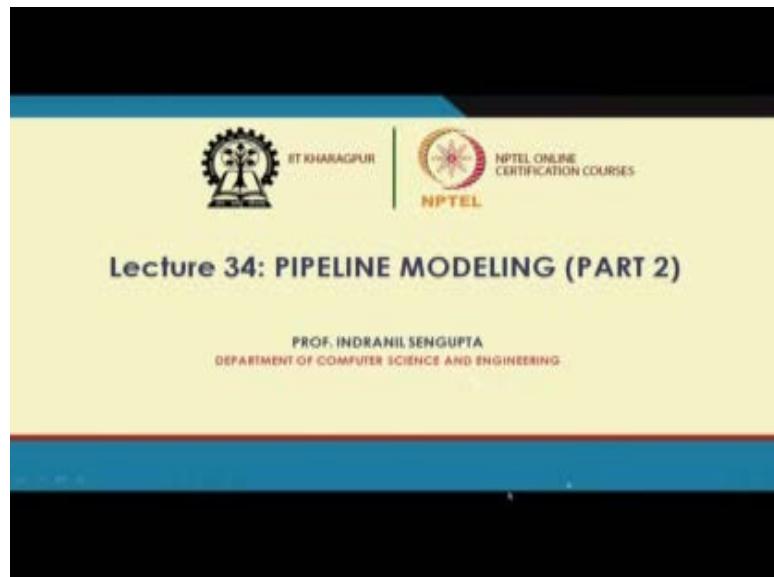
Thank you.

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

Lecture - 34
Pipeline Modeling (Part 2)

So, in our last lecture, we had seen an example where we can map a given computation into a pipeline and implemented in Verilog. In this lecture, we shall be taking up another example slightly more complex and look at a more conservative and a more proper way of clocking that will lead to correct operation without any race condition possibilities.

(Refer Slide Time: 00:51)



(Refer Slide Time: 00:56)

A More Complex Example

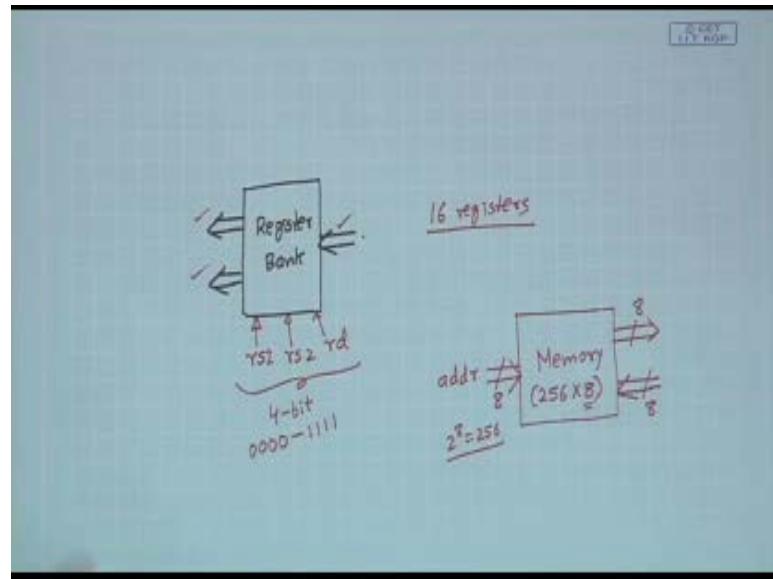
- Consider a pipeline that carries out the following stage-wise operations:
 - Inputs:* Three register addresses (*rs1*, *rs2* and *rd*), an ALU function (*func*), and a memory address (*addr*).
 - Stage 1:* Read two 16-bit numbers from the registers specified by "*rs1*" and "*rs2*", and store them in *A* and *B*.
 - Stage 2:* Perform an ALU operation on *A* and *B* specified by "*func*", and store it in *Z*.
 - Stage 3:* Write the value of *Z* in the register specified by "*rd*".
 - Stage 4:* Also write the value of *Z* in memory location "*addr*".

IIT KHARAGPUR NITEL ONLINE CERTIFICATION COURSES Hardware Modeling Using Verilog

So, this is the second part of our lecture in pipeline modeling. Now, here the example that we take is actually a 4 stage pipeline example. Now here, you do not try to understand that why we are doing this. It is just some example stages and the reason we have taken this example is that later on; we shall be looking at the complete design of a processor. Processor means a central processing unit or a CPU; how it can be implemented in Verilog, in a pipeline fashion, there some of the concepts that I will show in this example will be used. So, just to get a feel; so, we have incorporated a few of the complex blocks in our design, here, let us see what we have done.

Here we are assuming that there is a register bank and ALU. So, I will explain this, first thing to notice that in this problem, I am assuming that there is a register bank. Register bank we have seen that you can have multiple registers stored here and you are assuming that there are 2 read ports and 1; sorry, 2 read ports and 1 write port, yes.

(Refer Slide Time: 02:01)



Now, in order to activate them, I am assuming that you have the control signals; one is read source 1, read source 2 and register source 1 (rs1), register source 2 (rs2) and register destination (rd). So, depending on rs1 and rs2; the corresponding registers will be read here and here and depending on rd, the values applied here will be written into the corresponding register

Now, we shall see how this register bank, let us say if there are 16 registers in this register bank, then all this rs1, rs2 and rd will be 4-bit numbers because you can specify one out of 16 registers in 4-bits, 0000 to 1111, register number 0 up to register number 15.

Similarly, we are assuming that we have a memory. This is a very simple example of a memory, we are assuming that there are 256 words in the memory and each word is of 8-bits. So, since there are 256-bits in the memory, there will be an address, this will be 8-bits in size because we have 8-bits of data. So, if there are separate data out and data in lines; they will be 8-bits and 8-bits. Also this is 8-bits because 2^8 is 256 that is why you need to 8-bits of address and this is 8 because data size is 8 and addition you have the control signals; read, write, all those are there.

So, with this, we are stating the problem; what we are trying to do? So, the inputs to our pipeline will be 3 register addresses that will be corresponding to the register bank. There is also an arithmetic logic unit, an arithmetic functional block, where we specify some

function; what kind of operation you want to do and for the memory we supply a memory address.

So, you see here; there are 3 functional units, we have talked about a register, we have talked about a memory and also there is an arithmetic logic unit; arithmetic and logic unit. So, ALU will be taking 2 input data, it will carry out some computation and what computation, that is determined by a control word called function, right.

So, these 3 kind of blocks are required in this pipeline implementation. So, the inputs will be, what are the register addresses; what is the ALU function; we want to do and what is the memory address and what are the computations that we have to do. Let us see in stage one; we read 2 numbers from the register bank, we assume that all numbers are 16-bit numbers in the registers; registers are 16-bits depending on whatever you have given in rs1 and rs2. We read two 16-bit numbers and store them in 2 temporary registers A and B.

Stage 2; we perform the ALU operation, we operate on A and B and what operation, it depends on this func; what function; it can be addition, subtraction, whatever I have specified and after the ALU operation is done, the result is sorted again in a temporary register Z.

In stage 3, what we do, the result Z, we are writing back into the register bank depending on whatever we have given in rd, destination register and in stage 4, the same data Z, we are also writing into memory at address addr, right.

(Refer Slide Time: 07:16)

The Assumptions

- There is a register bank containing 16 16-bit registers.
 - 4-bits are required to specify a register address.
 - 2 register reads and 1 register write can be performed every clock cycle.
 - Register addresses are "rs1", "rs2", and "rd".
- Assume that the memory is organized as 256×16 .
 - 8-bits are required to specify memory address.
 - Every memory location contains 16 bits of data, which can be read in a single clock cycle.
 - Memory address specified as "addr".

IT KHARAGPUR NITTEL ONLINE CERTIFICATION COURSES Hardware Modeling Using Verilog

So, this is just an assumption we are making, these are the stages of computation. Let us see, the assumption that we have made is that the register bank, we have said it consists of 16 16-bit registers because there are 16 registers, we need 4-bits to specify the register address. So, rs1 and rs2 and rd will be 4-bits and we assume that 2 register reads and 1 register writes are possible every cycle; there are 2 read ports and 1 write port.

Well and because the data size is 16-bits we have assumed and the memory also has to be 16-bit word; here we have said that the memory is 8-bits, but actually if to make it compatible; this has to be 16-bits. So, let us consider these to be 16, right; data bus size will be 16.

So, for this 256 word memory; this 8-bits will be required to specify the memory address and this memory address is specified by this addr. So, addr is an 8-bit quantity and the data will be 16-bits in size.

(Refer Slide Time: 08:29)

• The ALU function is selected by a 4-bit field "func", as follows:		
0000: ADD	0001: SUB	0010: MUL
0011: SELA	0100: SELB	0101: AND
0110: OR	0111: XOR	1000: NEGA
1001: NEGB	1010: SRA	1011: SLA

Now, the ALU functions; we assume are as follows because func is a 4-bit field and this 4-bit; well, we have not specified for all possibilities there are 0000 up to 1011 means there are 12 functions which are defined; let us see what functions are there.

(Refer Slide Time: 08:57)

Diagram of an ALU block with inputs A and B and output Z.

Handwritten list of 12 functions:

- ADD: $Z = A + B$
- SUB: $Z = A - B$
- MUL: $Z = A * B$
- SELA: $Z = A$
- SELB: $Z = B$
- AND: $Z = A \cdot B$
- OR: $Z = A \vee B$
- XOR: $Z = A \wedge B$
- NEGA: $Z = \sim A$
- NEGB: $Z = \sim B$
- SRA: $Z = A \gg 1$
- SLA: $Z = A \ll 1$

A bracket groups the first 12 functions as "12 func".

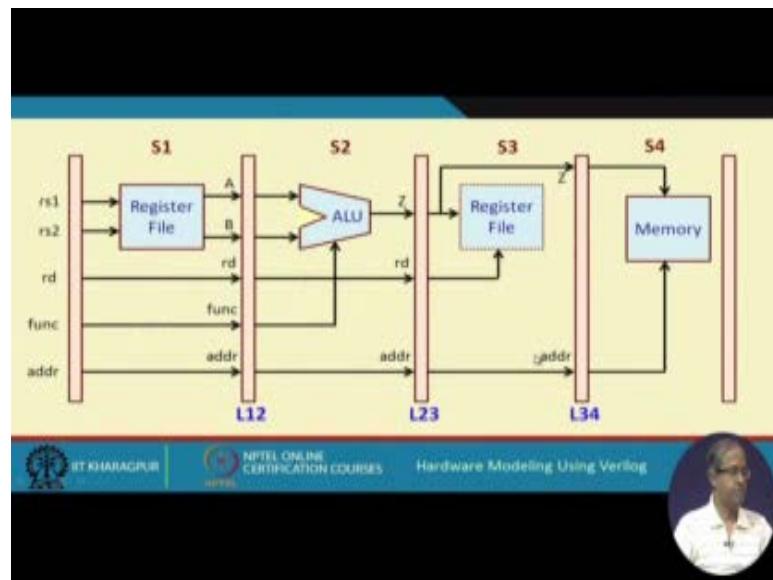
ADD means, let us look at it like this; suppose, this is my ALU, the 2 inputs are A and B and the output is Z. So, let us see one by one, ADD, 000 means ADD, ADD means $Z = A + B$; 0001 means SUB, SUB means $Z = A - B$; 10 means MUL, MUL means multiply,

MUL means $Z = A * B$, then fourth is select A, SELA means if you select SELA, then the value of A will be going to the output, you are selecting A.

Similarly, there is an option for selecting B, if you use SELB, then output will be B, then you have an option AND and OR and XOR. So, AND, OR and XOR; so, for AND Z will be bit by bit, $A \& B$; OR it will be $A | B$. And for XOR, it will be again bit by bit exclusive OR, $A ^ B$; then you have a function NEGA and NEG B. NEGA means it is just NOT bit by bit negation (\sim) of A and NEG B means bit by bit negation (\sim) of B, right and the last 2 functions are shift right A (SRA) and shift left A (SLA).

So, if you use SRA, this will mean you take A and shift it right by one position and if it is shift left A, Z it will be A shift left by 1 position. So, these are the 12 functions that the ALU supports and this is controlled by the input func, 4-bit value, fine.

(Refer Slide Time: 11:42)



Now, this is our overall pipeline diagram based on whatever we have said, let us see this $rs1$, $rs2$, rd , $func$ and $addr$; these are our inputs. So, we have a register file in stage one based on whatever you have given in $rs1$ and $rs2$, we are accessing the registers and values are read into A and B, nothing else is has been done in S1.

In S2, we are doing some functions on these 2 values which have read from the register bank, register file A and B which are coming, they will be the inputs of the ALU and the

function that we have given here that will be forwarded and that function will be controlling the ALU operation and the result will be Z.

In stage S3, we are writing in the register file, we are showing it as dotted because this is not a separate register file. It is the same register file, I am just showing it that it is being written here, that is why dotted.

So, what we are writing, we are writing the value of Z and where we are writing, that is rd, register destination. So, this has to be forwarded up to here and the last stage, we have the memory write, where, what we want to write, the value of Z. So, Z has to be forwarded here, from here to here and address has to be forwarded all the way from here down to here. So, it is a 4 stage pipeline where these operations are going on, right.

(Refer Slide Time: 13:22)

Clocking Issue in Pipeline

- It is important that the consecutive stages be applied suitable clocks for correct operation.
- Two options:
 - a) Use master/slave flip-flops in the latches to avoid race condition.
 - b) Use non-overlapping two-phase clock for the consecutive pipeline stages.

clk1 | 5 5 5 5 5 5 5 5 5 5
clk2 |

IT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES Hardware Modeling Using Verilog

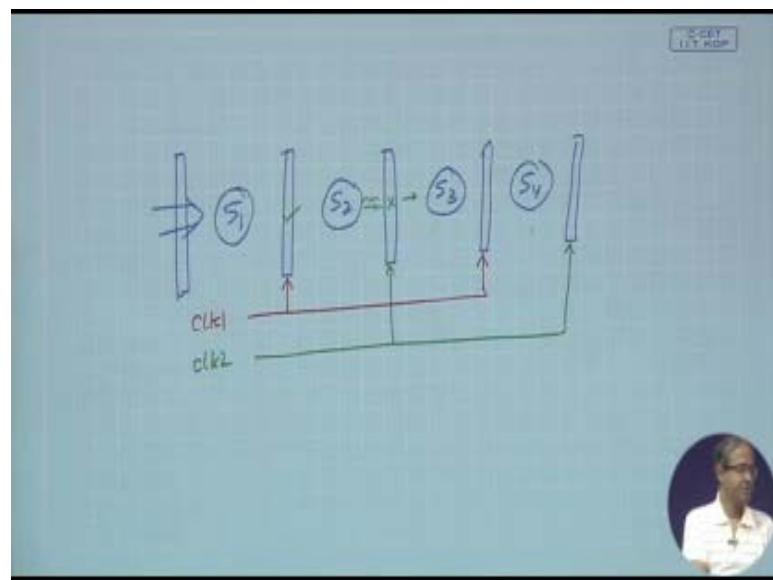
So, let us see how we can code it in Verilog, but before that there is an important clocking issue that I have said during the last lecture that we have to apply clock in a proper way, we mentioned that the two of the safe way is used to either use master-slave flip-flops or use non-overlapping clocks for the consecutive stages. So, if we use non-overlapping clocks, then you can also use latches rather than clocks which are triggered by edges.

Non-overlapping, an example is shown here clk1 and clk2; you see clk1 is high here, again it is high here, again it is high here. So, the clock period is 20, from here to here 20

and clk2 is high, when clk1 is not high and there is a period in between where both the clocks are inactive, both are 0. So, there are 2 clocks, sometimes clk1 is high, sometimes clk2 is high, but they are not overlapping, they are non overlapped and in order to take care of variable delays like clocks queue, there is also a gap in between. This is a very safe kind of a clocking scheme where we have non-overlapping clock with the safe margin in between, this is what is meant by 2 phase clock.

Now, in a pipeline, what we are intending to do is as follows.

(Refer Slide Time: 14:58)



In this example, we have 4 stages, right, this is stage S1, then stage S2, then stage S3, then stage S4, this we are ignoring; here, we are just supplying the inputs. Now out of this 4 latches; what we are saying is that latches or flip-flops whatever we say. That we will be clocking, these two, by clk1 and we will be clocking these two by clk2; which means we are alternately clocking the latches using the 2 phases.

Because we are doing that there is no scope on an overlap because when this latch is on it is guaranteed that this latch is not on. So, whatever S2 is computing, it can never reach here because this latch is off; because of this non overlap in nature, this kind of complete isolation of one stage from the other can be achieved.

So, in our Verilog test bench that we see or in the implementation also, we shall be assuming that we are having this kind of 2 phase clocking. Let us look at the Verilog code now.

(Refer Slide Time: 16:30)

The screenshot shows a presentation slide with a yellow header and footer. The main content is a Verilog module definition:

```
module pipe_ex2 (Zout, rs1, rs2, rd, func, addr, clk1, clk2);  
    input [3:0] rs1, rs2, rd, func;  
    input [7:0] addr;  
    input write, clk1, clk2;      // Two-phase clock  
    output [15:0] Zout;  
  
    reg [15:0] L12_A, L12_B, L23_Z, L34_Z;  
    reg [3:0] L12_rd, L12_func, L23_rd;  
    reg [7:0] L12_addr, L23_addr, L34_addr;  
  
    reg [15:0] regbank [0:15]; // Register bank  
    reg [15:0] mem [0:255];   // 256 x 16 memory  
  
    assign Zout = L34_Z;
```

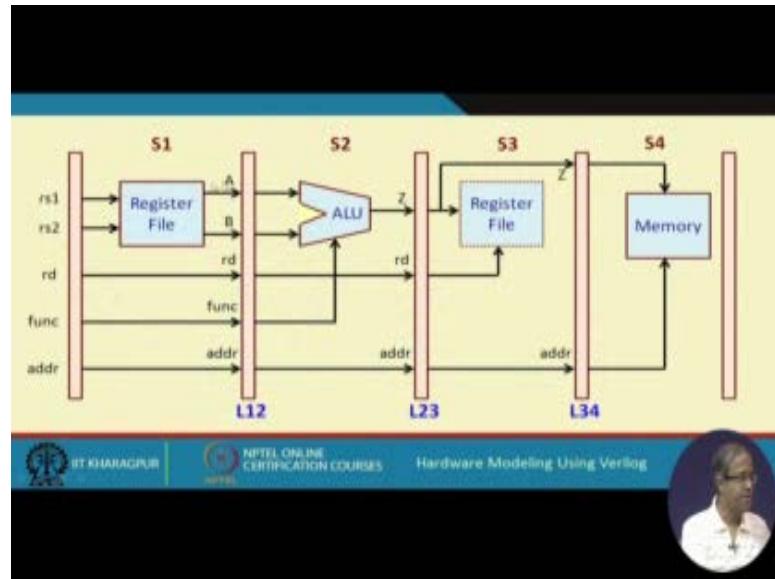
A red box highlights the word "Pipeline" and "Modeling" in the top right corner of the slide content area.

This is our pipeline description. So, we have the final output Zout; Z out again, this is a 16-bit quantity, we are assigning it from the final value of Z and rs1, rs2, rd, func, addr, there are all inputs and of course, there are 2 clocks, clk1, clk2.

Register select and also the function select, these are all 4-bit quantities and memory address because there are 256 words, it is 8-bits and write of course, it is not required actually, write can be there, clk1, clk2.

Now, these are the intermediate register variables we are defining, you see this A and Z these are holding the results, these are 16-bit quantities but rd and func these are 4-bit and address is 8-bits, you see here you just go back to that diagram.

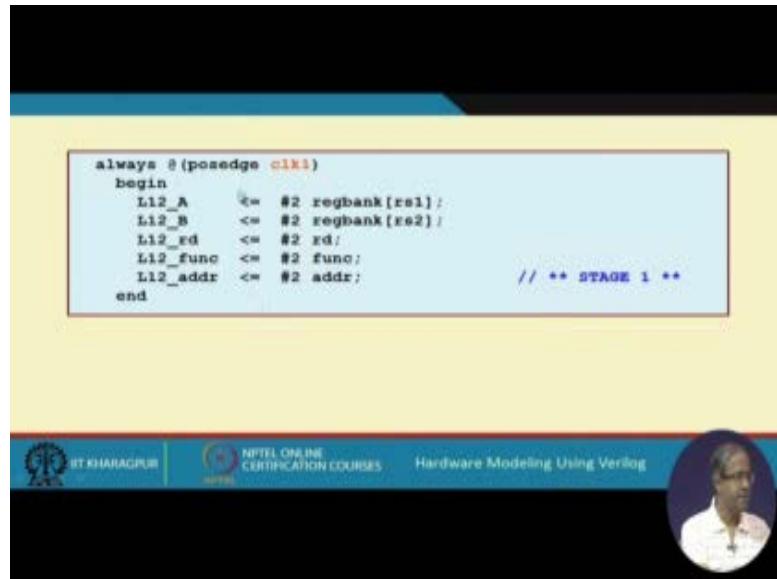
(Refer Slide Time: 17:42)



A, B, rd, func and addr are to be stored in L12; Z, rd and addr in the L23 and Z and addr in L34. So, we have defined so many variables and the convention is as same as we followed in the last lecture. It will be L12_A, L12_B, L12_rd and so on, right. So, same convention we have followed.

And we have defined a register bank and we have defined a memory separately, both declarations are similar, both are defined as a 2-dimensional array of registers in fact. So in the first case, we have defined that there are 16 registers, each of 16-bits. In the second, we have declared a memory of 256 words, each of 16-bits and the final L34_Z which is been computed that is assigned to the final output Zout.

(Refer Slide Time: 18:50)



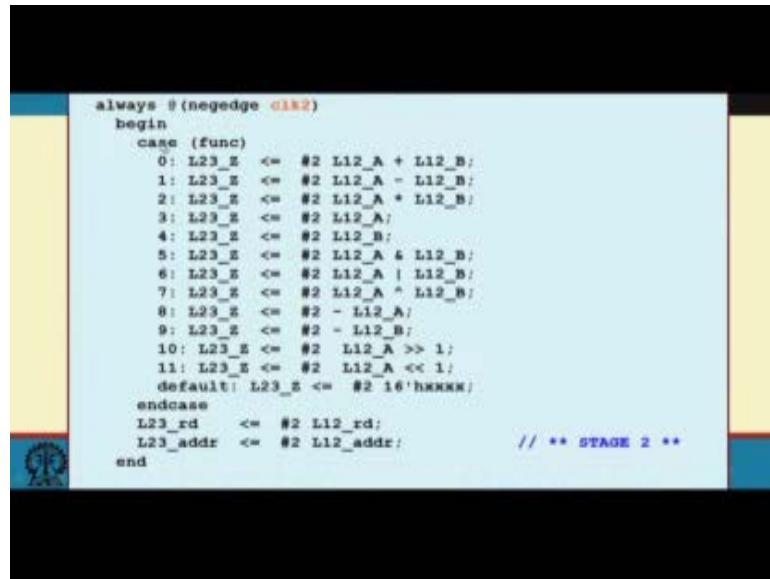
Now, you see; now how we have done; just I am showing the overall thing; you see this is the first stage, this is activated by clk1; this is the second stage, stage 2 activated by clk2; stage 3 activated by clk1; stage 4 activated by clk2. So, we are alternately applying clk1 and clk2 to the successive stages clk1, clk2, clk1, clk2 like that, this is the first thing we have done here.

And if we look at the stage definitions, it is straight away it follows from whatever we have done, like you see at the diagram once more, first stage what we are doing? This A, B, we are reading from the register file, from address rs1 and rs2; rd we are forwarding; func we are forwarding; addr we are forwarding; now see I have done exactly that.

This L12_A is reg bank; rs1 reading from the register ban; L12_B reg bank; rs2 read, we are forwarding func; we are forwarding addr, we are forwarding and here we are assuming that everything takes a uniform delay. So, all 2, 2, 2; we have shown here.

Second stage, we have an ALU and some signals are forwarded, like you go back, once more, you see rd is forwarded, addr is forwarded, remaining thing is the ALU function, it takes A, B and func and generates Z.

(Refer Slide Time: 20:27)

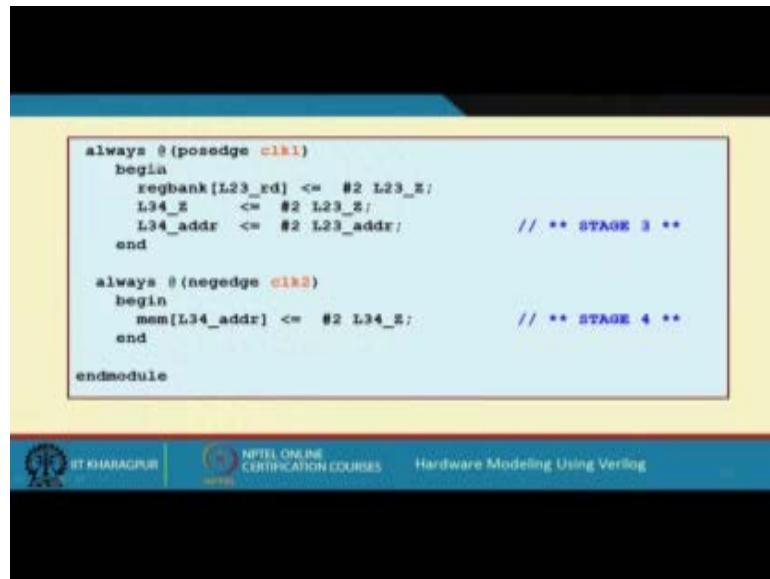


```
always @(posedge clk1)
begin
    case (func)
        0: L23_Z <= #2 L12_A + L12_B;
        1: L23_Z <= #2 L12_A - L12_B;
        2: L23_Z <= #2 L12_A * L12_B;
        3: L23_Z <= #2 L12_A;
        4: L23_Z <= #2 L12_B;
        5: L23_Z <= #2 L12_A & L12_B;
        6: L23_Z <= #2 L12_A | L12_B;
        7: L23_Z <= #2 L12_A ^ L12_B;
        8: L23_Z <= #2 ~ L12_A;
        9: L23_Z <= #2 ~ L12_B;
        10: L23_Z <= #2 L12_A >> 1;
        11: L23_Z <= #2 L12_A << 1;
    default: L23_Z <= #2 16'hXXXX;
    endcase
    L23_rd <= #2 L12_rd;
    L23_addr <= #2 L12_addr;          // ** STAGE 2 **
end
```

So, the ALU function is here case (func). So, if it is func is 0, that means 0000, then you do addition, you add store in Z; if it is 1, you do subtraction, multiplication, select A, select B, AND, OR, XOR, \sim A, \sim B, shift right A, shift left A and if it is any other value, these are undefined, then Z will be xxxx and the other two values we are forwarding.

This is the definition of state 2, stage 3 is very similar, in stage 3 what we are doing, again let us go back in stage 3, we are writing into the register file and in stage 4, we are writing into the memory.

(Refer Slide Time: 21:24)



```
always @(posedge clk1)
begin
    regbank[L23_rd] <= #2 L23_Z;
    L34_Z <= #2 L23_Z;
    L34_addr <= #2 L23_addr;          // ** STAGE 3 **
end

always @(negedge clk2)
begin
    mem[L34_addr] <= #2 L34_Z;      // ** STAGE 4 **
end

endmodule
```

So, we are doing exactly that here. In stage 3, so, whatever is the value of Z, you have computed, we are writing into reg bank where the address is L23_rd and Z we are forwarding; addr we are forwarding, this will be requiring for the memory write and in the memory lastly, the value of Z which has been forwarded that is been written into memory L34_addr.

Just one thing here we have mentioned an input write, but actually write is not required, this write you can emit, this here, there is no write, fine, there is no write.

(Refer Slide Time: 22:18)

```

module pipe2_test;
    wire [15:0] Z;
    reg [3:0] rs1, rs2, rd, func;
    reg [7:0] addr;
    reg clk1, clk2;
    integer k;

    pipe_ex2 MYPIPE (Z, rs1, rs2, rd, func, addr, clk1, clk2);

    initial
        begin
            clk1 = 0; clk2 = 0;
            repeat (20) begin
                #5 clk1 = 1; #5 clk1 = 0;
                #5 clk2 = 1; #5 clk2 = 0;
            end
        end

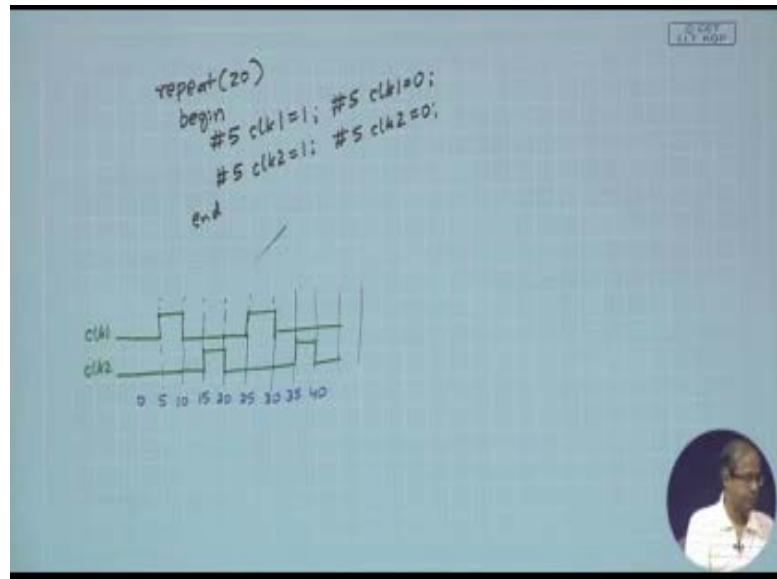
    initial
        begin
            for (k=0; k<16; k=k+1)
                MYPIPE.regbank[k] = k;
        end
endmodule

```

Now for this pipeline, let us try to write the test bench. So, here we have instantiated our pipe, this is the output; rs1, rs2, rd, func, addr are the inputs and the two clocks. So, these are declared as reg because we will be initializing them 4-bits. This addr is also 8-bits reg; clk1, clk2 is also reg; but Z is the output, let us say it is a wire 16-bits and we have declared an integer k for the purpose we will see.

First thing is that let us see how we are initializing the clock, you see clock is 0; clk1, clk2, both are initialized 0 at time 0 and we are generating 20 pulses because 20 is enough for this example, we have given repeat (20). So, what you are doing? So, I am just writing the code and explaining; what is happening?

(Refer Slide Time: 23:17)



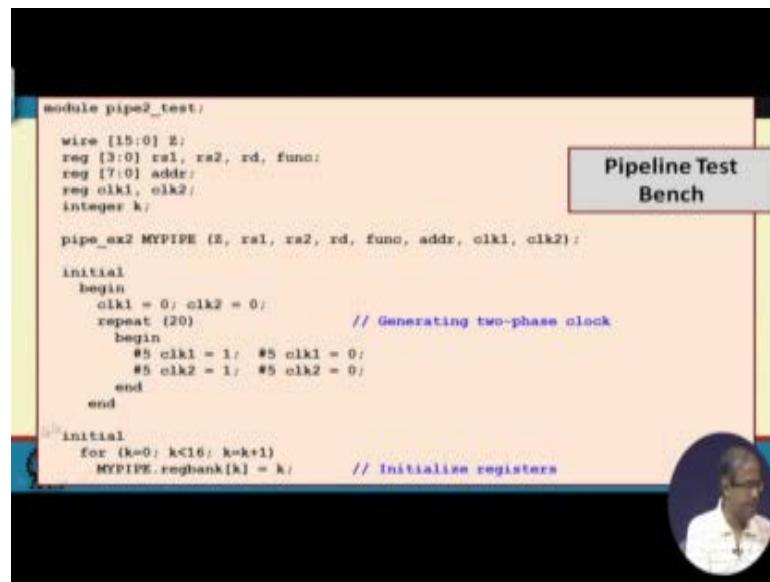
So, the Verilog code; we have written is repeat (20) and inside this, we have given a gap of 5, $\text{clk1} = 1$; again a gap of 5, $\text{clk1} = 0$; then again a gap of 5, $\text{clk2} = 1$; again a gap of 5, $\text{clk2} = 0$, let us say what happens here.

Let us assume these are our time scale of 5, these are our time scale of 5, 5, 5, 5, these are gaps of 5; this is 5, this is 10, this is 15, this is 20, this is 25, 30, 35, 40 and so on, right.

Now, initially and this is 0 of course, this is 0, it starts with 0, right, initially both clk1 and clk2 are 0s, this is clk1 and this is clk2 , both are 0s, this is also 0, this is also 0. So, at time 5, we are making $\text{clk1} = 1$. So, at time 5, we are making $\text{clk1} = 1$. So, again after a delay of 5, we are making $\text{clk1} = 0$.

Clk2 was 0 so long; so, again after a gap of 5, we are making $\text{clk2} = 1$; after gap of 5 $\text{clk2} = 1$; after a gap of 5, $\text{clk2} = 0$, this. So, clk1 is 0 in the meantime, so, this we repeat. So, again after a gap of 5, we make $\text{clk1} = 1$; again after gap of 5, $\text{clk1} = 1$; after a gap of 5, 0; again after a gap of 5 after this make this 1, you see we have generated a perfect 2 phase clock like this, right, just the earlier diagram we showed. So, we are generating a 2 phase clock here.

(Refer Slide Time: 25:41)



```
module pipe2_test;
    wire [15:0] z;
    reg [3:0] rs1, rs2, rd, func;
    reg [7:0] addr;
    reg clk1, clk2;
    integer k;

    pipe_ex2 MYPIPE (z, rs1, rs2, rd, func, addr, clk1, clk2);

    initial
        begin
            clk1 = 0; clk2 = 0; // Generating two-phase clock
            repeat (20)
                begin
                    #5 clk1 = 1; #5 clk1 = 0;
                    #5 clk2 = 1; #5 clk2 = 0;
                end
        end

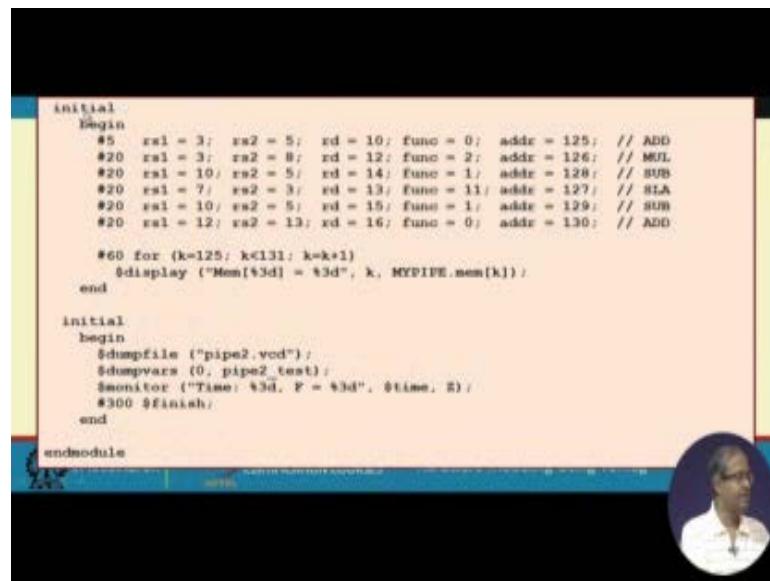
    initial
        for (k=0; k<16; k=k+1)
            MYPIPE.regbank[k] = k; // Initialize registers

```

And in this initial block, we are initializing the register bank, you see register bank is not accessible directly as a parameter. So, we are accessing it like this MYPIPE is the instantiated module dot regbank was the variable which was declared inside it, right.

You see inside it, there is a regbank, right. So, we are referring it like this MYPIPE.regbank[k], in this for loop, k goes from 0 up to 15; regbank[k] = k, which means register0 get 0; register1 gets 1; 2 gets 2; 3 gets 3, like that you are initializing register15 gets 15, this is how we are initializing registers.

(Refer Slide Time: 26:34)



```
initial
    begin
        #5 rs1 = 3; rs2 = 5; rd = 10; func = 0; addr = 125; // ADD
        #20 rs1 = 3; rs2 = 8; rd = 12; func = 2; addr = 126; // MUL
        #20 rs1 = 10; rs2 = 5; rd = 14; func = 1; addr = 128; // SUB
        #20 rs1 = 7; rs2 = 3; rd = 13; func = 11; addr = 127; // SLA
        #20 rs1 = 10; rs2 = 5; rd = 15; func = 1; addr = 129; // SHB
        #20 rs1 = 12; rs2 = 13; rd = 16; func = 0; addr = 130; // ADD

        #60 for (k=125; k<131; k=k+1)
            $display ("Mem[%3d] = %3d", k, MYPIPE.mem(k));
    end

initial
    begin
        $dumpfile ("pipe2.vcd");
        $dumpvars (0, pipe2_test);
        $monitor ("Time: %3d. T = %3d. $time, %");
        #300 $finish;
    end
endmodule

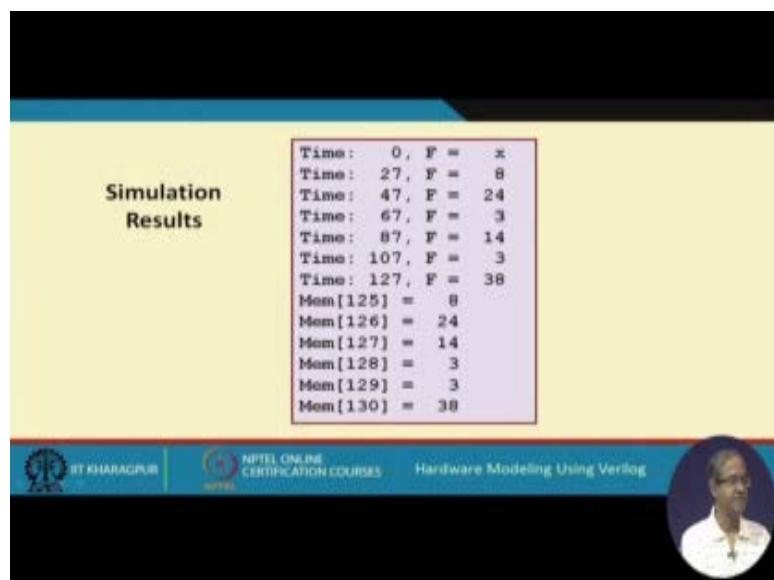
```

Then in this initial block, we are applying some sample inputs. So, we are giving a delay of 5 and after the delay of 5, we are giving 20, 20, 20 gaps. So that next clocks are applied. So, what I am doing; just giving $rs1 = 3$ and $rs2 = 5$; that means, we are reading from registers 3 and 5. So, what we expect here register 3 contains 3; register 5 contains 5, that is all we have initialized, right and func 0 means add. So, 3 and 5 result will be 8 and this result 8 will be storing in register number 5 as well as memory address 125.

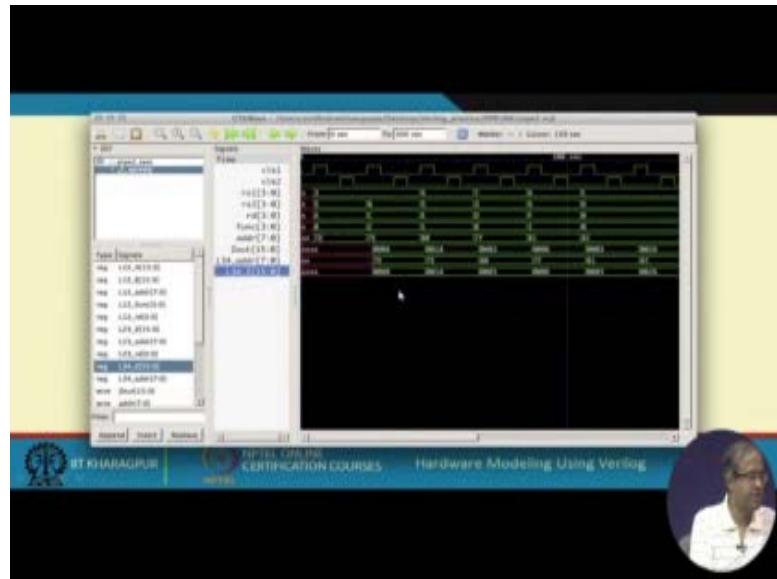
Similarly, next one is 3 and 8. So, register number 3 is still contains 3; 8 still contains 8 and the function is multiplication, 2. So, $8 * 3$ is 24, this 24, you are storing in register number 12 as well as here, 126. So, in this way you have some examples and at the end after everything is over, here in a for loop after some delay, you are displaying the memory contents that what is there in memory, starting from 125 address onwards, right and also we are monitoring the value of F, time and F.

So, if you run this simulation, you will see the output coming like this, you see first output, I had said the result will be 8, second one will be 24 This you can verify because you see, you have already modified register number 10, rd equal to 10, it becomes 8. So, in the third one, when you are accessing register 10, it is not 10, it has become, this has become 8. So, 8 and 5, it is subtraction 8 minus 5, you see result is 3 and you can verify the memory contents also, they also contain the same values, right.

(Refer Slide Time: 28:42)



(Refer Slide Time: 28:57)



So, if you look at the timing diagram using gtkwave, the same thing you can observe. So, you can see the 2 phase clock very nicely, this is 2 phase clock, this is the values of rs1, rs2 and rd, you have given 3, 5 and 10; function, you have given 0; address, you have given something. You see the test bench the address is 125, first 1, 3, 5, 10, function 0, address 125. So, 3,5, A is 10, 0, 7D is 125. So, this result will be final 0008. So, you need some delay for that. So, that pipeline delay will required and at the end of the third clock, you will be generating this result and after that result will be generated once every clock, right. So, this is exactly what is happening here.

So, what we have seen in this lecture is that this is the more recommended way to implement a pipeline that you use a 2 phase clock, but in the latches you can either use edge triggered in this Verilog code we have written, we have used posedge triggered flip-flops, but you can also use level triggered latches there as long as they took clock phases are non-overlapping and they sufficient gap between them even if we use non-overlapping latches means those are called transparent latches not edge triggered there is no problem, it will still work very fine.

So, we shall see later as it said with the more compressive example the design of a processor; there we will see how we can implement the instruction execution hardware, data path of a processor in a pipeline fashion, there we will be following some of the lessons that we have learnt over the last two lectures.

Thank you.

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

Lecture - 35
Switch Level Modeling (Part 1)

So, in this lecture we shall be moving away slightly from what we have been discussing and we shall be looking at something called Switch Level Modelling using Verilog. Now, this is something which I thought that as a designer someone should know that how we can model some circuits not only at the level of gates and functional blocks, but also at a lower level when our building blocks are transistors and switches. So, we shall see, what are the facilities that are provided by the language Verilog and what kind of modelling you can do or carry out using that, ok.

(Refer Slide Time: 01:08)

The slide has a dark blue header bar. Below it, a yellow section contains the title 'Introduction'. Underneath the title is a bulleted list of points. At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the text 'NPTEL ONLINE CERTIFICATION COURSES', and the course name 'Hardware Modeling Using Verilog'.

Introduction

- A switch level circuit comprises of a netlist of MOS transistors.
- It is not very common for a designer to design modules using transistors.
 - May be required in very specific cases, for designing leaf-level modules in a hierarchical design.
- Verilog provides the ability to model digital circuits at the MOS transistor level.
 - Transistors function as switches; they either conduct (ON) or are open (OFF).
- The four logic levels 0, 1, X, Z and the associated signal drive strengths help in the modeling.
- Two types of switches supported: *ideal* or *resistive*.

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

Let see. So, so when you talk about switch level circuit basically we are talking about a circuit that consists of MOS transistors. Now for those of you who are familiar with MOS transistors, you will be knowing that there are two kinds of MOS transistors, to the NMOS and PMOS, depending on the kind of impurities you put in the source and the drain regions, right, ok.

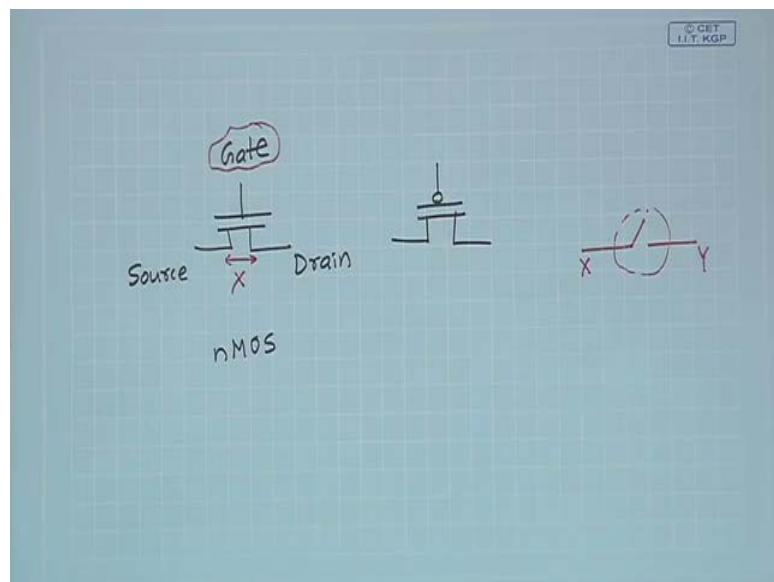
Now, the point to notice that as a designer when you are using Verilog to design a digital circuit it is not very common for you to design a model using transistors, but there may

be some very specific cases, where a low level module which is sometimes called leaf level module, you can be using MOS transistors to model them as a hierarchical design. Just I am give an example suppose you want that well Verilog provides gate level primitives AND, OR, NAND, NOR, XOR, all right, but you want some, your own design using MOS transistors to implement some gate.

So, you can write a very low level module using MOS transistor, let say to implement an exclusive or gate and that XOR gate you can use, you can instantiate it in higher level designs to create more complex designs. So, it is only under these conditions you can possibly use switch level modelling, normally you do not use switch level models in a real design.

Now, as I had said Verilog provide some facilities for modelling at the MOS level where the transistors are regarded as a switch, you see in MOS level a transistor is represented like this.

(Refer Slide Time: 03:13)



This is called the Gate and the other two terminals, one of them is Source, other is Drain, this is an example of an NMOS transistor, the symbol and PMOS transistor is same where there is a bubble here, same kind.

Now, we are regarding a transistor as a switch, let us see what is a switch? a switch schematically represents like this, there are 2 terminals X and Y, they may be connected

if I close the switch; they will be not be connected if I open the switch. Here also depending on what voltage I am applying to gate either the source and drain may be conducting, they may be connected or they may not be connected.

So, you can regard a transistor as a switch that is what is meant by switch level modelling, right. So, earlier we have seen that in Verilog, there are 4 logic values which are supported 0, 1, x and z and earlier we also talked about signal drive strength, drive strengths. Now, here when you are just working with MOS transistors and the basic primitives we shall be seeing, we shall be appreciating that why the signal drive strength are required in the modelling and regarding the switches there are 2 types of switches which are supported, one is called ideal, other is called resistive.

Now, in ideal switch means when you close a switch there will be 0 resistance and a resistive switch means when you close the switch there will be a low resistance but not 0, a finite low value of resistance. So, if there is a low resistance what will happen is that if I, if we apply a signal at one side, on the other side this strength of the signal will be reducing a little bit, this is how the concept of signal strength comes in, ok.

(Refer Slide Time: 05:44)

Various Switch Primitives in Verilog

- Ideal MOS switches
 - nmos, pmos, cmos
- Resistive MOS switches
 - rnmos, rpmos, rcmos
- Ideal Bidirectional switches
 - tran, tranif0, tranif1
- Resistive Bidirectional switches
 - rtran, rtranif0, rtranif1
- Power and Ground nets
 - supply1, supply0
- Pullup and Pulldown
 - pullup, pulldown

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

When a signal is passing through a resistive switch, the strength of the signal decreases, fine. Now the various switch level primitives which are supported by Verilog are as follows, the first are the Ideal MOS switches, they are represented by this primitive's nmos, pmos and cmos, we shall see them. There are resistive versions of these also just

an r before this names, very similar, but there will be a non-zero resistance here and this switches are normally assume to be conducting current in one direction. But there is another kind of a switch which is called bidirectional switch which is assume to conduct in both directions, ok.

So, you can have bidirectional switches also, these are denoted by tran, tranif0, tranif1, we will see what these are and similarly there are resistive versions of these switches with r. And there are some keywords to indicate supply voltages power supply1 and supply0, directly you can mention them and there is something all pullup and pulldown, this also we shall see.

(Refer Slide Time: 07:03)

(a) NMOS and PMOS Switches

- Declared with keywords "nmos" and "pmos".
- Format for instantiation:
`nmos (or pmos) [instance_name] (output, input, control);`
 Here, "instance_name" is optional.

Also called pass transistors.

nmos	control	0	1	x	z
0	z	0	L	L	
1	z	1	H	H	
x	z	x	x	x	
z	z	z	z	z	

(a) nMOS switch

pmos	control	0	1	x	z
0	0	z	L	L	
1	1	z	H	H	
x	x	z	x	x	
z	z	z	z	z	

(b) pMOS switch

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

Now, let us come to NMOS and PMOS switches first, this NMOS and PMOS switches they are declared with the keywords nmos and pmos. Now as I had shown you earlier, this is a schematic of an NMOS transistor acting as a switch and this is a PMOS transistor. There are 2 input and output terminals and there is a control, gate is the control, here also it is similar.

Now, the way it works is that if the control for an NMOS switch, if the control is at high, if the control is 1 then the gate is conducting. So, if the input is 0, this is input, then the output is 0, this is output; if the input is 1, the output is 1, but if the control is 0 then the switch is off. So, the output will be in the high impedance state, it has not connected to anything, ok.

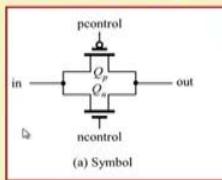
Just ignore these 2 columns for the time being, if the control is either undefined or in the high impedance state, well the Verilog language specifies that if we apply a 0, the output voltage will also be at the low level, ground level. If we apply a 1, the output will be at a high level because there is no voltage drop. But if you apply, I means, x and z in the input then the output will be indeterminate x and z, similarly for a PMOS, it is just a reverse, if the control is set to 0, then the switch is conducting; if the control is 1, then the switch is off, ok.

We need only this much, this part of the table and the way you can instantiate is you write either nmos or pmos, well instance name is optional, you can give a name and first the output, then the input, then the control, this is the order.

(Refer Slide Time: 09:11)

(b) CMOS Switch (Transmission Gate)

- Declared with keywords “*cmos*”.
- Format for instantiation:
`cmos [instance_name] (output, input, ncontrol, pcontrol);`
 Here also, “*instance_name*” is optional.



(a) Symbol

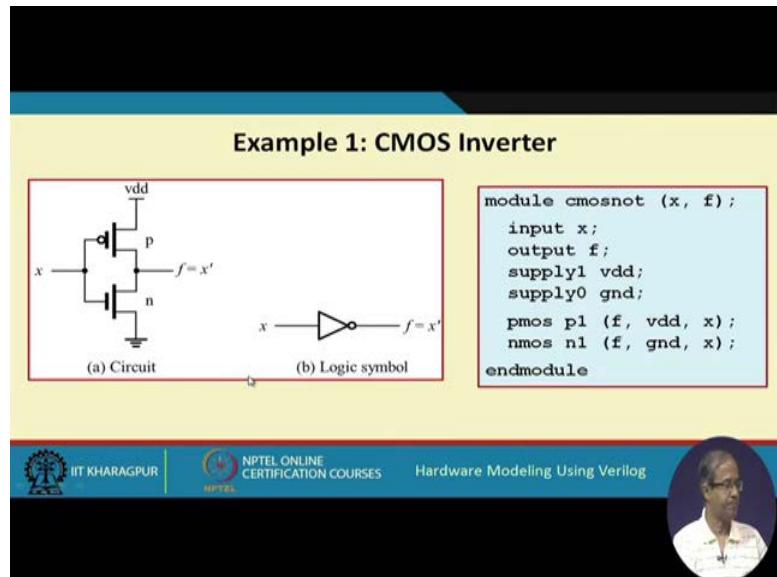
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

These switches are sometimes also called pass, well there is a CMOS version of, CMOS stands for complimentary MOS, complimentary MOS.

So, in a CMOS switch there is an NMOS switch and a PMOS switch which are connected in parallel and there are 2 control signals ncontrol and pcontrol. So, when we instantiate it, it is output, input, ncontrol and pcontrol. Normally this ncontrol and pcontrol are complements of each other, say if I apply 1 here and 0 here, then it will be conducting; if I apply 0 here and 1 here, both of them are off, so it will be off. This is a better switch as compared to a single transistor switch because when you use single n-

type or p-type transistor as a switch, there is a voltage degradation which happens, but if you use two back to back switches that voltage degradation is avoided.

(Refer Slide Time: 10:22)



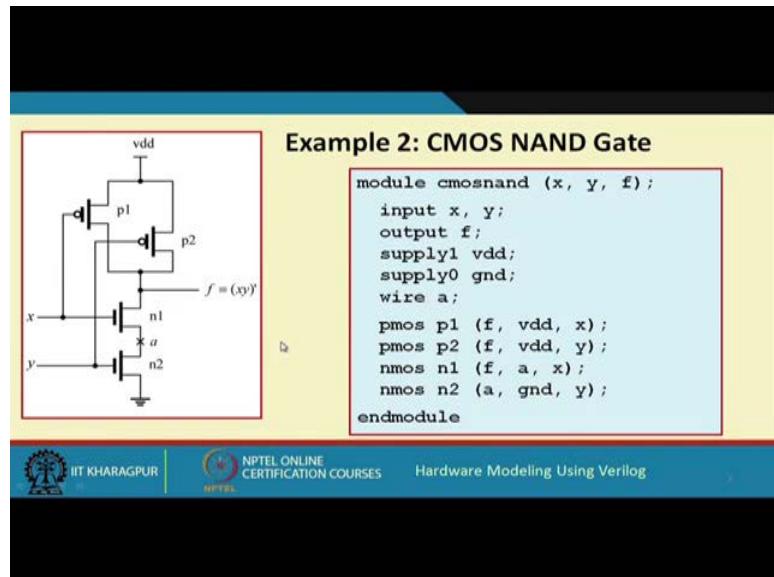
So, I am not going into details of this here, but just remember this that the CMOS switch is a better kind of a switch. Let us take some example designs, a CMOS inverter for those of you who have studied MOS level gates will just identify the circuit. There is an NMOS transistor, there is a PMOS transistor whose gates are connected, this is the input and the sources or the drain whatever you call, this is output, other side is connected to power supply vdd and ground, this is the symbolic notation.

Now, the way it works is very simple, if the input x is 0, if it is 0 then the p-type switch is ON and the n-type switch is OFF. If p-type is ON that vdd will be connected to the output, output will be 1, if it is 0, output is 1 and if the input is 1 then the n-type will be conducting, this switch is ON.

So, the output will be connected to ground, output will be 0, so it is NOT gate. So, you can directly specify this net list in Verilog like this, let us call it `cmosnot`, x is the input, f is the output, so input x , output f . So, here you are using vdd and ground terminals also, that is why you define 2 variables, one you called as vdd another called gnd of type `supply1` and `supply0`. There is 1 pmos transistor let's call it $p1$, f and vdd are the 2 terminals, this is f and this is vdd and the gate is x .

Similarly, there is another transistor nmos, just call it n1, f and gnd are the 2 terminals and the input is x. So, this is the complete description.

(Refer Slide Time: 12:11)



Similarly, you can have a NAND gate, 2 input NAND gate. So, 2 input NAND gate, there are 2 n-type transistors here and 2 p-type transistors here, for a NAND gate you recall, when both the inputs are 11, output is 0. So, when both x and y are 1 and 1, both the switches n1 and n2 will be conducting, ON. So, the output will be connected to ground, it will be 0. But if, if any one of them is 0, so, either p1 or p2 will be conducting. So, there will be a path from vdd to the output. So, the output will be 1.

So, if any one of the input is 0, output is 1, that is NAND. So, description, see you see is I am not here trying to teach you how to design circuits using MOS transistor. So, what I am saying is that given a MOS level circuit, how to modulate in Verilog. You see this circuit can be model in Verilog like this x, y and f; inputs are x, y; output f, again vdd, gnd are the 2 supplies and intermediate there is one line a, I declared as wire. pmos, just this p1, f, vdd, x, f is this, this is f, this is vdd and this is x. For p2, f, vdd, y; f, vdd, y. For n1, they said f, a, x; f, a, x and for n2, gnd, a, or a, gnd, whatever order you specify and y; a, gnd, y, ok.

(Refer Slide Time: 13:55)

```
module cmosnand_test;
    reg in1, in2;
    wire out;
    integer k;
    cmosnand MYNAND2 (in1, in2, out);
    initial
        begin
            for (k=0; k<4; k=k+1)
                begin
                    #5 {in1,in2} = k;
                    $display ("In1: %b, In2: %b, Out: %b", in1, in2, out);
                end
        end
endmodule
```

In1: 0, In2: 0, Out: 1
In1: 0, In2: 1, Out: 1
In1: 1, In2: 0, Out: 1
In1: 1, In2: 1, Out: 0

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

Now, this cmosnand, if you write a simple test bench to see whether it works or not. So, so you can simulate and see that it works, really works. So, we have instantiated this cmosnand in a test bench like this, we called it MYNAND2, in1, in2, out; the inputs were declared as reg; out as wire and defined an integer k. So, why? just in order to apply all possible inputs. You see this is a 2 input circuits, so, what we did, in this initial block, we give a for-loop which runs from k equal to 0 up to 3; k less than 4; 0, 1, 2, 3. If there are 2 inputs, so, the combinations will be 00, 01, 10 and 11 which means 0, 1, 2, 3 and counting in decimal that k.

So, in k I am counting decimal 0 up to 3 and that value of k, I am assigning to the inputs using concatenation operation. Say for example, if k is 3 it means 11, in1 will be 1, in2 will be 1. So, this will be happening automatically and we are displaying the values of in1, in2 and out.

(Refer Slide Time: 15:24)

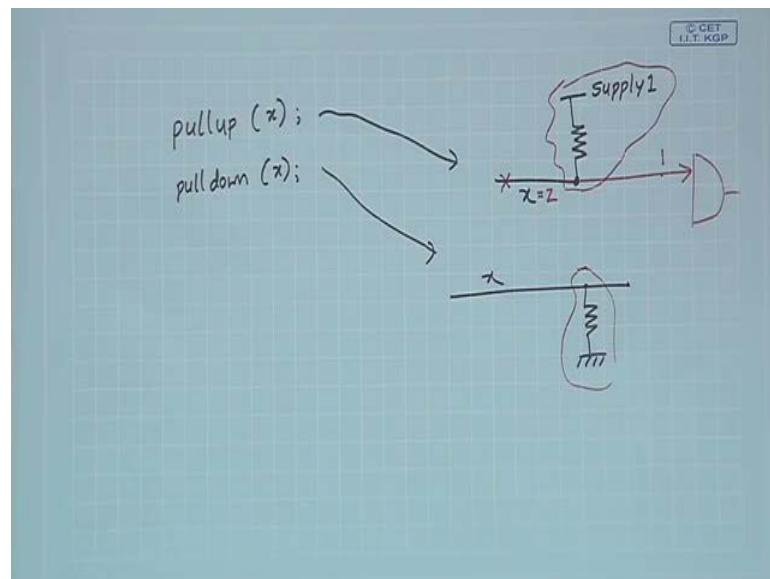
Example 3: Pseudo-NMOS NOR Gate

```
module pseudonor (x, y, f);
    input x, y;
    output f;
    supply0 gnd;
    nmos nx (f, gnd, x);
    nmos ny (f, gnd, y);
    pullup (f);
endmodule
```

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

So, if you just simulate it, we get this result 00, output is 1; 01, output is 1; 10, 1; 11, 0, this is NAND, fine.

(Refer Slide Time: 15:41)



Let us take another example, this is called pseudo-NMOS NOR gate, but before we explain this let us tell you about the primitive that are available, pullup and pulldown. See you can, with a signal, let say x, you can either write pullup (x) or with a signal y. Say x, you can write pulldown (x), what does this mean, pullup(x) means there is a signal line x, pullup means from here you are connecting a resistance to supply1; that means,

positive supply voltage, this is called pullup. So, when you say pullup, this whole thing will be included by default there.

So, the meaning of pullup is that suppose this output your feeding to the input of some gate let us say and because of some reason this x is in the high impedance state, let say $x = z$, then because of the pullup this input will still be at 1 because it is connected to the supply with the resistance. So, it will not be in the high impedance. So, input will not be z .

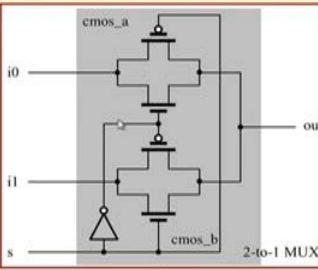
Similarly, pulldown means this x is the signal you connect a resistance to ground, this means pulldown. So, in circuits we often use this kind of pullup and pulldown configurations to implement several things. Like here we are showing an alternate way of designing gates, this is a NOR gate and this is called pseudo-NMOS. Pseudo means here you have a resistance in the pullup and down you have 2 NMOS transistors. You see why it is NOR, in a NOR gate how does it work. So, if the inputs are 00 then only the output is 1 otherwise the output is 0, let us say here if the inputs are 0 and 0, both the transistors are off.

So, f is not connected to ground, so f will be connected through this pullup to 1. So, f will be high or 1, but if any one of them is 0 that transistor will be ON and f will be connected to ground. Now, see f is connected on one hand to the ground through an ON switch, on the other side it is connected to a pullup resistance to a supply voltage. So, will the output be ground or high, you see the, the idea is pullup or pulldown has a lower signal strength because of the resistance. So, when two such signals are tied together, the signal which is stronger, it will dominate; like here on one side you are trying to connect to ground and other side you are connecting to 1, but ground is a stronger signal here.

So, ground will dominate because this is ideally 0 resistance, so f will be 0, right. So, declaration is very simple x, y input; f is output and here we need to declare only ground supplies gnd; there is one nmos nx , another nmos ny ; $f, gnd, x; f, gnd, y$ and there is a pullup at point f , at point f is a pullup, this is the module description. So, this module in a same way if you do a simulation that same test bench, we have just included MYNOR instead of the other one pseudonor. So, if you simulate it, remaining part is same, you see the output is coming like this 00 is 1 otherwise output is 0, this is NOR.

(Refer Slide Time: 20:01)

Example 4: CMOS 2x1 Multiplexer



2-to-1 MUX

```
module mux_2to1 (out, s, i0, i1);
  input s, i0, i1;
  output out;
  wire sbar;
  not (sbar, s);
  cmos cmos_a (out, i0, sbar, s);
  cmos cmos_b (out, i1, s, sbar);
endmodule
```

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



So, this is one more example let us take, here we use a CMOS switch, just see this diagram, there are two cmos switches, one cmos switch is here another cmos switch is here. And we are implementing a 2 to 1 multiplexer and the 2 inputs are connected to this cmos switches and the select line s, is connected to the nmos transistor of one of the switches and to the pmos transistor of the other switch and there is a NOT gate, NOT of s is connected to the p-transistor of this switch and n-transistor of this switch. What does this mean? If s is 0; if s is 0 then this will be 1, NOT, this will be ON, this will also be ON. So, this switch is ON, but this switch is OFF because this is ON, i0 will go to the output and if s equal to 1 then this will be ON and also this will be ON, but this will be OFF.

So, i1 will go to out. So, representing in Verilog is very simple, this input s, i0, i1, output is out and there is an intermediate line, output of this NOT gate I call it sbar. So, I instantiate this not, s is the input, sbar is the output and 2 cmos gates I have just instantiated, one I called cmos_a, other I called cmos_b. So, the inputs are, output the i0 and control is, I means sbar in n-type and s in p-type. So, s-bar, n type and p-type and for cmos_b, out, this is i1 and here it is s in n-type and sbar in p-type; s and sbar.

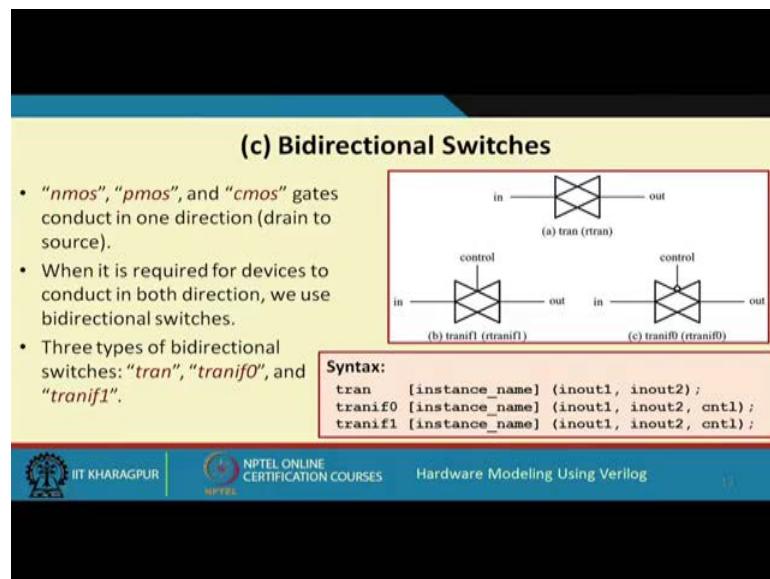
(Refer Slide Time: 22:18)

```
module cmosmux_test;
    reg sel, in0, in1;
    wire out;
    integer k;
    cmosmux MUX21 (out, sel, in0, in1);
    initial
        begin
            for (k=0; k<8; k=k+1)
                begin
                    #5 {sel,in0,in1} = k;
                    $display ("Sel: %b, In0: %b, In1: %b, Out: %b",
                              sel, in0, in1, out);
                end
        end
endmodule
```

NPTEL ONLINE CERTIFICATION COURSES Hardware Modeling Using Verilog

So, this again see here there are 3 inputs total, right. So, we have written test bench similarly, but we have applied 8 pattern 0 up to 7 and this case assigned to select n0, n1 like this. So, we have printed the values of sel, in1, there should be in0, actually in0, in1 and out. So, you see, if select is 0 then n0 is getting selected; if select is 1 then n1 is selected, 0 1 0 is getting selected, right. So, it works as a multiplexer.

(Refer Slide Time: 23:00)



So, and lastly in this lecture we shall be talking about something called bidirectional switches. So, of course, will not be giving example, just to tell you what it is. See in a normal switch PMOS, NMOS or CMOS well it is assumed that current flows in only one

direction, but this is only for the purpose of simulation, in a real switch current actually frozen both directions. So, if in a design you need to have a switch where you need current to flow in both directions then you should use something called bidirectional switchs which are called tran.

So, there are 3 kinds of bidirectional switches tran, tranif0 and tranif1. So, the syntax of the tran switch is tran, well instance name as usual is optional means because it is bidirectional, I call them inout, inout1, inout2, there are 2 terminals and this is always conducting. But tranif0 and tranif1 means there is a control signal; tranif0 says if the control signal is 0 then it will conduct and tranif1 says if the control signal is 1, if 1 then it will conduct and if it is control is other way around then output will be tri-stated, this will be OFF, this is how the bidirectional switches work.

So, with this we come to the end of this lecture where we talked about some of the low level MOS switch primitives which also you can use in Verilog modelling, but again I am telling you these are mostly used for simulation purposes, most of the synthesis tools will not be supporting this kind of MOS level primitives. So, in the next lecture we shall be looking some more examples on this kind of switch level modellings and a few other things.

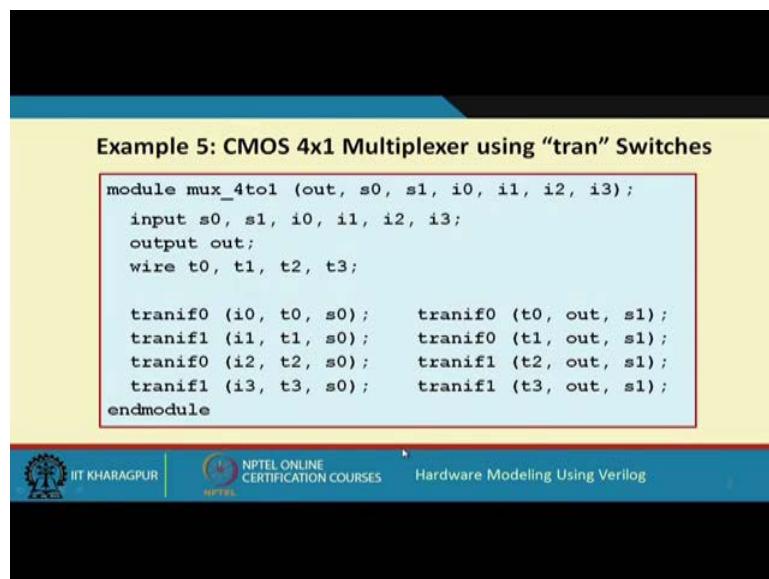
Thank you.

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

Lecture - 36
Switch Level Modeling (Part 2)

So, in this lecture we continue our discussion on switch level modelling in Verilog. You recall in our last lecture we talked about the NMOS, PMOS and the CMOS switches, we took some examples, some simple gate implementations. And we also saw how a bidirectional switch works, but we did not see any examples of that. So, today we shall be starting with giving some examples of such bidirectional switches and some other things. So, the topic of this lecture is switch level modelling the second part, ok.

(Refer Slide Time: 01:03)



The screenshot shows a presentation slide with a yellow header bar. The title in the header bar is 'Example 5: CMOS 4x1 Multiplexer using "tran" Switches'. Below the title, there is a Verilog code block:

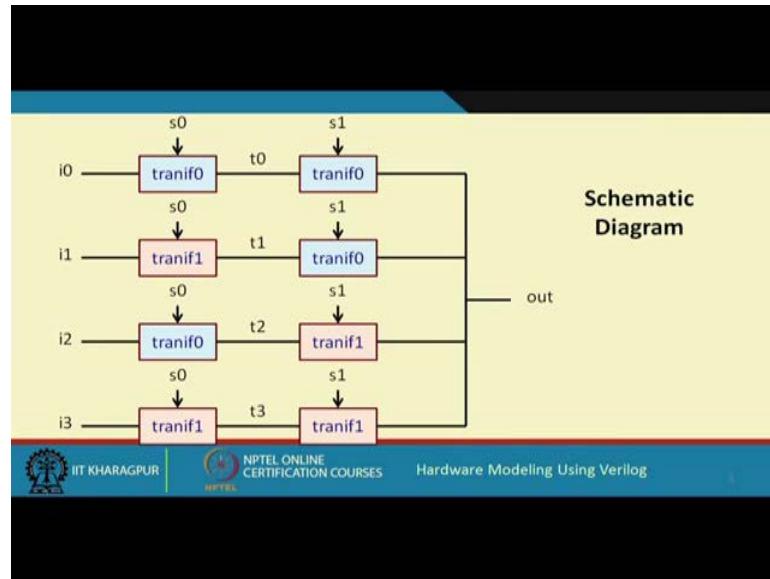
```
module mux_4to1 (out, s0, s1, i0, i1, i2, i3);
    input s0, s1, i0, i1, i2, i3;
    output out;
    wire t0, t1, t2, t3;

    tranif0 (i0, t0, s0);    tranif0 (t0, out, s1);
    tranif1 (i1, t1, s0);    tranif0 (t1, out, s1);
    tranif0 (i2, t2, s0);    tranif1 (t2, out, s1);
    tranif1 (i3, t3, s0);    tranif1 (t3, out, s1);
endmodule
```

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the text 'Hardware Modeling Using Verilog'.

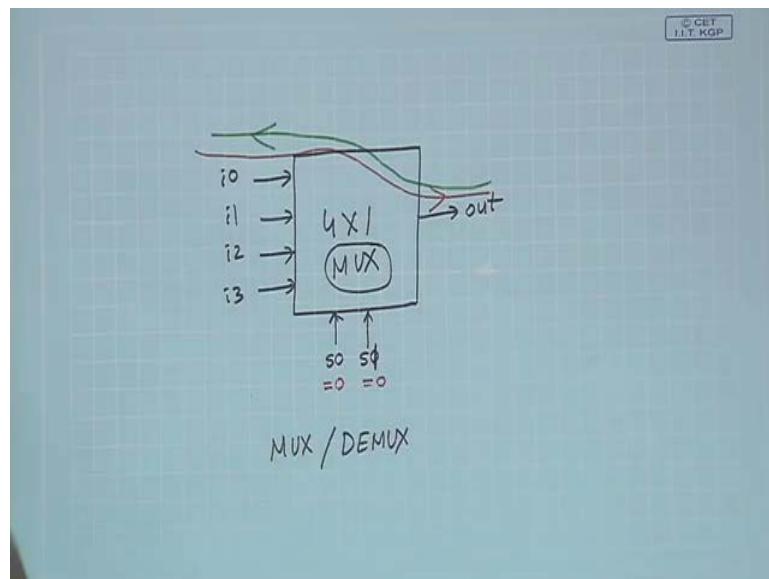
So, here what we try to do is, we are trying to implementing a 4 by 1 multiplexer using bidirectional switches, but let us look into this Verilog code later.

(Refer Slide Time: 01:23)



First let us see the schematic how we are trying to design. Here we are trying to implement a 4 line to 1 line multiplexer, ok.

(Refer Slide Time: 01:39)



You just try to understand the functionality in a 4 to 1 multiplexer. So, you have the 4 inputs i0, i1, i2 and i3. So, we have the output, let us call it out and the 2 select lines, say s0 and s1, depending on the select line one of the inputs will be connected to the output.

Now, here because we are using bidirectional switches, we are actually not just implementing MUX, we are implementing multiplexer cum demultiplexer. What does

that mean? This means suppose I have applied $s_0 = 0$ and $s_1 = 0$, both 0; that means, i_0 is selected, which means from input i_0 current will flow to out. Not only that because the switches are bidirectional, from the output node current will also flow into i_0 .

Similarly, if I apply some other control signal, this current will get diverted to that particular input and this will be bidirectional, current can flow in any direction, right. So, although I have called it a multiplexer it is actually a multiplexer cum demultiplexer. Multiplex means many to one, demultiplex means one to many, both ways, ok.

Now, let us see how it works. Here we have used a combination of $\text{tranif}0$ and $\text{tranif}1$ switches. Now you recall what a $\text{tranif}0$ switch means if the control signal is 0, if 0 then it will be conducting. And a $\text{tranif}1$ switch means if the control signal is 1 then it will be conducting, ok.

Now, let us see. The 4 inputs are i_0 , i_1 , i_2 and i_3 and this is the out. Now when I have to select i_0 ; you see this s_0 I have connected to all the switches in the first level and s_1 I have connected to all the switches in the second level. So, if I want to select i_0 , so, s_0, s_1 will be both 0 and 0. You see the first row of the structure. I have connected to $\text{tranif}0$ switches, if s_0 is 0, this will be on; if s_1 is 0 this will be on. So, when both s_0, s_1 are 0 this path will be on and i_0 and out will be connected.

Now, suppose s_1 is 0 and s_0 is 1 then i_1 should be selected. That is why we have used a $\text{tranif}1$ switch here. If s_0 is 1 and s_1 is 0 then this path will be selected. Similarly, if s_1 is 1 and s_0 is 0 then i_2 is selected. And if both of them are 1 then the last one is selected.

So, depending on the value of s_0 and s_1 , exactly one of the rows is selected and that input is getting connected to the output, right. Now this has been coded directly in Verilog here. You see there are 8 tran switches, 4 and 4, 8. This is a multiplexer out; s_0, s_1 are the select lines and i_0, i_1, i_2, i_3 are the inputs.

So, the intermediate lines these one t_0, t_1, t_2, t_3 they are declared as wires. So, we have just simply coded you see $\text{tranif}0$, first row has 2 $\text{tranif}0$ switches. So, i_0 to t_0 and t_0 to out. You see i_0 to t_0 and t_0 to out. Second one has a $\text{tranif}1$ and $\text{tranif}0$, this is $\text{tranif}1$, $\text{tranif}0$. This is i_1 to t_1 and t_1 to out. Similarly, i_2 to t_2 , t_2 to out; i_3 to t_3 , t_3 to out. So, this exactly implements this net list.

Now, here you see in this circuit there will be 6 inputs right, 4 inputs and 2 select lines. So, in 6 inputs there can be 64 total combinations.

(Refer Slide Time: 06:44)

```
Test Bench
module mux41_test;
reg s0, s1, i0, i1, i2, i3;
wire out;
integer k;
mux_4to1 MYMUX41 (out, s0, s1, i0, i1, i2, i3);
initial
begin
  for (k=0; k<64; k=k+1)
    begin
      #5 {s0,s1,i0,i1,i2,i3} = k;
      $display ("Sel: %2b, In: %4b, Out: %b",
                {s0,s1}, {i0,i1,i2,i3}, out);
    end
end
endmodule
```

So, here we have simulated this also using a test bench, where we have instantiated this multiplexer. All this 6 inputs have declared as reg, and in a similar style in a for-loop we are generating all this 64 patterns, 0 up to 63. And these 6 variables s0, s1, i0, i1, i2, i3, we are assigning the value of k to that.

So, it will be converted to binary, the last 6 bits of k will be assigned here. And you are displaying the value of s0 and s1 in 2-bit binary, sel; 4-bit inputs in and the output. So, this loop will be running 64 times, right.

(Refer Slide Time: 07:34)

Simulation Results

Sel: 00, In:	Out:
0000	x
0001	0
0010	0
0011	0
0100	0
0101	0
0110	0
0111	0
1000	0
1001	1
1010	1
1011	1
1100	1
1101	1
1110	1
1111	1

Sel: 01, In:	Out:
0000	1
0001	0
0010	0
0011	1
0100	1
0101	0
0110	0
0111	1
1000	1
1001	0
1010	0
1011	1
1100	1
1101	0
1110	0
1111	1

So, there will be 64 lines of outputs. So, I am showing it into slides, this is first 16 lines.

So, when sel is 00, In is 00 up to this you see, the first line In0 is getting selected, this is In0.

Now, you can ask why the first one is coming as x. You see the reason is as follows, see I means you are displaying it here, right and see sel is 0, In is 0. So, sel is 0, In is 0. So, you are giving a delay of 5 and then you are assigning the value of k, but when you are displaying it there is no delay there. So, this will be displaying the previous value that is why before the assignment is done you are getting displayed. Similarly, when select is 01 you see the second input is getting selected, right; similarly 10, similarly 11, right, fine.

(Refer Slide Time: 08:44)

Example 6: Full Adder using Transistor Level Modeling

```
module fulladder (sum, cout, a, b, cin);
    input a, b, cin;
    output sum, cout;
    fa_sum SUM (sum, a, b, cin);
    fa_carry CARRY (cout, a, b, cin);
endmodule
```

```
module fa_carry (cout, a, b, cin);
    input a, b, cin;
    output cout;
    wire t1, t2, t3, t4, t5;
    cmosand N1 (t1, a, b);
    cmosand N2 (t2, a, cin);
    cmosand N3 (t3, b, cin);
    cmosand N4 (t4, t1, t2);
    cmosand N5 (t5, t4, t4);
    cmosand N6 (cout, t5, t3);
endmodule
```

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

Now, let us see slightly more complex example of a transistor level modelling. Now here we are trying to implement a full adder using transistor level modelling. So, the top level module says that for the full adder, we are using a sum module and a carry module. So, fa_sum and fa_carry. So, fa_carry we are directly implementing using this cmosand.

(Refer Slide Time: 09:23)

```
module cmosnand (f, x, y);
    input x, y;
    output f;
    supply1 vdd;
    supply0 gnd;
    pmos p1 (f, vdd, x);
    pmos p2 (f, vdd, y);
    nmos n1 (f, a, x);
    nmos n2 (a, gnd, y);
endmodule
```

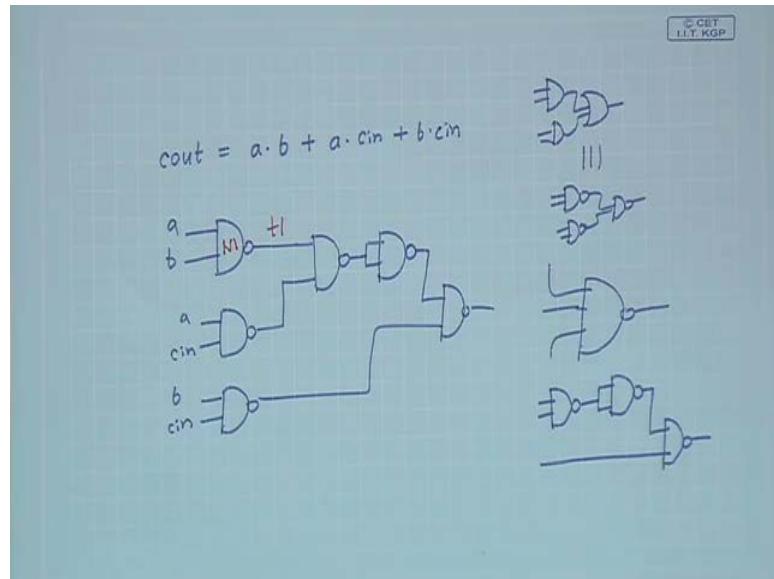
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

You see cmosnand is something which you have implemented later, we shall come to it, cmosnand you already seen earlier, right, using 2 PMOS and 2 NMOS transistors can

implement a nand gate. So, this cmosnand is already there, we have already implemented this.

So, using that we are using 6 nand gates to implement the carry, how? you see for a, for the carry function.

(Refer Slide Time: 09:50)



So, how does carry function is implement? $a.b + a.cin + b.cin$. This is the carry function.

So, it is a 2 level AND-OR circuit.

Now, you know any 2 level AND-OR circuit is functionally equivalent to a 2 level NAND-NAND circuit. So, we are using this principle, this we are implementing as a 2 level NAND-NAND. So, in a first level, we are connecting $a.b$, we connect $a.cin$, we connect $b.cin$. Next level there has to be a NAND gate, there has to be a NAND gate that will be taking the 3 inputs, but our cmosnand gate is only a 2 input NAND gate. So, how to implement a 3 input NAND gate? 3 input NAND gate we implement it like this. We take a 2 input NAND gate, we do a NOT by connecting the same input and using another 2 input NAND gate.

So, exactly we have done this, we have connected the first 2 to a NAND gate then we have used another NAND gate for inversion and a third NAND gate connected like this. There are total 6 NAND gates. So, exactly we have done or created that net list here, this

you can verify and t1, t2, t3, t4, t5 will be this intermediate line, there are 5 intermediate lines you can see 1, 2, 3, 4 and 5.

So, I leave it as an exercise for you to verify that whether this net list as I have specified is correct or not and which one is N1, which one is N2 and so on this you can see. For example, let me tell you one of them, see this is N1 (t1, a, b); t1, a, b will be this; a, b is this, this is t1 and this gate is N1, right. Similarly, with others you can identify it, right.

(Refer Slide Time: 12:37)

The screenshot shows a presentation slide with a yellow background. At the top, there is a large black rectangular area. Below it, the slide content is displayed. The content consists of two Verilog code snippets in red-bordered boxes:

```
module myxor2 (out, a, b);
    input a, b;
    output out;
    wire t1, t2, t3, t4;

    cmosnand N1 (t1, a, a);
    cmosnand N2 (t2, b, b);
    cmosnand N3 (t3, a, t2);
    cmosnand N4 (t4, b, t1);
    cmosnand N5 (out, t3, t4);
endmodule
```

```
module fa_sum (sum, a, b, cin);
    input a, b, cin;
    output sum;
    wire t1, t2;

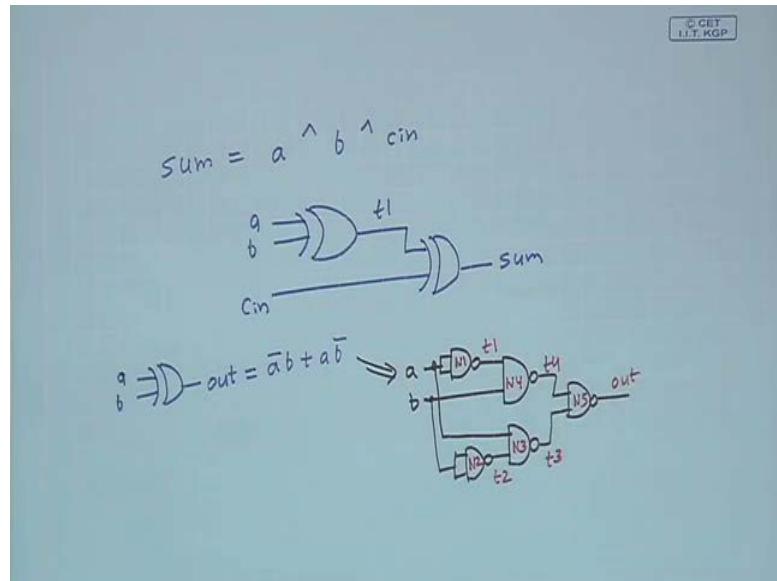
    myxor2 X1 (t1, a, b);
    myxor2 X2 (sum, t1, cin);
endmodule
```

At the bottom of the slide, there is a footer bar with the following elements from left to right:

- IIT KHARAGPUR logo
- NPTEL ONLINE CERTIFICATION COURSES logo
- Hardware Modeling Using Verilog

Now, after we have implemented carry then we have to implement the sum. Sum at the higher level you implemented using 2 XOR gates.

(Refer Slide Time: 12:46)



Because sum is what? Sum is nothing but sum is, $a \oplus b \oplus \text{cin}$. So, you can implement it using 2 XORs, say first you can XOR let say a and b , you generate let us say a temporary $t1$ then use another XOR connect cin to it, you generate sum, right.

So, we have used this kind of a configuration, this net list to express fa_sum in a structured way. There are 2 XOR gates, one of them is taking a , b and generating $t1$; other is taking $t1$ and cin and generating sum. Now these are 2 input XORs, well again 2 input XOR I have to implement using CMOS.

Now, let us see. So, how to implement a 2 input XOR? let say a , b and out . So, the XOR function is given as $a \oplus b$ or $\bar{a}b + a\bar{b}$. So, again using 2 level gates NAND-NAND realisation, you can implement this using, this is $\bar{a}b$. Suppose this is your a , you make it \bar{a} with a NAND gate make it inverted a and b , this is b and $\bar{a}b$, this is a you connect directly here a and $\bar{a}b$. So, b you take make a knot of it b OR. So, OR 2 level and OR is equal to 2 level NAND-NAND. This will be your equivalent net list, 5 NAND gates, 2 input NAND gates. So, here we have exactly done that, you see 5 NAND gates.

Let us identify the gates here cmosnand N1 ($t1$, a), a is this one, this one will be $t1$, this is N1; second N2 ($t2$, b , b) and the output is $t2$, this one $t2$, b and b , this is N2. And N3 is a and $t2$, a and $t2$, this is N3 and the output your calling $t3$. And N4 is b and $t1$, b and $t1$,

this is your N4 and output your calling t4 and the last gate, this is N5, t3 and N4 we generate out, right.

(Refer Slide Time: 16:10)

The screenshot shows a Verilog code editor with a yellow background. A red-bordered box highlights the following Verilog code:

```
module cmosnand (f, x, y);
    input x, y;
    output f;
    supply1 vdd;
    supply0 gnd;
    pmos p1 (f, vdd, x);
    pmos p2 (f, vdd, y);
    nmos n1 (f, a, x);
    nmos n2 (a, gnd, y);
endmodule
```

Below the code, there is a small circular icon of a person's face. At the bottom of the slide, there is a blue footer bar with the IIT Kharagpur logo, the NPTEL logo, and the course title "Hardware Modeling Using Verilog".

So, this is how you design all of them using cmosnand and of course the final transistor level cmosnand is this. So, you have done a perfectly hierarchical design with the transistor level 2 input NAND gate as at the leaf, and using that 2 input NAND in a hierachal way you have designed a full adder, right.

(Refer Slide Time: 16:30)

The screenshot shows a Verilog code editor with a yellow background. A red-bordered box highlights the following Verilog test bench code:

```
Test Bench
module fulladder_test;
    reg a, b, cin;
    wire sum, cout;
    integer k;
    fulladder FA (sum, cout, a, b, cin);
initial
begin
    for (k=0; k<8; k=k+1)
        begin
            #5 {a, b, cin} = k;
            $display ("Inputs: %b Sum: %b, Carry: %b",
                      (a,b,(cin)), sum, cout);
        end
    end
endmodule
```

Below the code, there is a small circular icon of a person's face. At the bottom of the slide, there is a blue footer bar with the IIT Kharagpur logo, the NPTEL logo, and the course title "Hardware Modeling Using Verilog".

So, again we have written a test bench. So, we have instantiated the full adder, there are 3 inputs a, b, cin. So, again we have done this for-loop for 7 times, generated all values are a, b, cin, and you are printed the values of a, b, cin together as a 3-bit number and sum and carry.

(Refer Slide Time: 16:56)

Simulation Results

Inputs: 000	Sum: 0	Carry: 0
Inputs: 001	Sum: 1	Carry: 0
Inputs: 010	Sum: 1	Carry: 0
Inputs: 011	Sum: 0	Carry: 1
Inputs: 100	Sum: 1	Carry: 0
Inputs: 101	Sum: 0	Carry: 1
Inputs: 110	Sum: 0	Carry: 1
Inputs: 111	Sum: 1	Carry: 1

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

So, when you have simulated, the output is coming like this. Let say one, let say 011 if we add them sum is 0, carry is 1; 111 sum is 1, carry is 1; you can verify the others 001 sum is 1, carry is 0. So, it is giving correct result, fine.

(Refer Slide Time: 17:15)

Data Values and Signal Strengths

- Verilog supports 4 value levels and 8 strength levels to model the functionality of real hardware.
 - Strength levels are typically used to resolve conflicts between signal drivers of different strengths in real circuits.

Value Level	Represents
0	Logic 0 state
1	Logic 1 state
x	Unknown logic state
z	High impedance state

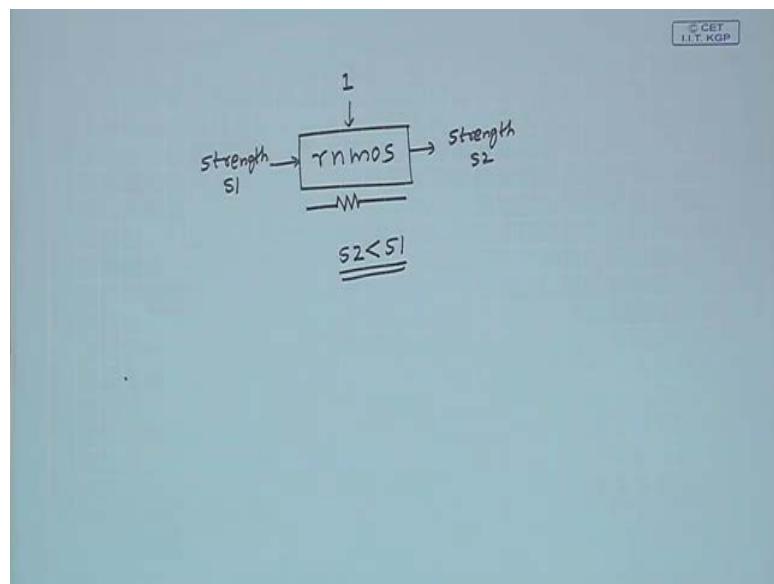
Initialization:

- All unconnected nets are set to "z".
- All register variables set to "x".

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

Now, we have seen that how we can use this kind of switch level primitives, the transistors, CMOS switches to design low level circuits gates and using gates you can design more complex things of course. But as I said if you are really want to designing circuits using low level CMOS or MOS transistors, you need to understand the concept of signal strength very clearly. Well, here as part of this course we shall not be going to the very details of the signal strength, but what I told you earlier means I am just trying to say the same thing. Suppose I use a primitive here which is let say rnmos, resistive NMOS, ok.

(Refer Slide Time: 18:03)



So, there will be a gate, there will be a source and a drain. So, this acts as switch. Suppose I have a signal here let say the input I have a signal of strength, let say s_1 and this switch is ON, in the control input I have applied logic 1, so, the switch is ON. So, the output I am getting another signal which is of strength s_2 . So, because this is resistive, so, even when this is ON, there will be a resistance in series, this s_1 , this s_2 value, strength of s_2 will be less than the strength of s_1 . There will be a degradation in strength, you have to remember this thing, right.

So, this is something which also we mentioned earlier. So, in Verilog there is 4 logic levels supported, you already know 0, 1, x and z. And 8 signal levels, these 8 levels are used to model the various kinds of scenarios for real hardware. Like you think, the strongest signals will be the one which are directly connected to the power supply lines.

If you take a signal directly from vdd or ground, those will be the strongest signals; you take the output of a gate, strength will be less; you take the output from a MOS switch which is resistive this strength will be even less; you take a pull up or pull down strength will be even less.

So, there are various circuit design scenarios where you can identify various different signal levels and depending on that you can actually do a calculation of the signal level. The simulation or the simulator does exactly that. Simulator understands that how signal strength values are degraded and how they get modified as they passed through different circuit modules and they will provide you the calculation of the simulation result accordingly, right.

(Refer Slide Time: 20:36)

Strength	Type
supply	Driving
strong	Driving
pull	Driving
large	Storage
weak	Driving
medium	Storage
small	Storage
highz	High impedance

↑
Strength Increases

- If two signals of unequal strengths get driven on a wire, the stronger signal will prevail.
- These are particularly useful for MOS level circuits, e.g. dynamic MOS.
- When a signal passes through a resistive switch, for example, its strength reduces.
- There is a convention followed for signal strength computation in MOS level circuits.
 - Details not discussed here.

Hardware Modeling Using Verilog

So, this is something again which we mentioned, these are the 8 signal values which are supported. Supply has the highest strength as I said then you say strong, pull, large, weak, medium, small and high impedance. High impedance has the lowest strength, high impedance is something which is like floating, a wire floating. I just float a wire in my hand I say what is the voltage here, there is no voltage because it is not electrically connected to anything.

So, the signal strength of the high impedance signal or a line or a variable will be the lowest and the supply values 1 and 0 that will be the highest. And there are many I mean intermediate values, there are some strength which are used for storage, some used for

driving other gates. So, I am not going to the details of this, but the point to notice that I also mentioned this earlier if two signals of unequal strengths drive a single wire then the stronger signal will prevail. Suppose there are 2 modules, the output of the 2 module is driving a single wire, but the output signals they have different strengths.

So, the signal which has highest strength, that will over ride the other and the output value will get it, but if they are of the same strength and their values are conflicting then the output will become indeterminate, x, fine. So, for MOS level circuit modelling this multiple values of signal strengths are useful. Like I will also mentioned this that when a signal will pass through a resistive switch, its strength will reduce. But the details as I said, details of signal competition the kind of conventions that are followed, that we have not discussed here.

(Refer Slide Time: 22:49)

Point to Note

- Most of the synthesis tools do not support switch level modeling.
 - Because it uses technology mapping from a given library.
 - Common gates and simple functional blocks are present in the library.
 - Thus it is not required to specify circuits at the transistor level.

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

So, the point to note finally is that well we use transistor level modelling for mainly simulation. For those of us who are into transistor level design, who have designed a circuit at the level of transistors, who want to simulate and see that well my design is correct or not, you can use these facilities which are provided in Verilog for switch level modelling to do that. But if your target is for synthesis, for example, if you are doing an FPGA mapping, see FPGA there is nothing like MOS, there is nothing like MOS transistor that everything you can map into are some modules and blocks. There call CLBs combination logic blocks, some flip-flops and so on.

So, synthesis tools will not be accepting any design which are at the level of the transistors. So, this is true not only for FPGAs, but also for ASICS. For ASICS, I told you typically the designs are created by picking up the cells from a technology library and the technology library already has some very well optimised transistor level layouts. The synthesis tool will not allow you to manipulate on that and give a different net list of the transistor level, of course let me tell you there are some designs where you may have to design something at the level of transistors. You may have to create a layout, but there are separate low level tools and software available for that. You do not use a high level synthesis tool that takes Verilog and translates it to hardware for that purpose, ok

So, most of the synthesis tools, not most means almost all of them you can say do not support switch level modelling. Because the final hardware that is generated it is not generated from the switch level models you have specified, they are usually taken from a library, ok.

So, circuits at the transistor level is not required to be specified if your final target is synthesis, all right. So, with this we come to the end of this lecture. So, we have seen very briefly the feature that is available in the Verilog language to model circuits at the level of MOS transistors, this is called switch level modelling.

Now, in the next lectures we shall be taking a case study, a fairly complex case study of a processor and you shall see how you can implement it using a pipeline fashion using Verilog.

Thank you.

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

Lecture - 37
Pipeline Implementation of A Processor (Part 1)

So, in the last few weeks we have seen how we can design and use the various features available in the Verilog language and through a number of examples we saw how we can model both combinational and sequential circuits. Towards the end we have seen how we can design efficient digital circuits say using the concept of pipelining let say. So, we saw that if we use pipelining then without investing more in terms of hardware we can gain a significant speed up.

Now, we shall be looking at a much more complex example namely that of designing a complete processor. See when I talk about a processor, I talk about something called an instruction set architecture. So, when you are using a processor or programming a processor, you have some internal registers and you have an instruction set and using those instructions, belonging to the instruction set, you can write a program to solve any problem, this is the idea.

So, in this lecture we shall be starting our discussion on the design of a processor in Verilog with an utilizing the concept of pipelining and the kind of processor that we look at is something called reduced instruction set architecture or reduced instruction set computer RISC, in short RISC which is rather easy to implement in hardware and in particular in a pipeline, ok.

So, the topic of our discussion in this lecture is pipeline implementation of a processor, the first part.

(Refer Slide Time: 02:21)

Introduction

- We shall first look at the instruction set architecture of a popular *Reduced Instruction Set Architecture (RISC)* processor, viz. MIPS32.
 - It is a 32-bit processor, i.e. can operate on 32 bits of data at a time.
- We shall look at the instruction types, and how instructions are encoded.
- Then we can understand the process of instruction execution, and the steps involved.
- We shall discuss the pipeline implementation of the processor.
 - Only for a small subset of the instructions (and some simplifying assumptions).
- Finally, we shall present the Verilog design of the pipeline processor.

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

Now, as I have said that we shall be looking at the instruction set architecture of a particular processor which belongs to the RISC category, reduced instruction set architecture. Now, now as them implies a RISC architecture or a RISC instruction set will consist of fewer number of instructions, simpler instructions and large number of registers, mostly general purpose registers, the result is that it is much easier to implement this kind of processors in hardware.

So, this is the reason why we have chosen a RISC architecture as the, you can say platform for our example. So, a few things about the processor the risk processor we are looking at is MIPS32, this is a well known processor, but actually we are not considering the entire instruction set of MIPS32, but rather a small subset of it because our objective is to show you how we can implement the processor in Verilog. Let us say we start with 50 instructions, but if I give you 50 instructions following the same principal, you can also implement them. So, the purpose of taking a small subset is to illustrate how we can design the processor, fine.

So, the important thing to look at before we think about implementation is what are the instruction types and how the instructions are encoded and after we have seen this, we shall be understanding the different steps of instruction execution, right. So, as I have said we shall be looking at the pipeline implementation of this particular processor using

a small subset of MIPS32 instruction set, not all the instructions, right and towards the end we shall be looking at how we can model this pipelined implementation in Verilog.

(Refer Slide Time: 04:38)

A Quick Look at MIPS32

- MIPS32 registers:
 - a) 32, 32-bit general purpose registers (GPRs), R_0 to R_{31} .
 - Register R_0 contains a constant 0; cannot be written.
 - b) A special-purpose 32-bit program counter (PC).
 - Points to the next instruction in memory to be fetched and executed.
- No flag registers (zero, carry, sign, etc.).
- Very few addressing modes (register, immediate, register indexed, etc.)
 - Only load and store instructions can access memory.
- We assume memory word size is 32 bits (*word addressable*).

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

Let us have a quick look at MIPS32 the main features, first thing is that there are 32 general purpose registers, they are called R_0 to R_{31} and each of this registers are of 32-bit in size.

Now, we use this registers for temporary storage of data during some computation because there are large number of registers 32. There is lot of flexibility available to the programmer and this register R_0 is a special register which is assumed to always contain the constant 0, means you cannot write any other value in to R_0 because in many application you need the number 0, so you can use R_0 for the purpose. There is a program counter, which is also a 32-bit register and a program counter you may be knowing this is a register which always points to the next instruction in memory which is to be fetched.

So, when an instruction is being executed, we have to fetch the instruction from memory, decode the instruction, what kind of instruction it is and then follow through the steps of execution and while we are doing this we will also be incrementing PC to point to the next instruction. So, that after our current instruction execution is over we can go back and fetch the next instruction, right, ok.

Now, in MIPS32 there are no flag registers as are present in many other processors, flag registers means zero flag, carry flag, sign flag, some of you may be aware and there is a very limited set of addressing modes, register immediate register, indexed and not many more. And another important thing is that the only instructions that can access memory are load and store and all other instructions they operate only on the CPU registers. They cannot use any data directly that stored in memory, you will first have to load the data in to a register and then use it in some computation, right. And another assumption we are making because everything is 32-bits to makes thing simple we are assuming that memory is word addressable meaning that every word has a unique address and the memory word size is 32-bits.

(Refer Slide Time: 07:23)

The MIPS32 Instruction Subset Being Considered

- Load and Store Instructions


```
LW R2,124(R8)    // R2 = Mem[R8+124]
SW R5,-10(R25)   // Mem[R25-10] = R5
```
- Arithmetic and Logic Instructions (only register operands)


```
ADD R1,R2,R3    // R1 = R2 + R3
ADD R1,R2,R0    // R1 = R2 + 0
SUB R12,R10,R8   // R12 = R10 - R8
AND R20,R1,R5    // R20 = R1 & R5
OR  R11,R5,R6    // R11 = R5 | R6
MUL R5,R6,R7    // R5 = R6 * R7
SLT R5,R11,R12   // If R11 < R12, R5=1; else R5=0
```

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

So, the instructions subset that you are considering are summarized. Firstly, there are load and store instructions. So, I am illustrating with examples how they look like, load instruction is like this load word, LW and this R2 is a register, R8 is a register and this 124, this is how we write. The meaning is the value of the R8 register is added to this number we have specified, we call this an offset, we add them up that is treated as a memory address, from that memory address we fetch or read the content of the memory location and the data is loaded into R2. So, the memory is loaded into R2.

Similarly, store word (SW), store word is reverse same is a register R5 its value will be stored in a memory location where same way the content of the register R25, you add -10

to it; that means, minus 10. these numbers will be represented in 2s compliment signed form, this will see later. There are some arithmetic and logic instructions which will be considered these few numbers where only register operands are used like, add R1, R2, R3. So, here the convention is that the first register is our destination, the other two are the source.

So, ADD R1, R2, R3 means add R2 and R3 store the result in R1. Similarly, ADD R1, R2, R0 means you add R2 to R0 which, which always contain 0 and store it in R1. So, you see, this is, you can see indirect way to move the content of a register in to another register, this effectively is copying the value of R2 into R1. So, when you need to do this, you can use this add instruction with R0. Subtract a similar, subtract R10 and R8 store in R12 and bit by bit AND; similar bit by bit OR; multiply R6, R7 store in R5. And this is a special instruction this is called set less than (SLT), set less than means it checks whether the second operation R11, R12 it is less than or not, if it is less than then the target register is set to 1, otherwise the target register is set to 0, ok.

(Refer Slide Time: 10:10)

- Arithmetic and Logic Instructions (immediate operand)


```
ADDI R1,R2,25    // R1 = R2 + 25
SUBI R5,R1,150   // R5 = R1 - 150
SLTI R2,R10,10   // If R10<10, R2=1; else R2=0
```
- Branch Instructions


```
BEQZ R1,Loop    // Branch to Loop if R1=0
BNEQZ R5,Label   // Branch to Label if R5!=0
```
- Jump Instruction


```
J Loop         // Branch to Loop unconditionally
```
- Miscellaneous Instruction


```
HLT             // Halt execution
```

These are some of the instructions and there will be some arithmetic and logic instructions where some offset or an immediate operand is given, like you have a version of add and subtract instruction, we call them add immediate, subtract immediate. So, how do use it, ADDI R1, R2, the last parameter is a number, this means you add the contents of R2 with 25, result store in R1.

Similarly, SUBI R5, R1, 150, so, R1 minus 150, store in R5. Similarly, there is a version of set less than immediate. So, you can right like this SLTI R2, R10, 10, so, here you compare with a R10 is less than 10 or not, if it is true then the target R2 set to 1 else R2 set to 0. There are 2 types of branch instruction would be considering, branch equal to zero (BEQZ) and branch not equal to zero (BNEQZ). Branch equal to zero means there is a register specified, we check whether this register is 0 or not, if it is 0 then we jump to loop otherwise we proceed with the next instruction. Similarly, not equal to zero, if R5 is not equal to 0 then you jump to this level. And there is some unconditional jump instruction J loop, it will always jump, but of course here we shall not be implementing this jump loop and the last instruction of our program will be halt instruction. So, it will stop the instruction execution.

(Refer Slide Time: 11:53)

MIPS Instruction Encoding

- All MIPS32 instructions can be classified into three groups in terms of instruction encoding.
 - R-type (Register), I-type (Immediate), and J-type (Jump).
 - In an instruction encoding, the 32 bits of the instruction are divided into several fields of fixed widths.
 - All instructions may not use all the fields.
- Since the relative positions of some of the fields are same across instructions, instruction decoding becomes very simple.

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

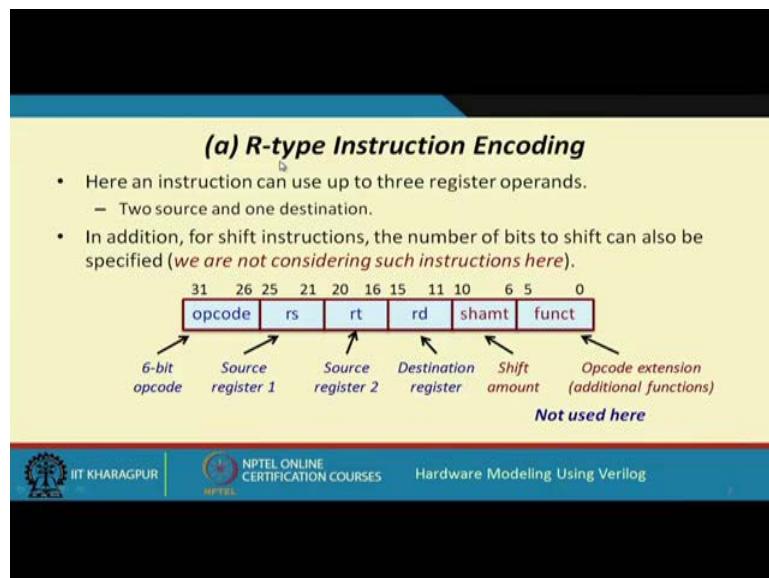
Now, it is important to know that how these instructions are actually encoded; that means, in terms of bits, I told that all word size in memory is 32-bits. So, every instruction will also be encoded in 32 bits. So, we will have to know what these 32-bits actually indicate, ok.

So, once you know it only then you can proceed with the hardware implementation or the Verilog modeling whatever you say. So, we need to understand how these instructions are encoded. Broadly speaking MIPS32 instructions can be classified into either register type, immediate type and jump type, of course we are not considering

jump type in our implementation because in the previous slide we said the J instruction we are not implementing so J instruction uses this type, right.

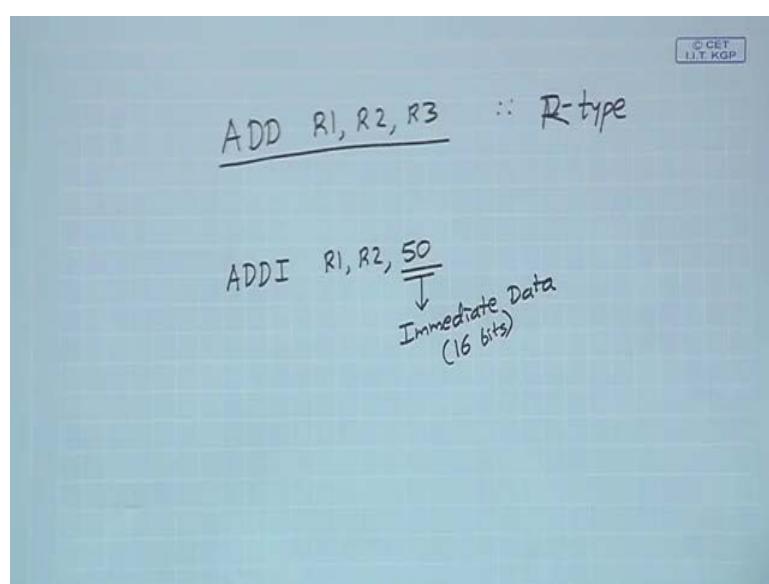
So, when you say encoding as I said the 32-bits of the instruction, they will be divided in to some fields and each field will be having some fixed widths and all instructions may not be using all the fields, let see that what this means.

(Refer Slide Time: 13:11)



Let start with the R-type instruction encoding, say - type instruction encoding is like the add instruction, ADD R1, R2, R3 like the instruction that I have said.

(Refer Slide Time: 13:23)



Some instructions like this where there are 3 register operands. So, you add R2 and R3, store the result in R1, this is an example of I-type instruction, not I, R-type, R-type instruction encoding, register type, right.

Now, in this type, how we are encoding in 32-bits you see, the first 6-bits, bit number 26 to 31, this is 32-bit word, 0 up to 31, this we are calling opcode. This opcode actually tells you, what kind of operation this instruction refers to because we are considering only a small subset of instructions 6-bits are sufficient for us.

So, in 6-bits actually we can represent 2^6 or 64 possible instruction, but in our case it is much less. So, 6-bits is more than enough for us. The next 3 fields, they indicate some registers, in R-type instruction there will be 2 source register and 1 destination register. The source registers are stored in the first 2 fields, it is called rs and rt, they are stored in bit numbers 21 to 25. So, recall there are 32 registers right R0 to R31. So, to address 32 registers uniquely, you require 5-bits because 2^5 is 32.

So, in all this register fields you need 5-bits each, 5-bits here, 5-bits here and 5-bits for the destination. So, the bits are given 11 to 15, 16 to 20 and 21 to 25. Now the last bits there used in MIPS32, but for our implementation we are not using them, because this 5-bits is normally used to specify something called shift amount. And last 6-bits is used to specify some ALU operations which is sometimes called opcode extension, but these two we are not using here in this implementation. So, let us ignore this, we will be setting all these bits to 0s.

(Refer Slide Time: 15:46)

• R-type instructions considered with opcode:

Instruction	opcode
ADD	000000
SUB	000001
AND	000010
OR	000011
SLT	000100
MUL	000101
HLT	111111

SUB R5 ,R12 ,R25

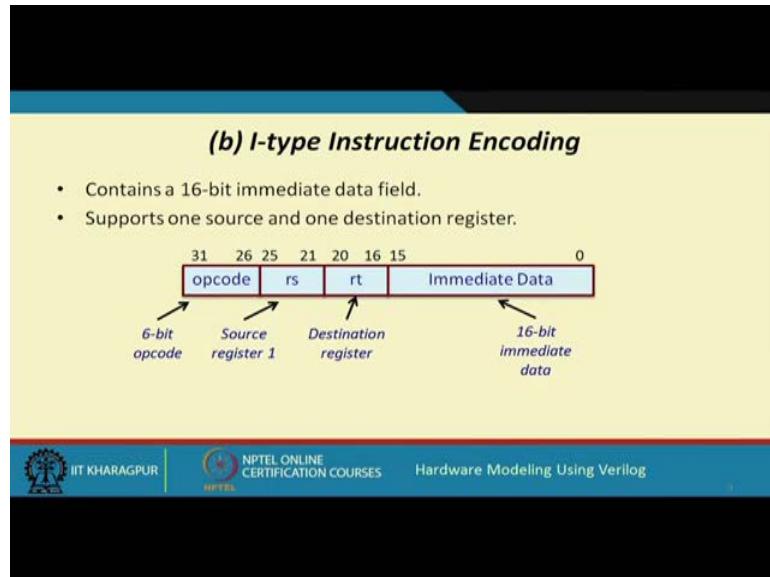
000001 01100 11001 00101 00000 00000
SUB R12 R25 R5
= 05992800 (in hex)

Let us take some examples. So, the instructions that we are considering, these instructions can fall in the R-type category they require 3 registers most of them, halt does not require any, 0 registers. So, halt we are also putting in this category because you can fit it in the same thing because opcode will be halt, other fields will all be blank, they will not be required. And we are assuming that these are the opcodes, all 0 is for ADD; 1 is for SUB; 2, 2 in decimal is AND; 3 for OR; 4 for SLT; 5 for MUL and all ones this is the actually 30, this is actually 63 in 6-bits, this will represent halt.

Now, an example of encoding let us take a subtract instruction, SUB R5, R12, R25. So, subtract, the opcode is 00001. So, when you convert it to the instruction format for subtract to write this 0001, then the 2 source registers R12 and R25, 12 in binary is 01100 this indicates register 12 and 25 is 11001 this indicate register 25 and destination is 5, 00101 is R5 and the last bits we are not using as I said these are blank.

So, if you break them up into four bits each and write in hexadecimal, this instruction in hexadecimal will be 05992800, right.

(Refer Slide Time: 17:30)



Fine, now let us look at something called I-type or immediate type instruction encoding. Let us take an example what kind of instruction talking about, let say add immediate. Say add immediate, we are saying R1, R2 then a number 50, this 50 is sometimes known as immediate data, that is why we call it immediate mode or I-mode. So, there will be 2 registers, the operation and a immediate data, now here this immediate data is encoded in 16-bits, right.

So, you see so how the instruction encoding is done here, again the first 6-bits are for opcode, the next 5-bits are for the source register, like in this case the source register is R2. In the example that we have given R2 is the source register and R1 is the destination register, right. So, the first field is for the source the second field is for the destination because we need only 2 registers and the last 16-bits is left for the immediate data, 16-bit immediate data. Sometime this also represents an offset for instructions like branch, this we shall see.

(Refer Slide Time: 18:57)

The slide contains a list of I-type instructions and their opcodes, followed by two examples of instruction encodings.

- I-type instructions considered with opcode:

Instruction	opcode
LW	001000
SW	001001
ADDI	001010
SUBI	001011
SLTI	001100
BNEQZ	001101
BEQZ	001110

LW R20, 84 (R9)

001000 01001 10100 0000000001010100
LW R9 R20 offset
= 21340054 (in hex)

BEQZ R25, Label

001110 11001 00000 YYYYYYYYYYYYYY
BEQZ R25 Unused offset
= 3b20YYYY (in hex)

At the bottom, there is a logo for IIT Kharagpur, NPTEL Online Certification Courses, and a portrait of a man.

Now, the instructions that fall under this I-type category are these load word (LW); load, store, they also belong to this category load, store, add immediate, subtract immediate, set less than immediate, branch not equal to zero and branch equal to zero and these are the opcodes that are chosen by us, 6-bits.

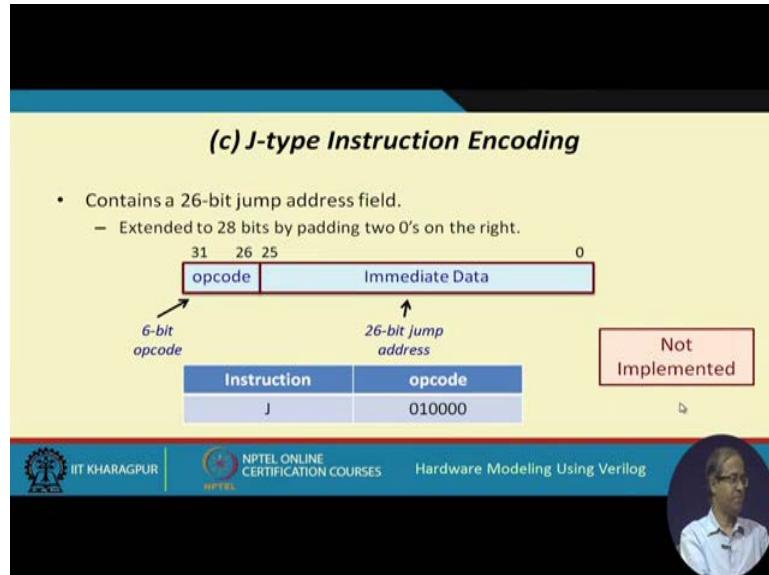
Now, some example encoding, take a load instruction. So, I said earlier there are load instruction looks like this. So, R9 is added to 84 that will be our memory address, from there data will be loaded into 20. So, load opcode is this, source is R9, 9, 01001; destination is R20, this is 20 and in the 16-bit offset we write 84, 84 is in decimal if you convert it to binary this will be the binary equivalent for 84, this you can check.

So, again if you convert this into hexadecimal, the hexadecimal form becomes 21340054, this instruction is encoded like this. Let us take another example, branch equal to zero, this I said it checks, there is only 1 register, here 2 registers are not required, only 1. So, if this R25 is 0 then you jump to level otherwise do not jump. So, here BEQZ the opcode is this 001110, there is only one source register R25, this is 25; but destination register in this instruction is unused, we leave it as all 0s and then the 16-bit is offsets. So, here we writing it yyy, this can be anything in fact.

So, the place where you want to jump will be specifying some offset here. So, what will happen is that this offset value will be added to the contents of the program counter and that way the address of the next instruction will be calculated, this is how branch will

take place, right and again this number if you convert to hexadecimal, it will be 3b20; the last y's, I write it yyyy, this is 3b2 and 0, fine.

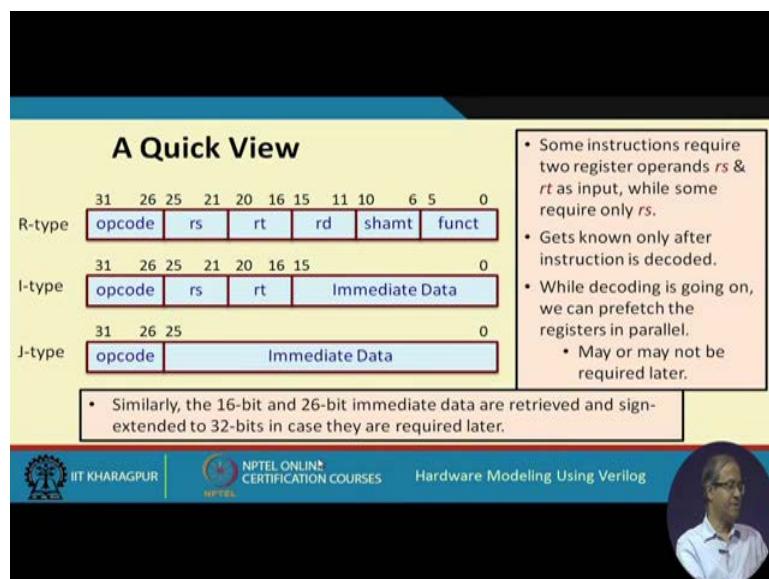
(Refer Slide Time: 21:24)



The J-type instruction that I had said that we are not implementing this, but actually J-type instruction looks like this. It consists of an opcode and a 26-bit immediate data, this is only for the jump instructions.

So, here we have also assigned a opcode, but in the implementation we will not be using this, this is not implemented.

(Refer Slide Time: 21:51)



So, a quick view of the three instruction encoding types. So, one thing you see, you forget J-type because you are not implementing this, we are not looking at J-type. But among the R-type and I-type one thing is very clear you see, the first 6-bits are opcodes; the next 6 bit is always the source; the next 6 bit is sometimes a source, sometimes the target destination and for R-type the next 6-bits is the destination.

So, what you saying is that we really do not know what type of instruction this is unless we decode the instruction, but the strategy that will be following is as follows to speed up things we will be assuming that there are two source registers. We will be taking the two source register numbers, there will be a register bank, we will be prefetching two registers for those number, register numbers. We do not know whether they are actually required or not, this will come to know later only after decoding, but in case they are required the registers can be fetched and they will already be available with us, this way time is saved, right.

So, now what we are saying is that I have just summarize it here, some instructions required two registers operands rs and rt, like add, subtract, while some required only rs like add immediate, subtract immediate. But only after decoding is completed that means when you decode the opcode, you will come to know exactly what kind of instruction is this. So, what I have just now said is that is a while this process of decoding is going on, you prefetch the registers assuming that these are, it is an R-type instruction assuming that you need both the registers. So, register numbers already there you access the register bank and prefetch them. But if later on we find that it is an I-type instruction then the second register that you have fetched, you will simply ignore it, only the first one will be used, this may or may not be used it to a later on.

Similarly, for the immediate data for I-type there can be 16-bit, well J-type you have not implemented, for J-type it will be 26-bits. So, we do something called sign extension, we will talk about it later, we convert it in to a 32-bit number and get ready. So, that later on when you need it already the values available in 32-bits.

(Refer Slide Time: 24:30)

Addressing Modes in MIPS32

- Register addressing **ADD** *R1,R2,R3*
- Immediate addressing **ADDI** *R1,R2,200*
- Base addressing **LW** *R5,150(R7)*
 - Content of a register is added to a "base" value to get the operand address.
- PC relative addressing **BEQZ** *R3,Label*
 - 16-bit offset is added to PC to get the target address.
- Pseudo-direct addressing **J** *Label*
 - 26-bit offset is added to PC to get the target address.

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

So, the addressing modes that are available MIPS32, this is very quickly summarized here. Register addressing an example is this where all the operands are stored in registers R1, R2 and R3, the destination in R1. The immediate addressing, one of the operand is an immediate data 200. Base addressing, there is a register, there is a number, you are adding this number to the register to get the effective address or the operand address, this is base addressing.

PC relative addressing branch, so, whatever offset is specified here that is added to the program counter to get the, to get the actual address of branch and jump of course, you are not consider it is very similar, this 26 offset will be added to PC. So, with this we come to the end of this lecture where we gave you an over view of the processor that you are trying to design. Just one thing you remember some of you may or may not be having proper background in computer architecture and organization, for those of you have already done a course on computer architecture or organization you will be understanding much clearly what I am talking about. I am talking about a processor, I am talking about a instructions, decoding, steps of execution and so on.

But if you are a designer wanting or trying to learn the language Verilog you just stay with me, you need not understand all the things that I am talking about, but ultimately you try to understand what is the problem that you are trying to model and how we are modeling in Verilog, you are implementing it in pipeline and what is the process

involved. In the next lecture we shall be continuing from here, we shall be talking in more detail about the steps of instruction execution and how we can design the, you can say controller you can say, how the instruction execution is done inside, ok.

Thank you.

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

Lecture - 38
Pipeline Implementation Of A Processor (Part 2)

So, in the last lecture we started our discussion on the MIPS32 instruction set architecture; a small subset of it and was talking about the instructions that we are actually going to implement as part of these lectures.

Now, in the present lecture we shall try to see what are the steps involved in instruction execution. We shall not be talking about pipelining now; we shall be looking into pipelining later. Right now you forget pipelining, but what we are trying to understand and learn is that; for any instruction it can be add, it can be load, it can be branch; what are the steps that need to be carried out inside the processor so as to execute or fulfil the functionality what is actually required of the instruction. So, this is our part two of the discussion on this.

(Refer Slide Time: 01:22)

MIPS32 Instruction Cycle

- We divide the instruction execution cycle into five steps:
 - a) IF : Instruction Fetch
 - b) ID : Instruction Decode / Register Fetch
 - c) EX : Execution / Effective Address Calculation
 - d) MEM : Memory Access / Branch Completion
 - e) WB : Register Write-back
- We now show the generic micro-instructions carried out in the various steps.

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

Let us come straight to a breakup of the instruction cycle; now talking about a instruction cycle, instruction cycle refers to a time period that is required for the execution of a complete instruction. Now, here we are saying because the processor or the instruction

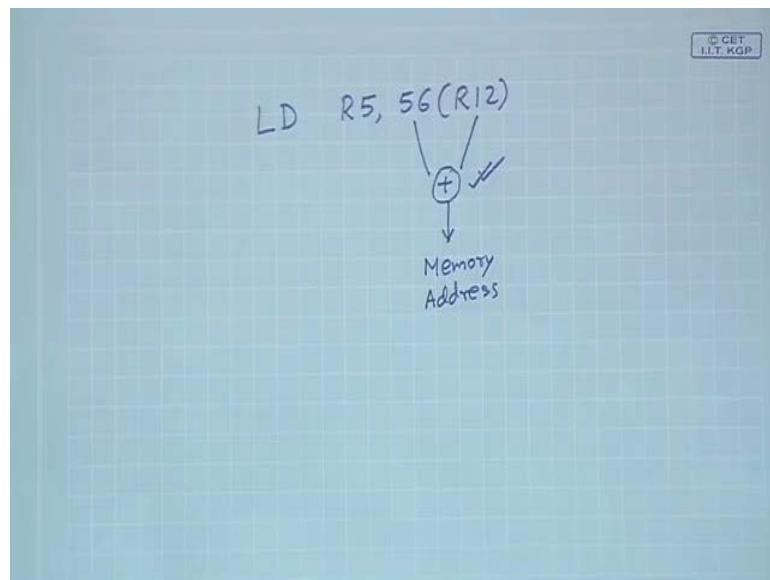
set that we are trying to design and implement is very simple; this instruction cycle can be very regularly divided into 5 different you can say sub cycles or steps.

So, what we are saying is that; the total instruction execution cycle we divide into these five steps. So, what are these 5 steps? First step is called Instruction Fetch; so, here we are fetching an instruction from the memory. So, we know that the program counter, the PC register already contains the address of the next instruction. So, simply whatever is there in PC from that memory location we read that is instruction fetch.

Then second step is instruction decode; here we are trying to decode the Opcode and find out what kind of instruction it is? Is it add, subtract, multiply or what. And in parallel while decoding is going on as I have said; we also do some register prefetching. Well we are assuming that the instruction will be requiring both source registers, we will be prefetching the registers, not only that assuming that there will be a 16-bit immediate data, we will take that last 16-bit of that instruction and we will be doing a sign extension to 32-bits. These things we are doing in anticipation; we really do not know whether these things will actually be required or not, it will be known only after decoding of the instruction is complete.

And the third step is execution, where actually an arithmetic logic unit is used. Here we execute the instruction or for some instructions, we can we have to compute the effective address.

(Refer Slide Time: 03:56)



Well you think of a load instruction; the load instruction that we talked about was like this; load let us say R5, 56, R12. So, the value of 56 and R12 they will be added and the result of the addition will be your memory address, from where the data will be loaded into R5.

So, you see you require an addition step here; in the load instruction no other arithmetic calculations required, only this addition to compute the effective address or the operand address. So, that is, what is mentioned here in the EX stage; for such load and store instructions, we can also do the effective address calculation. Then in MEM, we can actually do the memory access, read and write from memory and also for branch instruction, we can also decide whether to branch or not and the last stage WB, it is register write back.

Let us say I have an ADD R1, R2, R3 instruction; so, R2 and R3 will be added; the result will be stored in R1. So, the result stored in R1; this will be taking place in the WB stage last stage, WB stands for Write-back, Register Write-back.

(Refer Slide Time: 05:40)

(a) IF : Instruction Fetch

- Here the instruction pointed to by *PC* is fetched from memory, and also the next value of *PC* is computed.
 - Every MIPS32 instruction is of 32 bits.
 - Every memory word is of 32 bits and has a unique address.
 - For a branch instruction, new value of the *PC* may be the target address. So *PC* is not updated in this stage; new value is stored in a register *NPC*.

IF: <pre>IR ← Mem [PC]; NPC ← PC + 1;</pre>	For byte addressable memory, PC has to be incremented by 4.
--	---

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

And a different steps that are going on even within each of these steps, there are several sub steps, these are called micro instructions and micro operations. We shall be showing you what are the micro operations that are required in the different steps, you will be able to understand it.

Let us start with instruction fetch; here I have said that the instruction pointed to by PC, PC always contains the address of the next instruction to be executed. So, whatever PC contains, you fetch that word from memory and store it in an internal register, it is called an instruction register, you see it later, but this fetch is going on. And we are assuming this already we have mentioned earlier that every instruction is 32-bits long, also every memory word is 32-bits and has a unique address so that after an instruction has been fetched, you will have to increase PC by 1 to point it to the next instruction.

So, what you do? This is the instruction fetch MEM (PC), this is memory just like accessing array, something similar to that. You are accessing memory, this address is in PC and you are storing into a register called IR, this IR stands for instruction register. And you increase the PC and do not store it back in PC because for jump instruction, the target address can be something else you do not know it yet. So, you use another register called new PC, NPC store it temporarily in NPC, this is what we do during the instruction fetch stage.

Now, just one thing for some memory systems, the memory words are byte oriented means every byte has a unique address. So, if we talk about 32-bit instructions, so, we will have to increase PC by 4 to go to the next instruction because there are 4 bytes in every instruction, but for our example, we are considering word addressable to keep it simple.

(Refer Slide Time: 07:47)

(b) ID : Instruction Decode

- The instruction already fetched in *IR* is decoded.
 - *Opcode* is 6-bits (bits 31:26).
 - First source operand *rs* (bits 25:21), second source operand *rt* (bits 20:16).
 - 16-bit immediate data (bits 15:0).
 - 26-bit immediate data (bits 25:0).
- Decoding is done in parallel with reading the register operands *rs* and *rt*.
 - Possible because these fields are in a fixed location in the instruction format.
- In a similar way, the immediate data are sign-extended.

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

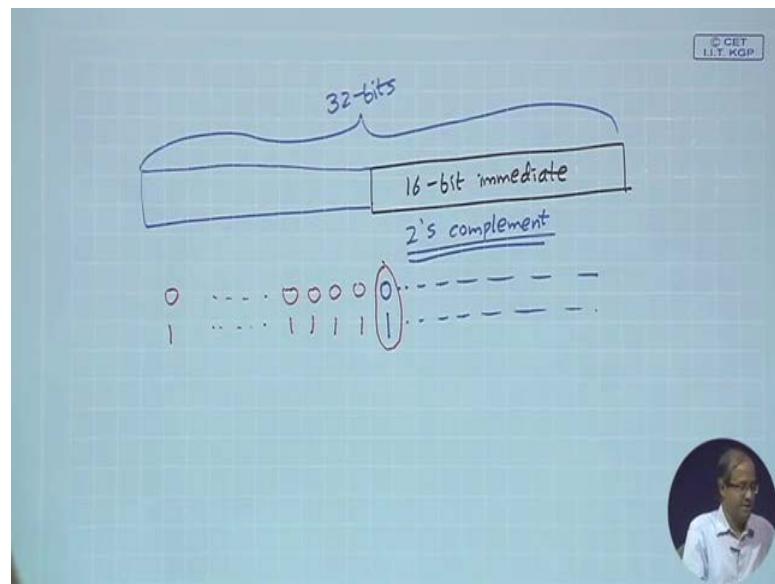
Next let us come to instruction decode; now we have seen that the instruction format is very regular, the opcode is always in the first 6-bits, bit number 26 to 31. The first source operand, so wherever is there will be in bit number 21 to 25 and the second source operand in bit number 16 to 20. 16-bit immediate data, 0 to 15 and 26 immediate data for jump instruction, we are not using this so, ignore this for the time being. So, decoding and register perfecting rs and rt will go on in parallel. Similarly, immediate data will be sign extended, let us see what happens.

(Refer Slide Time: 08:33)

These are the micro operations; we are doing register prefetch you see, the two fields are rs and rt. So, we are writing it like this Reg [rs], Reg [rt]; we are fetching the register, there are two read ports. So, in the earlier examples of register bank we saw, you recall we assume that there are two read ports and one write port, now you can correlate why we need that.

You see here we are reading two things at the same time in the same cycle. So, two register are fetched, one you store in A, other you store in B. And this is how sign extension is carried out, the last one you can ignore because this we are not implementing, let us try to understand what we are doing here.

(Refer Slide Time: 09:36)



If you see we have a 16-bit immediate field, we have to extend it to 32-bit. So, what we are doing is as follows; so, we have a 16-bit immediate field. Now, this 16-bit immediate field is considered to be a signed number in 2's complement, this is how it is regarded 2's complement signed number.

So, if the number is positive, it will start with 0, 0 then anything. If the number is negative, it starts with 1 then anything. Now, sign extension means we want to extend this number to 32-bits, to make it 32-bits. Now in 2's complement, the rule for sign extension is very easy, it says whatever is the sign bit, you replicate that. If the number is positive just add 16, 0's in the beginning, that will make same number in 32-bits. For negative numbers similarly if the sign is 1, you replicate this first 16-bits with 1, the value remains the same.

So, sign extension means just this, you replicate the sign bit 16 times. Now, this we are representing it like this notationally, you see IR, the last 16-bits I am writing as 0 to 15 like this and this 15th bit; IR_{15} is a sign bit. This we are replicating it 16 times, to the power 16 means it is just a notation, 16 times replication and this double hash means concatenation.

Well of course, when you write in Verilog; we have some other ways of representing, there those curly brackets, concatenation operation, replication operation those are there.

But this is a symbolic notation not in Verilog, I am just writing it in symbolic notation and this A, B, Imm etc, these are all temporary registers which will be required later.

(Refer Slide Time: 11:56)

(c) EX: Execution / Effective Address Computation

- In this step, the ALU is used to perform some calculation.
 - The exact operation depends on the instruction that is already decoded.
 - The ALU operates on operands that have been already made ready in the previous cycle.
 - *A, B, Imm, etc.*
- We show the micro-instructions corresponding to the type of instruction.

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



Then comes the executions step EX; here we either execute the arithmetic and logic operation whatever is asked for or we calculate the effective address for load and store instructions. Basically in this step the ALU is carrying out some operation, now exact operation will of course depend on the instruction and the operation will be carried out on numbers which are already generated by the previous stage A, B, Imm. So, it will be operating on these numbers A, B, Imm etc.

(Refer Slide Time: 12:45)

Memory Reference:
ALUOut \leftarrow A + Imm;
Example: LW R3, 100(R8)

Register-Register ALU Instruction:
ALUOut \leftarrow A func B;
Example: SUB R2, R5, R12

Register-Immediate ALU Instruction:
ALUOut \leftarrow A func Imm;
Example: SUBI R2, R5, 524

Branch:
ALUOut \leftarrow NPC + Imm;
cond \leftarrow (A op 0);
Example: BEQZ R2, Label
[op is ==]

The slide also features the IIT Kharagpur logo, NPTEL Online Certification Courses logo, and a portrait of a man.

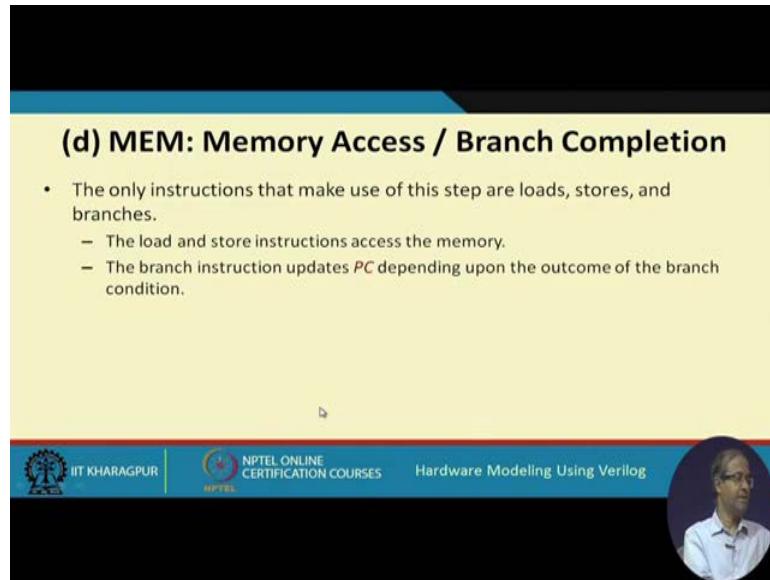
Let us see depending on the type of the instruction, how the steps in the EX stage can vary. Well if it is a memory reference I will load and store instructions; so, I have shown an example of a load. Here we calculate the effective address in this step. You see A will contain the value of R8. There is a source register and Imm will consist of this immediate data.

So, A plus Imm, this R8 plus 100 is calculated and the effective address is temporarily stored in a register called ALUOut. Similarly, for register-register ALU operation like this let us say, SUB R2, R5, R12; here A is R5 and B is R12 and func is actually the kind of the function. SUB means this is subtraction. So, you actually carry out the operation specified by the instruction on the operands which will be A and B and again the result is stored in ALUOut. And if it is an immediate operand, there is immediate operand, register immediate, then it is similar to this because second operation is not B, second operand will be Imm; R5 will be A and 524 is Imm.

And for branch, you do these things, here the level of whatever is the offset that will be the last 16-bits, that will be your Imm, that will be added to NPC. Because you still now do not know whether you are taking the branch or not taking the branch. But in case you take the branch, where to go, you are making that ready, you are adding the program counter that NPC with this offset and getting ready with the new address in case you decide later that well I have to take a branch.

And also you calculate the condition because there is branch equal to zero; you check whether A, R2 is a equal to 0 or not and that result of this comparison you store in a flip-flop, a 1 bit register call cond, this will be required or used in the next stage.

(Refer Slide Time: 15:06)



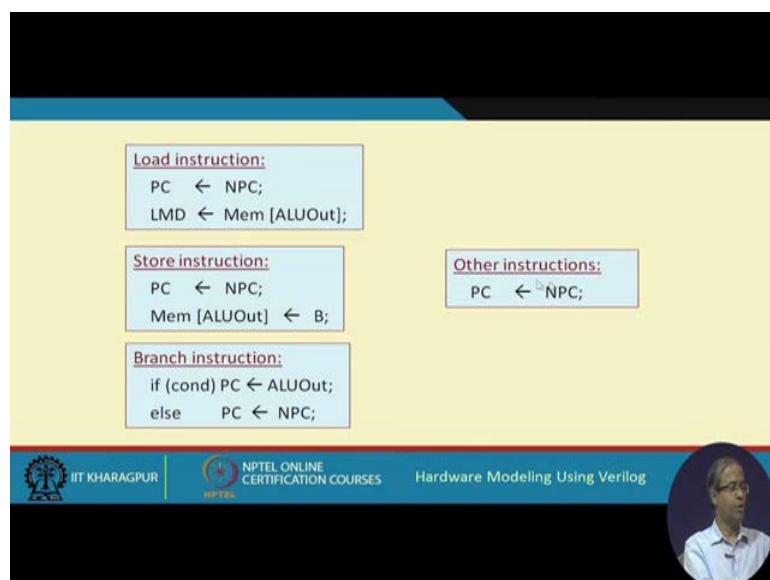
(d) MEM: Memory Access / Branch Completion

- The only instructions that make use of this step are loads, stores, and branches.
 - The load and store instructions access the memory.
 - The branch instruction updates *PC* depending upon the outcome of the branch condition.

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

Now, come to the MEM: MEM here we do memory access or branch completion. So, only load, store and branches, they use this stage other instructions they do not do anything in MEM, let us see what is to be done.

(Refer Slide Time: 15:23)



Load instruction:
 $PC \leftarrow NPC;$
 $LMD \leftarrow \text{Mem [ALUOut]};$

Store instruction:
 $PC \leftarrow NPC;$
 $\text{Mem [ALUOut]} \leftarrow B;$

Branch instruction:
if (cond) $PC \leftarrow ALUOut;$
else $PC \leftarrow NPC;$

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

For the load instruction, you simply do this because load instruction the PC will be the NPC. So, just the NPC you load into PC and you do a memory read because in the previous stage already the address of the operand is stored in ALUOut. So, you access memory with that address and the data that is coming out, you temporarily store in an register LMD, load memory data, this is also temporary register.

For store instruction; it is just the reverse well first thing is again NPC goes to PC and B whatever is there is in B that you store in MEM, ALUOut. This is what you do because for a store instruction, your B is the target, second operand is the target, that will be stored here, ALUOut and for branch instruction in the earliest stage we have already computed cond.

So, now you check if cond is 1, then there is a branch. So, the new value of address you have calculated that goes to PC or otherwise this NPC just the next instruction plus 1, that goes to PC, this is what is done during the MEM stage.

(Refer Slide Time: 17:00)

(e) WB: Register Write Back

- In this step, the result is written back into the register file.
 - Result may come from the ALU.
 - Result may come from the memory system (viz. a LOAD instruction).
- The position of the destination register in the instruction word depends on the instruction → *already known after decoding has been done.*

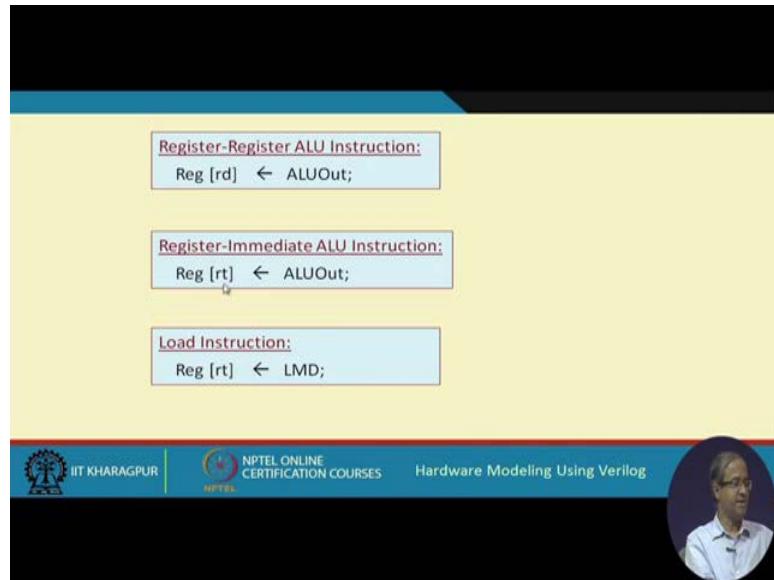
R-type	31 opcode	26 rs	25 rt	21 rd	20 shamt	16 funct	15 0
I-type	31 opcode	26 rs	25 rt	21 20	16 15	15 Immediate Data	0

IIT Kharagpur | **NPTEL ONLINE CERTIFICATION COURSES** | **Hardware Modeling Using Verilog**

And for all other instruction, just you copy NPC to PC. Lastly in the WB stage, you have to write back the register to store the result. Result is written back to the register, now the result that you are writing back, it can either come from the memory like a subtract, add, multiply instruction or it can be load instruction. Now the data is coming from memory, because you see earlier the data from the memory is coming into the register for LMD, so, from LMD it will go.

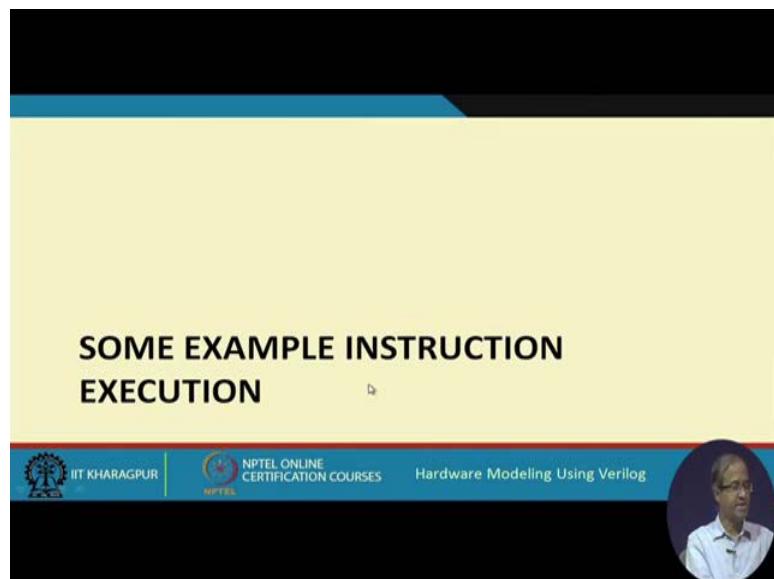
Now, one thing you see for a R-type instruction, your destination is rd from bit number 11 to 15 and I-type instruction, your destination is rt bit number 16 to 20. So, depending on the type of the instruction, you will have to decide on the destination register.

(Refer Slide Time: 18:00)



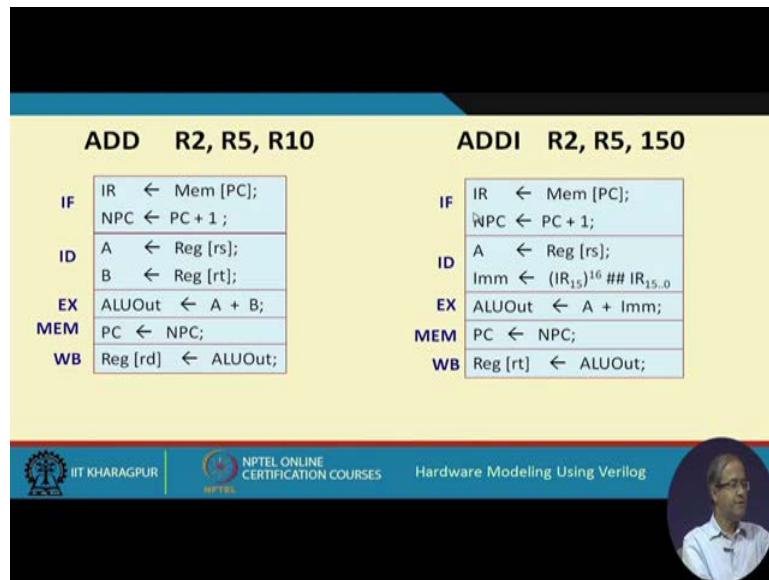
So, let us see, if it is a register-register ALU instruction, that means, this one, first one. So, rd will be your register target, so, ALUOut will be stored in Reg [rd], but if it is a register immediate, it will be stored in Reg [rt] because rt is the target.

(Refer Slide Time: 18:32)



Similarly, for a load instruction; the data is stored in LMD temporarily, that data will be loaded into Reg [rt]. So, this you have to remember use rd here, and rt here. Now, let us look at sum instruction execution in completion, the complete set of micro operations. So, if you look at this you will understand what is happening.

(Refer Slide Time: 18:38)



Take this instruction, ADD R2, R5, R10. So, R5 and R10 is added, result is stored in R2. Let us see in the five stages what is happening, for IF you are fetching this instruction in IR and incrementing PC that goes to NPC. Second stage well Imm, I am not showing because this instruction does not require Imm; it only requires Reg [rs] and Reg [rt]. So, they are stored in A and B. So, in the EX stage, it will decode it is an ADD, so, there will be an addition operation A plus B will be carried out, the result will be stored in ALUOut.

MEM, nothing other than NPC goes to PC, this copying nothing else is done and in the right back whatever ALUOut is computed that will finally be written to the register rd, rd is R2. So, in this way this instruction gets executed, these are the steps. Let us take another example, this is an immediate, add immediate instruction, there is an immediate operand here.

So, instruction fetch is identical, so, fetch an instruction, increment the PC. Here there is only one source operand R5. So, I am not showing B, I am only showing A, the relevant

ones I am only showing, A equal to Reg [rs], this is rs and immediate data is 150. So, in Imm, this 150 gets sign extended, you store in Imm.

In EX, you add A and Imm, so, R5 and 150 gets added. MEM, same NPC goes to PC and WB, similar, but instead of rd, here it goes to rt, because in this instruction format only two registers are specified rs and rt.

(Refer Slide Time: 20:42)



Let us look at a load instruction, LW R2, 200 (R6). So, here this 200 will be added to R6 to get the memory address from where data will be loaded into rt. So, IF is again same as the earlier instructions. This ID is also same as in add immediate, the first operand R6 is loaded in A and the immediate data 200 is loaded and sign extended Imm. In EX, for load and store instruction, this will always be add. So, this 200, R6 are added, result in ALUOut.

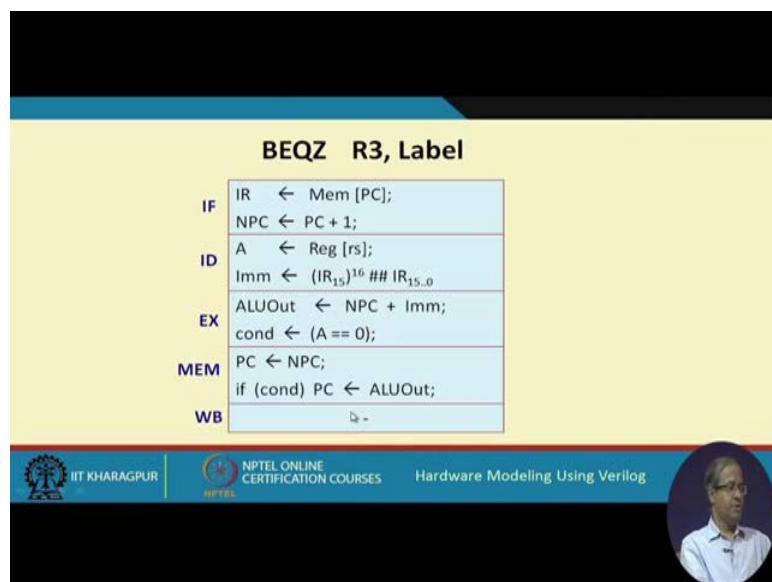
And in MEM well other than NPC to P, another thing is actually done, that means memory access. This ALUOut actually contains the memory address, MEM of that, that is read and the data get stored in LMD and lastly in WB that LMD is written into register rt, where rt is R2.

Now, lastly let us look at a store instruction; it will be quite similar, but in store instruction you see in the last step there is nothing to do. First time IF is again identical; well in ID, well here you are actually loading rs and rt both, R3 and R10, it depends on

the implementation, this is just one implementation I am showing. Now in our actually implementation, we will make it a little different.

So, let us say the first one is R3 is rs, R10 is rt let us say, that is the convention I have shown here in this. This ALUOut, A plus and same way we compute the address and here we write. B contains this R3, so, B will actually get stored in, R10 is A, R3 is B. So, so when you say store, this B will get stored into the memory and this store operation is complete within MEM. So, in WB you do not have anything to do now, this is a gap.

(Refer Slide Time: 23:09)

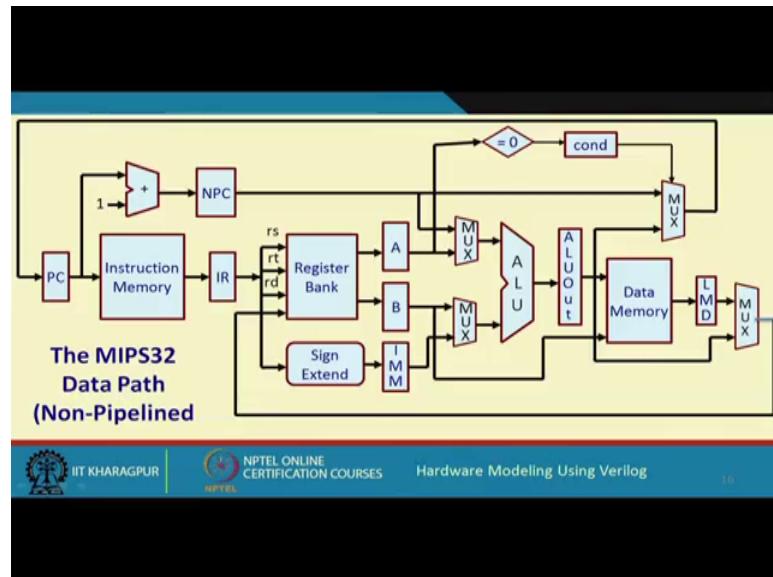


And one example of a branch instruction; branch equal to zero, it checks whether R3 is 0 or not, if it is 0 then it jumps to level. You say what is done here, instruction fetch is again same; instruction decode R3 is fetched into A and immediate whatever offset is there that is sign extended in Imm. In EX stage, we are adding R3 with this, not R3 it is sorry, we are just adding the program counter, the NPC with the offset to compute the branch address in case you decide to branch later. So, you are adding this two up.

And also you are checking the branch condition because the branch condition was equal to zero. You check this R3 is already loaded in A, you check whether A is equal to equal to 0 or not, just a comparator. In the result of the comparison you store in this flip-flop 1-bit register cond. In MEM, you check this cond; NPC, PC is of course, the first thing that you do, but you may have to override it, if cond your ALUOut goes to PC. So, here you

can also write it like this, if cond PC equal to ALUOut else PC equal to NPC, you can also write like that. WB again there is nothing for branch.

(Refer Slide Time: 24:41)



Now, you see whatever we have talked about just over all the data path can be very conveniently represented or drawn like this. This is our non pipelined data path of the MIPS32. You can identify the different stages, you see here this is the instruction fetch, where this is the memory which stores the instruction, PC is the address and you read the memory. The instruction is read out, you store it in the instruction register IR and there is also an operation NPC equal to PC plus 1. So, there is an adder that adds 1 to PC and result stores into NPC; so, this is your instructions fetch.

Now, instruction decode of course, the decoder is not shown. So, in the IR that opcode part of IR is decoded and the rs and rt part, the two source registers, they are prefetched. One of them is loaded in A, another is loaded in B and the last 16-bit offset, that is sign extended, that is stored in Imm.

In the EX stage; you operate on two numbers, well of course the function will depend on the instruction type, I am not showing that whether it is add, subtract, multiply what. But the data that you are operating on that can be different, the first data can either be A or NPC, NPC will be for the branch instruction recall because NPC will be added to something. So, it can be either NPC or it will be A.

The second multiplex will be selecting B or the immediate data, this also depends on the instruction and the ALU stores the result in ALUOut. And in the EX stage I said also you have a comparator which checks whether A equal to 0 or not and the result you store in a 1-bit register cond. In the MEM stage, you actually do either a read or write. So, the memory address is obtained from ALUOut and in case you want to do a write, the data is in B, B is written here.

And in case you do read, the data read will go to LMD and this is the write back stage, WB .Well see WB is nothing extra just as a multiplexer which writes back something into the register bank because register bank is already here there is a third write port using rd. So, rd is already known that what was the destination register and the data can be either coming from LMD for load instruction or from ALU instruction it will come from ALUOut. That data will be stored in register, so, you see this is the overall architectural execution process of the MIPS.

Now, because the instruction set was so simple because the instruction encoding and decoding is so simple. You see the whole data path I can fit in a single slide, but you think of a more complex processor, is it possible to do this? No you cannot, you can never do this for a more complex instruction set.

So, for the RISC or reduced instruction set computer this is the main advantage. Implementation becomes very very simple. So, this architecture which I have shown, so, for the data path, for a non pipeline case. In the next lecture we shall be showing how this can be partitioned into well defined pipelines, what will be the stages and how we have to modify the micro operations to suit the pipelining requirements. So, with this we come to the end of this lecture, we shall be looking at how to extend this data path to a pipeline data path in our next lecture.

Thank you.

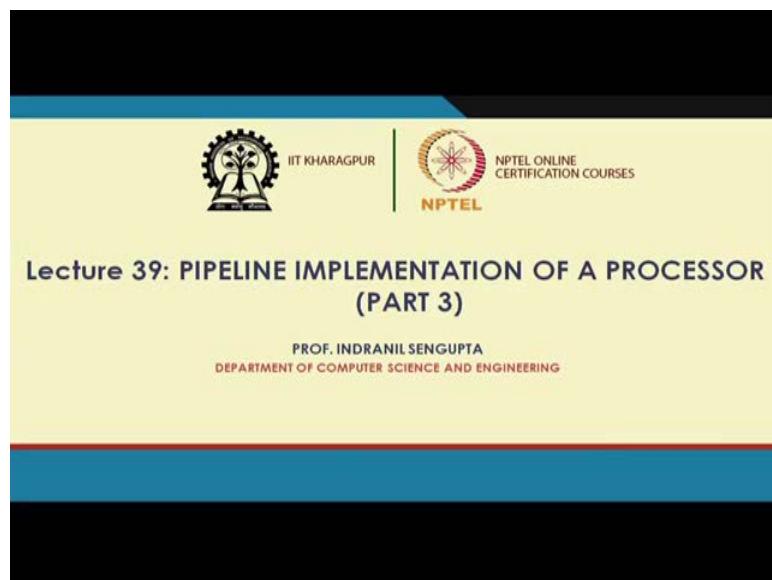
Hardware Modeling using Verilog
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 39
Pipeline Implementation of a Processor (Part 3)

So, now in this lecture, we shall be starting from where we left in the last lecture and we shall be looking at what kind of modifications we need in the micro operations and also in the data path, if we want to pipeline or MIPS32 instruction set architecture implementation. Now means one thing is happening naturally because earlier we talked about 5 different stages of execution or steps of execution.

So, for the pipeline implementation also, we shall be retaining that same definition for the pipeline stages, the same 5 steps will be converting into pipeline stages.

(Refer Slide Time: 01:08)



(Refer Slide Time: 01:12)

The slide has a yellow header bar with the title "Introduction". Below it is a bulleted list of requirements for pipelining the MIPS32 data path:

- Basic requirements for pipelining the MIPS32 data path:
 - We should be able to start a new instruction every clock cycle.
 - Each of the five steps mentioned before (IF, ID, EX, MEM and WB) becomes a pipeline stage.
 - Each stage must finish its execution within one clock cycle.

Below the list is a diagram of a MIPS32 pipeline. It shows five stages: IF, ID, EX, MEM, and WB. Each stage is represented by a light blue rectangle with a red "T" inside, indicating a pipeline register. Horizontal arrows connect the stages. Above each stage is a small red rectangle labeled with a Greek letter Δ, representing the pipeline delay. The entire diagram is set against a yellow background.

At the bottom of the slide, there is a footer bar with the IIT Kharagpur logo, the text "NPTEL ONLINE CERTIFICATION COURSES", and the course name "Hardware Modeling Using Verilog". To the right of the footer is a circular portrait of a man.

So, this is the title of our talk. So, first we look at what are the basic requirements for pipelining the data path that we have seen in the last lecture. So, what are the things to be done?

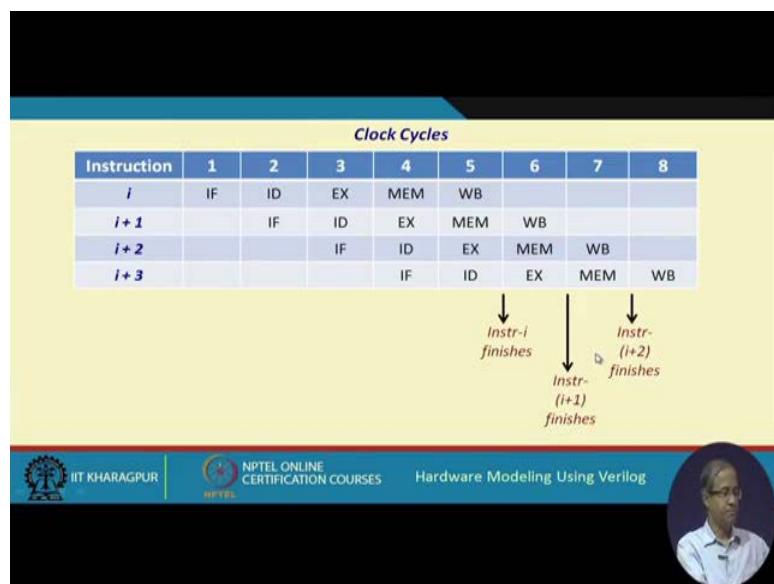
Well, in a pipeline what we have seen earlier through some examples, the main motivation for having a pipeline is that our input data is arriving continuously, once every cycle. So, we should be able to feed one new data every cycle. Now, in this case, we are talking about instructions, our instructions is like our data, in every cycle a new instruction is coming, a new instruction is fetched. This is how the pipeline will be executing and this is called a, this is referred to as instruction pipeline because this is a pipeline where instructions are executing and instructions are moving from one stage to the other IF, ID, EX, MEM, WB, fine.

So, the one requirement is that we should be able to start a new instruction every cycle because otherwise this pipelining will not give you the desired advantage and as I said the 5 steps that we had mentioned, these 5 steps, we will make them as 5 pipeline stages IF, ID, EX, MEM and WB.

Now, of course, one thing is important you see there is a global clock, with clock the data and the results are moving. So, every stage should be able to finish its execution within a clock cycle. So, that is another requirement, each stage must finish its execution within a clock cycle.

So, our overall pipeline will look something like this. This will be the 5 stages and there will be pipe, there will be pipeline latches separating the stage, just like we saw earlier for isolation we need this latches because when the first instruction as move from IF to ID, its temporary data are stored in this latch. So, ID can very peacefully do the instruction decoding because these data are stored in the latch and mean time, this second instruction can be fetched in IF. But if you do not have this latch while IF is going on this data will, this will keep changing and so the process of decoding might get disturbed, just for isolation we need the latches.

(Refer Slide Time: 04:08)

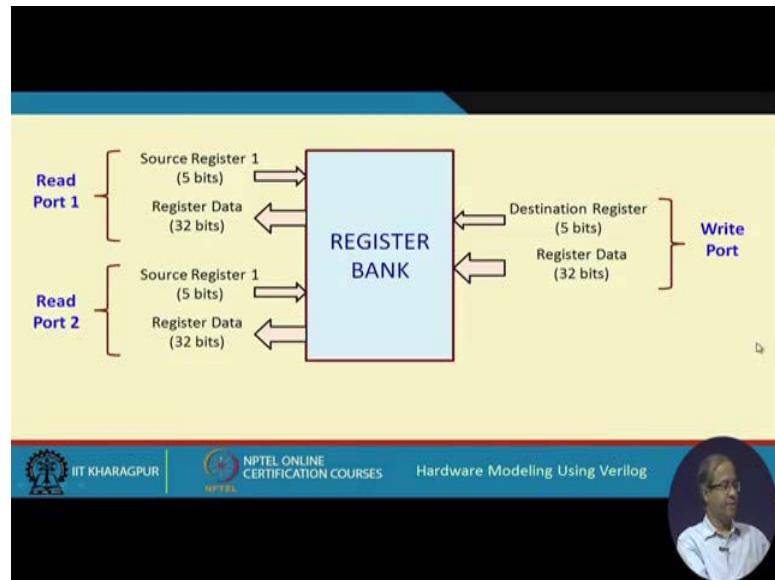


Now, in instruction pipeline the advantage that you get pictorially is depicted like this, you see in this direction we show the time the clock cycles. Let us say our instruction number i is coming, let us start with clock cycle number one. So, it is fetched, decoded executed, MEM and WB, it is finished. Now in pipelining what we do when the first instruction i moves from IF to ID, the next instruction in sequence, it can start IF. So, in time step 2 the first instruction is in ID, the next instruction is in IF. In third step the first instruction has moved to EX, second in ID and third in IF. In this way you see fourth step MEM, IX, ID, IF; fifth clock cycle WB, MEM, EX, ID.

Now, say after this WB the first instruction instead of now i will finish. So, it will finish here in after 5 cycles and after 6 cycles, $i+1$ will finish because WB is finished here and after seven cycles $i+2$ is finish and so on. So, here the ideas that you initially give some

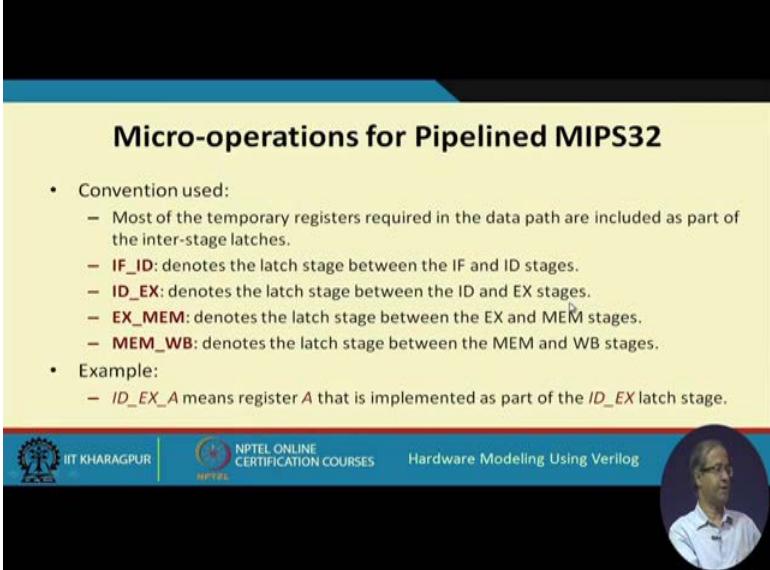
time for the pipe to fill up and after it gets filled up, you will be getting one instruction completed per clock cycle, this is the advantage that we gain in pipelining, we also saw the example earlier. If there are 5 stages, you can in the ideal case expect 5 times speed up, but of course, in instruction pipeline there are other constraints well talk about it later that limit the speed up, it may not be exactly 5 in fact much less than that and $i+3$ here.

(Refer Slide Time: 06:21)



And this is a picture you also saw earlier, for this case also we need a register bank that will be having 2 read ports and 1 write port. So, there will be 32 registers that is why the source and destination registers are 5-bits each and each register is 32-bits long. So, the register data are all 32-bits, right.

(Refer Slide Time: 06:49)



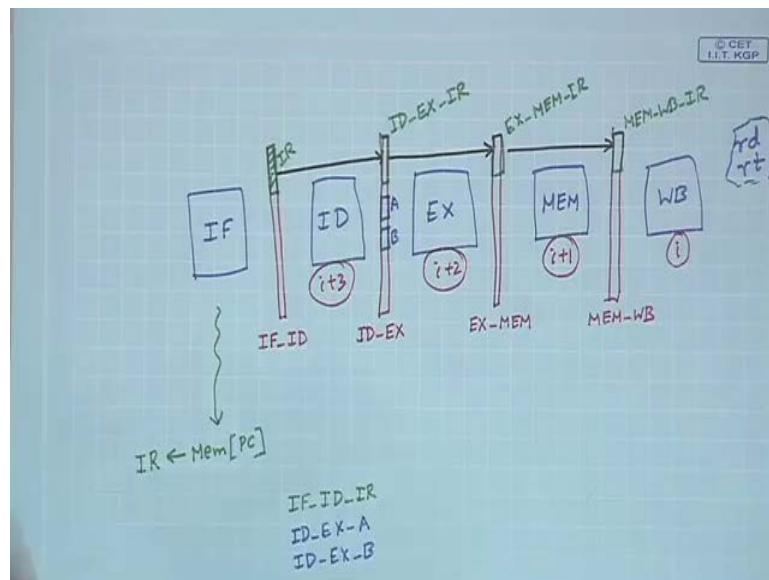
The slide title is "Micro-operations for Pipelined MIPS32". It contains a bulleted list of conventions and examples:

- Convention used:
 - Most of the temporary registers required in the data path are included as part of the inter-stage latches.
 - IF_ID**: denotes the latch stage between the IF and ID stages.
 - ID_EX**: denotes the latch stage between the ID and EX stages.
 - EX_MEM**: denotes the latch stage between the EX and MEM stages.
 - MEM_WB**: denotes the latch stage between the MEM and WB stages.
- Example:
 - ID_EX_A** means register A that is implemented as part of the **ID_EX** latch stage.

The footer includes the IIT Kharagpur logo, NPTEL Online Certification Courses logo, and a portrait of a man.

Now, let us try to modify the micro operations for the pipelined MIPS32 version. Now the first step is that we need to follow some conventions for this because the conventions that we are talking about it with respect to the naming conventions and we will be consistent with name, that means whatever we show the slides the same convention will be following when we show the Verilog codes.

(Refer Slide Time: 07:22)



Now, the idea is as follows. You see you have this 5 stages instruction fetch, instruction decode, execution, MEM and write back, WB. Now what I am saying is that you see

there will be before and after, but the most important latches are those between these stages. So, for our interest we need these 4 inter stage latches, we give some names to these latch stages, we call this stage as IF_ID, that means a latch stage that is sitting between IF and ID. This latch stage we call as ID_EX; we call this EX_MEM and we call this MEM_WB. So, this is the conventional we follow the latch stage, we will be representing using this underscore notation between which pair of stages this latch stage is, fine.

So, this is the convention that I have just talked about IF_ID, ID_EX, EX_MEM and MEM_WB and another thing the temporary registers that we have talked about let us take an example. Now in the IF stage, just look at the IF stage, in the IF stage you recall there was a micro operation which says that instruction register was loaded with memory location whose address was in PC.

So, the data must be stored in some register IR. So, where is IR? So, IR we read the some separate register, inside IR is actually known, actually just like in the example that we showed earlier that all temporary registers will be part of the latches register stages, let us say this one, this itself can be IR. So, and this register actually will be referring to as IF_ID_IR, this is the kind of naming convention we will be following.

And in case this register is also forwarded here let us say may be this IR will require later also. So, here also there will be an IR and this IR will be called as ID_EX_IR and this same IR will also be forwarding here. So, here will be calling it, there will be an IR register here, we calling it EX_ME_IR and again this will also be forwarded here this will be calling MEM_WB_IR.

Now, the reason I have taking the example of IR is, there is a good reason for this. You see this IR is just one example, there will be many other registers which will also be just handled in same way. For example, in the instruction decode when you prefetch the register bank, there will be one register called A, another register called B, there will be similarly named IF, this will be ID_EX, this will be ID_EX_A and ID_EX_B.

Now, the reason that we have to forward it like this you see instruction register will contain the entire instruction format, all the source register, destination register. For some of the instructions you need to take the decision only at the end, let us say write back in for some instructions you are writing in to register rd, for some other instructions

you are writing into rt. So, both the values of rd and rt should be available to you that is present in this instruction IR and you need to forward it because when instruction i has reached WB. Let us say here you have instruction number i, your next instruction is here, next instruction is here, and i+3 is here and so on.

So, whatever IR is stored here that is the IR of instruction i and whatever IR is stored here that is the IR of instruction i+1, this is the IR of instruction i+2 and this will be the IR of instruction i+3. So, this explains why we need to forward the same register across stages so many numbers of times because there are so many instructions moving and each of them will be needing their own IR, their instructions are different, so, their IR will also different, that is why.

(Refer Slide Time: 13:10)

```

(a) Micro-operations for Pipeline Stage IF

IF_ID_IR      ← Mem [PC];
IF_ID_NPC,PC ← ( if ((EX_MEM_IR[opcode]==branch) & EX_MEM_cond)
                  {EX_MEM_ALUOut}
                  else {PC + 1} );

```

So, now let us straight away look at the micro operations for the different stages well assuming that we are having a pipeline implementation. First the instruction fetch you see here whatever we are showing we shall be directly translating them into Verilog code later. We are fetching the next instruction MEM [PC], earlier we mentioned that we are storing it in IR, but now we are saying it will be IF_ID_IR, this IR will be as part of the IF_ID latch.

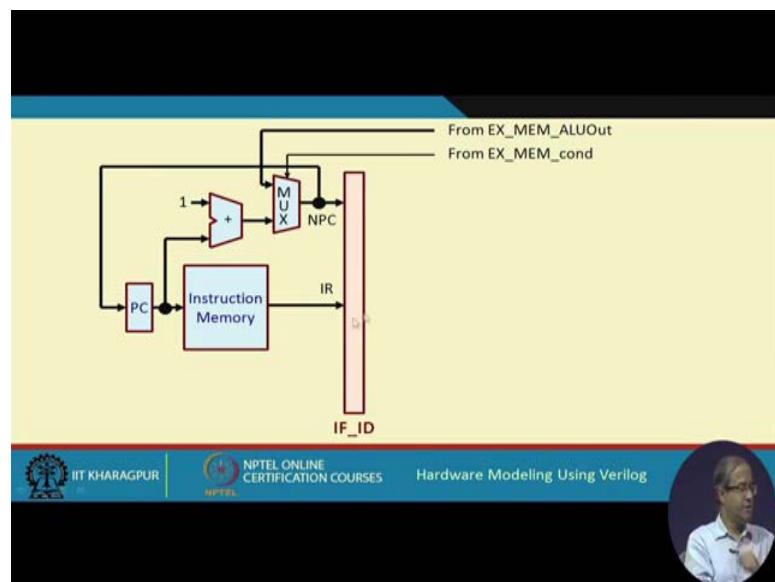
Then we check whether the opcode is a branch and the branch condition is true, you see a branch instruction is fetched, decoded, but the branch condition will be known only when it has reached the EX stage. Just you see this diagram once more, see an instruction

a branch instruction, so, once it reaches the EX stage, only then you know that is a branch instruction and the values of the cond and the next instruction address will be stored in this buffer.

So, if you decide that there is a branch. So, the next instruction has to be fetched from that address. So, there has to be some kind of a feedback from this back to here, next instruction fetch will depend on the outcome of these conditions of the branch, right that is reflected by this micro operation you see. Here it is seeing if the instruction registered in the EX_MEM, you see EX_MEM is this one, EX_MEM opcode was a branch and the corresponding condition, branch condition was true then only branch will take place, you see.

So, if EX_MEM opcode is branch and EX_MEM branch condition was true then whatever branch targeted this if calculated in ALUOut, EX_MEM_ALUOut that will be assigned both to IF_ID_NPC and also to PC else if this branch, it is not a branch or the branch condition is not true then simply PC plus 1 will be assigned to these two, fine.

(Refer Slide Time: 15:59)



So, in terms of hardware the instruction fetch stage will look like this, this will be your IF_ID stage instructional; IR will be part of this; NPC will also be part of this and as I said from MEM out stage EX_MEM stage these two signals are coming, right. So, you are just calculating the NPC accordingly and after calculating NPC that same value also goes to PC. You see previous one PC and NPC both are getting the same values else PC

plus 1 if the branch condition is, if cond is not true, then their lower input will be selected from the multiplex that is PC plus 1 that PC plus 1 will go back to PC plus 1.

Now, see one thing is true here see for a non pipeline implementation we were storing the next address in NPC much later NPC was copied to PC, but in a pipeline we cannot afford to do that why, in every cycle we are fetching a new instruction. So, we will have to continuously update the PC in every cycle that is why we made this change here, in every cycle we are updating also the PC, right. So, the PC is getting updated directly here, fine.

(Refer Slide Time: 17:37).

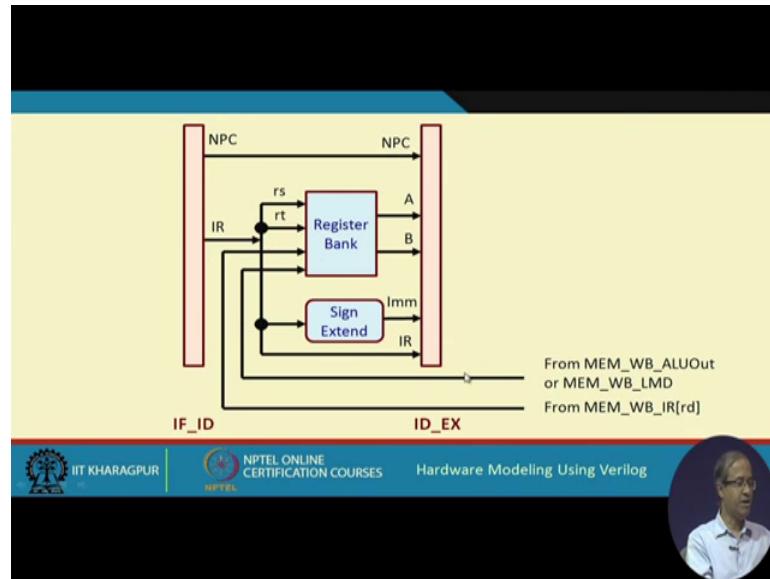
(b) Micro-operations for Pipeline Stage ID

```
ID_EX_A ← Reg [IF_ID_IR [rs]];
ID_EX_B ← Reg [IF_ID_IR [rt]];
ID_EX_NPC ← IF_ID_NPC;
ID_EX_IR ← IF_ID_IR;
ID_EX_Imm ← sign-extend (IF_ID_IR15..0);
```

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

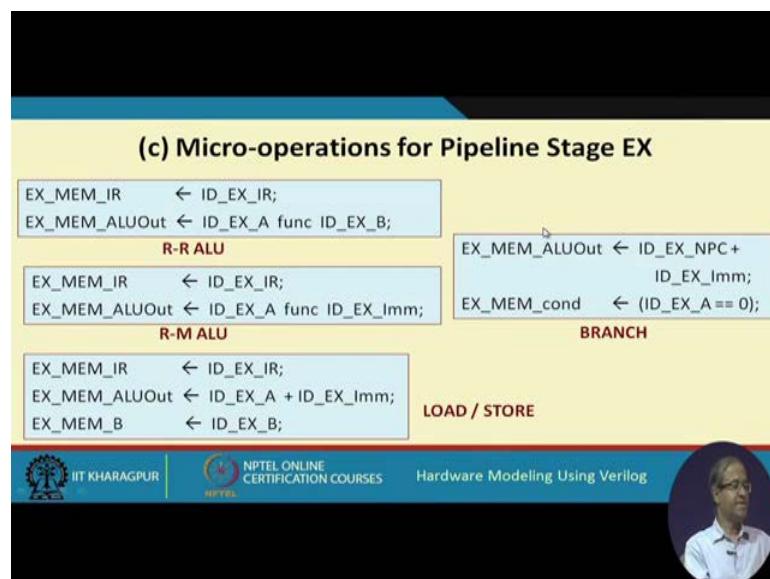
Now, let us move to the ID, well ID is very similar only the variable naming convention has changed. See Reg earlier we said Reg [rs], but now we are saying IF_ID_IR [rs] of that IF_ID_IR [rt], the two registers are prefetched, they will be stored in ID_EX_A and ID_EX_B. Similarly, NPC is just copied, IR is just copied as I said and Imm is sign extended. So, IF_ID_IR these bits will be a sign extended to ID_EX_Imm.

(Refer Slide Time: 18:20)



So, in terms of the hardware, it will look like this, register bank. So, NPC will be copied, this IR will be copied, that Imm part of the IR will be sign extended to get Imm; rs and rt, there will be prefetch register bank will go to A and B and from MEM_WB_ALUOut or LMD the data will be coming for writing the register bank and from MEM_WB_IR the target will be coming, this will be either rd or rt. So, here we although written rd, but this can be rt also depending on the type of instruction, this is how it will be in this stage.

(Refer Slide Time: 19:08)

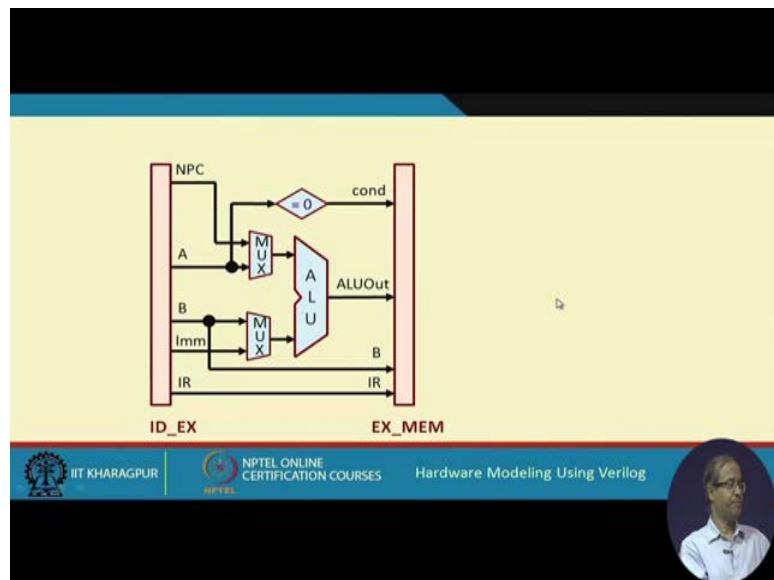


Similarly, for the EX stage depending on the instruction type micro operations will vary. For register-register ALU operation first this IR has to be copied across every stage, IR is simply copied and A, the function whatever add, subtract, multiply be restored in ALUOut. But these variables are all prefixed by the pipeline latch number, similarly raised memory ALU_IR is copied and here instead of B it is Imm.

For load store IR is copied, this A is added to immediate data to compute the effective address and B is also copied because for store instruction, this B will be stored. So, B also you need to copy forward. So, that in the MEM stage you can execute the store instruction and for branch you add NPC and Imm because NPC was coming and you check the condition A equal to equal to 0 that condition can be either true or false that is stored in cond, EX_MEM_cond.

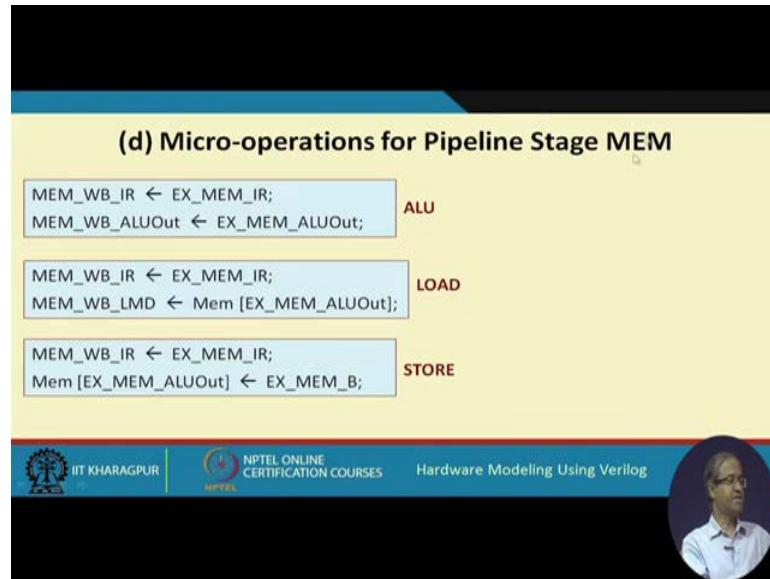
You recall this EX_MEM_cond actually is being used here you see earlier this EX_MEM_cond I showed here in the fetch instruction, fetch stage this is coming from there.

(Refer Slide Time: 20:39)



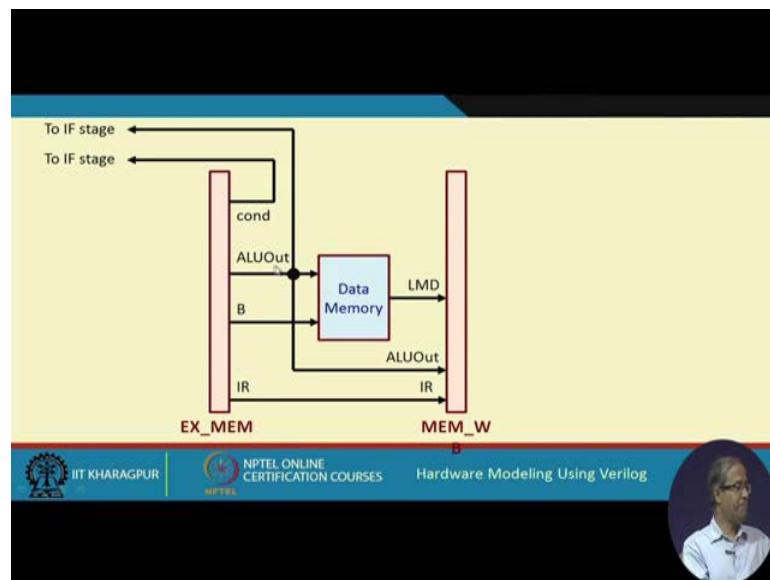
So, the EX stage the hardware will look as simple as this, ALU with some multiplexer, it can be either come from NPC or these are all part of this latch NPC, A, B, Imm, IR. So, all these registers are here and in the output here, cond is there, ALUOout is there, B is there, IR is there.

(Refer Slide Time: 21:01)



Now, let us come to MEM, for ALU operation again IR is forwarded and also ALUOut is forwarded nothing else to do in MEM, simply forward the data because you will be using them in the next stage, load IR is forwarded and do a memory read. So, ALUOut contains the address, you read it and store in LMD. For store you see B was forwarded earlier, this B is getting written to memory, ALUOut is the address, right. So, this will be your MEM.

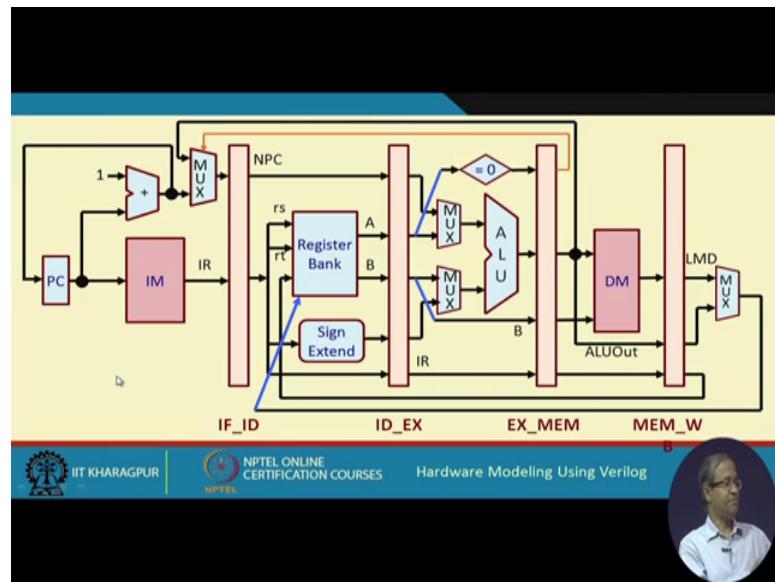
(Refer Slide Time: 21:38)



ALUOut is the address and B is for write and LMD is for read and from this cond and this ALUOut going to the previous stage, as we showed earlier and IR is again forwarded and also ALU is forwarded to the next stage.

Lastly for WB you see here both LMD and ALUOut are used that is why LMD is forwarded, LMD is there, ALUOut is there and also IR is there because IR you require rd or rt. So, for register-register ALU operation then ALUOut will be stored in rd that register; for register memory it will be stored in rt; for load LMD will be stored in rt, fine. So, here your WB is nothing but a multiplexer where we will be selecting either LMD or ALUOut depending on a control signal and it will be going to your means ID stage. So, it will this IR will contain which register number and this will contain what is the data.

(Refer Slide Time: 22:57)



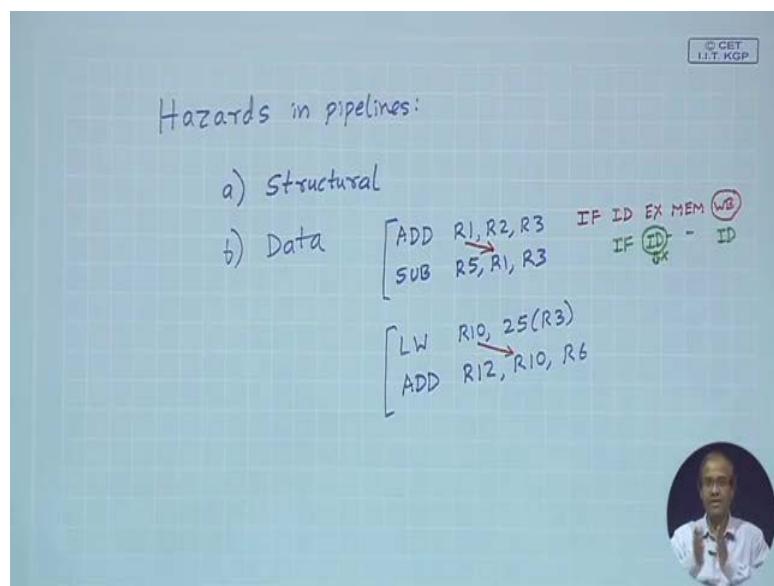
So, if you combine everything together whatever we have discussed so far, you see the picture looks like this. Just see the simplicity of the thing the whole processor pipeline is fitting within a single slide. Let me just make it bigger this, yeah, you see this is your instruction fetch, decode, execute, memory, WB. Just let us proceed with one just example let us say ADD R1, R2, R3. So, what happens in IF, PC contents the address of the instruction it is fetched. So, the instruction is loaded in IR, here the 2 operands are pre-fetched A and B, the 2 source operands in. For EX, this A is selected, B is selected, they are added, result will be stored in ALUOut. For memory, it does nothing, ALUOut

is simply forwarded and for WB this ALUOut will be written back into the register bank and this IR will contain which register number.

You see this is the simple data path that we have shown. I just one thing I wanted to clarify here you see here I am not trying to teach you what is the processor, what is a pipeline processor, how to design a pipeline processor basically I am taking it as an illustrative example to tell you how we can model complex pipelines in Verilog. Now well because we have taken the example of an instruction pipeline, let me tell you a few things.

Well, for a pipeline you may imagine that well if I have a k-stage pipeline, I can have a speedup of k potentially when there are large number of data coming one after the other. But when you talk about instructions, there are a number of other problems that may come, there are situations which are called hazards.

(Refer Slide Time: 25:13)



These hazards can happen in pipelines, first kind of hazard can be structural hazard. So, what is the structural hazard? structural hazard means 2 instructions which are already there in the pipeline, they are in 2 different stages, but they are trying to use the same hardware resource. If there is a single copy of the hardware resource they cannot access it together. So, one of them will have to wait and the other can proceed, this is an example where you may have to insert a so called stall cycle in a pipeline, that means one clock cycle may get wasted.

Like you see an example here in this pipeline in the IF you are fetching instructions from memory and in MEM for load and store instruction also you are accessing memory. Suppose if you had a single memory then you could not do this two things together that is why you see I have separated them out, instruction memory and data memory. I have separated out the two memory so that this kind of structural hazard can be removed. So, instruction will be stored in one memory, data will be stored in another memory something like that we can assume.

There are other more important kind of hazards like there is something called data hazard, data hazard can arise due to instruction dependency. Let us take just some examples, let us say I have a just ADD R1, R2, R3 followed by a instruction let us say subtract let us say R5, R1, R3, this is one example. Let us take another example load R10, let us say 25 [R3], ADD R12, R10, R6, this is another example.

So, you see one thing here the first instruction is generating R1 which is been used in the second instruction, here it is generating R10, it is used in the second instruction. So, you think like this, first instruction add this was going through instruction fetch, instruction decode then execute, MEM, then write back. So, under normal circumstance this R1 will be written result in written only during WB.

But for the next instruction, you see SUB instruction here fetching will be done here and we will be trying to do instruction decode and register pre-fetching here, but because R1 is not yet ready. So, here you will not be able to fetch the correct register value. So, you will have to wait, maybe possibly you can do your instruction decoding and pre-fetching here while this is being written. So, 2 cycles you may have to stall in between.

Same is the case for here, you are loading an instruction into memory, this will be done only in the MEM stage and in the next case you are using that value R10. So, again R10 written in WB, again something similar will happen. Well there are methods to reduce the impact of this kind of hazard there is something called data forwarding and other things. But this is something you have to remember, later on when you say some examples on the Verilog implementation, we will see that for sample programs we have to deliberately insert some dummy instructions in between such that the result is correct otherwise this kind of hazard will take place and the next instruction will try to read

some wrong value instead of the value which is being computed by the previous statement.

And the third kind of situation is called control hazard this arises because of branch instructions, like you see a branch instruction has entered the pipe. Now you take the decision of the branch only after EX, right. So, when the branch instruction has entered the EX stage already 2 other instructions are already in the pipe. So, when you decide that you have to branch. So, those 2 instructions have to be discarded. So, these are some reasons why you cannot utilize the pipeline to the fullest possible extent and there can be some stall cycles. But again, I am telling you in this class, in this sequence of lecture our objective is not to teach you computer architecture, but rather we want to teach you how to model this kind of pipelines in Verilog.

So, all these things we are just assuming that these are required that us why you are doing it, we are not going into the detail or we are not talking about any analysis on that. So, with this we come to the end of this lecture, from the next lecture onward we will be actually looking at the Verilog implementation of this processor that we have discussed and talked about so far.

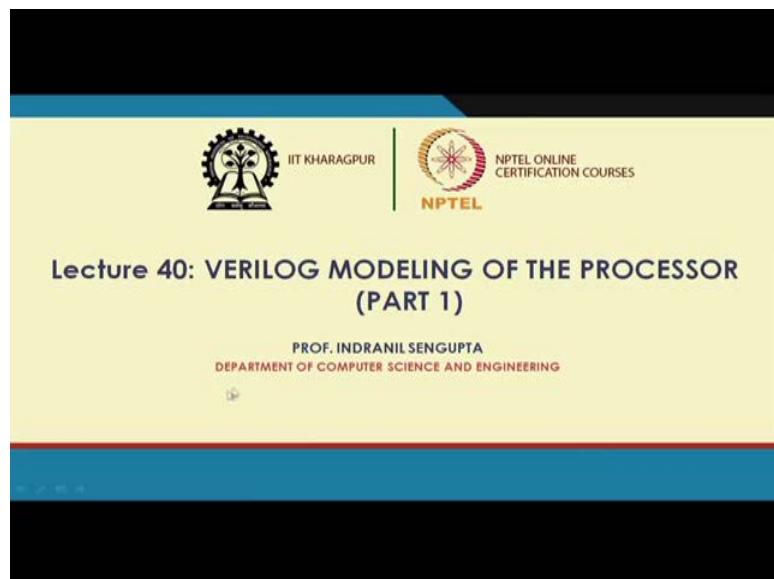
Thank you.

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

Lecture - 40
Verilog Modeling of the Processor (Part 1)

In the last few lectures, if you recall we were discussing how to implement a particular processor. Now, as an example, we took them MIPS32 reduced instruction set computer architecture and we looked at how we can have a pipelined implementation of a subset of the MIPS32 instruction set. Now, in this lecture what we shall be looking at that whatever we talked about with respect to pipelining of the MIPS32, there you recall we had identified the five stages instruction fetch, decode, execute, memory access and write back. And we also mentioned what are the required micro operations for the different stages. So, we shall now show how we can translate the description as we have presented in the last lecture into Verilog code directly.

(Refer Slide Time: 01:24)



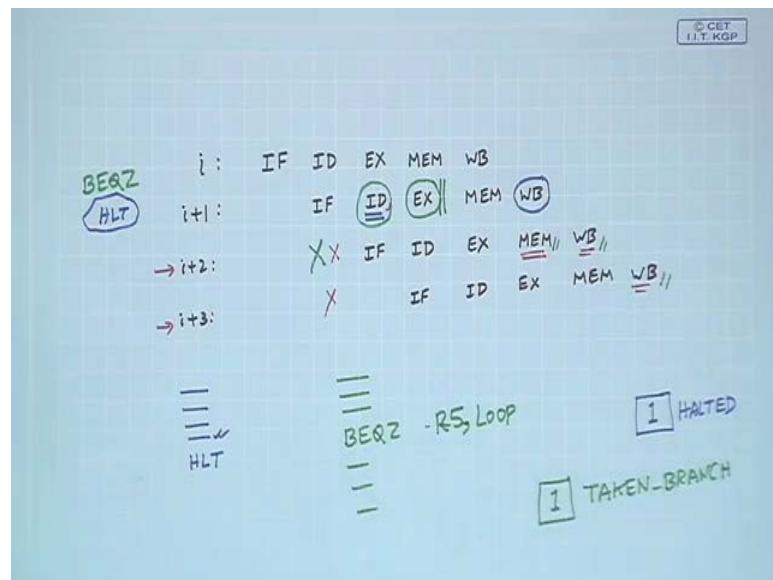
So, the topic of our lecture today is Verilog Modeling of the processor, the first part of it.

(Refer Slide Time: 01:28)

Verilog Implementation of MIPS32 Pipeline

So, when we translate the micro operations that we have discussed in the last lecture into Verilog there are a few things to remember. So, here we have identified two variables and they are used for very specific purpose, but before that let me just tell you once more that how instructions get executed in a pipeline.

(Refer Slide Time: 02:00)



Suppose, I have instructions i executing, so it will be going through fetch, decode, execute, memory access and write back then instruction $i+1$ comes, so it will be delayed by 1 cycle, like this, then $i+2$ comes it will be again be delayed and let say $i+3$ comes,

fine. Now, let us suppose that we have a sequence of instructions that we have written and the last instruction is a halt instruction. So, when we execute halt, the machine is supposed to stop. Let us just assume that instruction $i+1$, let say was the halt instruction. Let us assume, this was the halt. So, there was some instructions before it, $i, i-1, i-2$ there are some instructions before it.

You see this halt instruction will be decoded during the ID phase. So, it is here after instruction fetch is done we will be knowing that this is actually a halt instruction, but we cannot stop the processor right here. Why, because the previous instruction which was here before halt that has still not completed, it was still here in the EX stage only and the instruction before that $i-1$ that was in the MEM stage, and the one before that was in the WB stage. So, you cannot stop the processor as soon as you detect it is halt, but rather you wait till the write back stage is reached and only then you halt the processor.

But you remember that the halt instruction was found. So, you maintain a flip-flop kind of a thing this we referred to as halted, say a single bit variable, you set this flip-flop to 1 whenever you find there is a halt instruction. So, what this will do you see just after the halt instruction again some more instructions will automatically be entering the pipe, but the machine is supposed to stop here, these should not be executed. So, while you execute a instruction, you check whether this flag is set or not. So, instructions $i+2$ and $i+3$ will be finding and will be detecting that this flag is indeed set to 1. So, it will not be doing any write, like it. If it was a store instruction, it will not be writing into memory and if it is a register instruction, it will not be writing into register. So, writing will be disabled if this halted state is there.

Now, similar will be the case if let us assume a scenario, there is a branch instruction, let say branch equal to zero something. And there are some instructions after that also. Let us assume again that this $i+1$ this was my branch instruction BEQZ. So, it is during ID you detect that it is a branch and if you look at the micro operations you will be identifying only at the end of EX whether the branch is actually to be taken or not because the target address is also computed and also the condition. Let say I have specified here R5, it is jumping to some label loop. So, it will check whether R5 is zero or not that is also done in the EX stage. So, whatever instruction has been fetched in the next cycle, if the branch is actually taken then this will have to be discarded, right.

So, because of the same reason we maintain another flip-flop this we call as taken_branch. So, it means whenever you detect in the EX stage that the branch is actually to be taken that the condition is true, you will actually have to go to loop. So, you set this variable to 1 again. And all the following instructions which have already entered, it will be checking this variable. If this variable is equal to 1, it will similarly not write anything in the registers or memory. So, this is as good as saying that all the instructions which are following the branch, they will not be executed. So, this is one change that you are making with respect to the, you can see the micro operation that we have seen means earlier during our last lecture.

(Refer Slide Time: 07:39)

Verilog Implementation of MIPS32 Pipeline

- Here we show the behavioral Verilog code to implement the pipeline.
- Two special 1-bit variables are used:
 - **HALTED** :: Set after a HLT instruction executes and reaches the WB stage.
 - **TAKEN_BRANCH** :: Set after the decision to take a branch is known. Required to disable the instructions that have already entered the pipeline from making any state changes.

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

So, these are the two points I now just mentioned. So, we are using two 1-bit variables, halted and taken_branch. The reason I have already explained, right, fine.

(Refer Slide Time: 07:54)

```
module pipe_MIPS32 (clk1, clk2);

    input clk1, clk2;           // Two-phase clock

    reg [31:0] PC, IF_ID_IR, IF_ID_NPC;
    reg [31:0] ID_EX_IR, ID_EX_NPC, ID_EX_A, ID_EX_B, ID_EX_Imm;
    reg [2:0]  ID_EX_type, EX_MEM_type, MEM_WB_type;
    reg [31:0] EX_MEM_IR, EX_MEM_ALUOut, EX_MEM_B;
    reg      EX_MEM_cond;
    reg [31:0] MEM_WB_IR, MEM_WB_ALUOut, MEM_WB_LMD;

    reg [31:0] Reg [0:31];      // Register bank (32 x 32)
    reg [31:0] Mem [0:1023];   // 1024 x 32 memory

    parameter ADD=6'b000000, SUB=6'b000001, AND=6'b000010, OR=6'b000011,
              SLT=6'b000100, MUL=6'b000101, HALT=6'b111111, LW=6'6001000,
              SW=6'b001001, ADDI=6'b001010, SUBI=6'b001011, SLTI=6'b001100,
              BNEQZ=6'b001101, BEQZ=6'b001110;
```

Now, let us straight away come to the Verilog code. This is the header of the description of the pipeline, where you declare all the registers. So, you see here in this module the inputs are only the clock nothing else, because instructions are coming from memory, there have been loaded in memory that is all; there is no separate input that is coming to the processor only clk1 and clk2, let say these are inputs. So, just like with the last example of pipelining we discussed a few lectures back, we shall be using a two phase clock.

And stage wise we are identifying all the variables that are required, there will be part of the inter stage latch and we are using the same variable naming convention that we have mentioned last lecture. PC - program counted will be a 32-bit register and in addition for the IF_ID latch we need IR and next PC, NPC. For ID_EX you need IR, NPC, A, B and Imm, these are all 32-bit quantities. Now, in addition we also maintain some variable called type. Well, type let me just go ahead in next slide and show you what is type.

(Refer Slide Time: 09:27)



The screenshot shows a Verilog code editor with a yellow background. The code defines parameters and registers:

```
parameter RR_ALU=3'b000, RM_ALU=3'b001, LOAD=3'b010, STORE=3'b011,  
        BRANCH=3'b100, HALT=3'b101;  
  
reg HALTED;  
    // Set after HLT instruction is completed (in WB stage)  
  
reg TAKEN_BRANCH;  
    // Required to disable instructions after branch
```

The bottom of the slide features the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the title "Hardware Modeling Using Verilog". There is also a circular portrait of a man.

Type is actually the type of the instruction that is encoded in 3-bits, we are identifying a few classes of instructions like register-register ALU we call it type 000 be type zero; register to memory ALU 001; load – 010; store – 011; branch - 100 and halt - 101. So, these are the codes that you are defining and using parameters we can use this names instead of the numbers, right. So, after the instructions decoded in the ID stage you recall during the ID stage instruction is already being fetched, it is stored in IF_ID_IR the instruction register from there the first 6-bits, you recall the instruction encoding if first 6-bits are the opcodes. So, the opcode will be decoded and there you can find out whether it is an add instruction, add immediate, load, store, jump whatever.

So, depending on that you set this variable type. And as I said type is a 3-bit variable, so it is a 3-bit register, and this type we forwarded across stages. So, there will be one ID_EX type that will be forwarded to EX_MEM type, MEM_WB type and so on. Because in each stage, we will be taking some decision depending on the type of the instruction that is why we are keeping track of this information

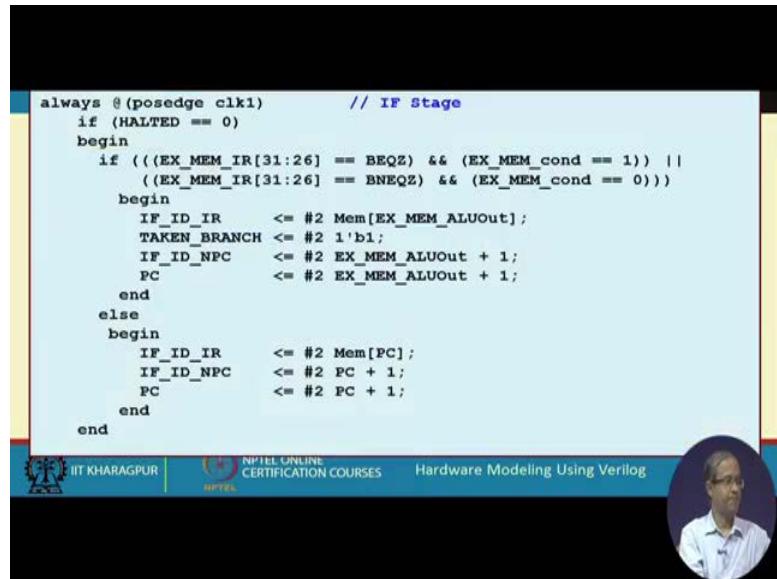
Then in the EX_MEM stage we have registers like IR, ALUOut, B and a single bit variable condition cond. This is required for jump or branch condition checking. And MEM_WB we have IR, ALUOut and for load instruction load memory data LMD. Here we declare the registers, they are, there are 32 registers 0 to 31 each of 32-bits. So, this is

a register bank 32 by 32. And this is how we declare the memory, we are assuming there is 1024 words, each word of 32. So, the memory is 1024 by 32

Well, here in this parameter declaration, we are defining the opcodes which we have already discussed in an earlier lecture, we had assigned some 6-bit opcodes to all the instructions, and this parameter we actually summarising that. So, instead of referring the opcodes by their bit patterns, we will be using just add, sub then add immediate and so on because it will make our Verilog code much easier to understand. We just using parameter this example illustrates, we can make our code more readable, because you can immediately know what that number actually stands for. So, here all these opcodes they are assigned a mnemonic ADD, SUB and so on. So, we have assigned opcodes to all the instructions we are considering, right.

Now, this we have already showed. These are the types and these are the two single bit flip-flops I mentioned. HALTED, this will be set after the halt instruction is completed in the write back stage after the halt instruction reaches WB. And TAKEN_BRANCH is another single bit register which will be set. After you decide that you are taking a branch and that is known only at the end of the EX stage as I have mentioned, only at the end of EX stage. You will be able to set this variable, so that all the following instructions which have entered the pipe they can check that value of the variable. And they will know that well the branch instruction is actually being taken, so we must not make any changes anywhere, so all writes will be disabled.

(Refer Slide Time: 13:53)

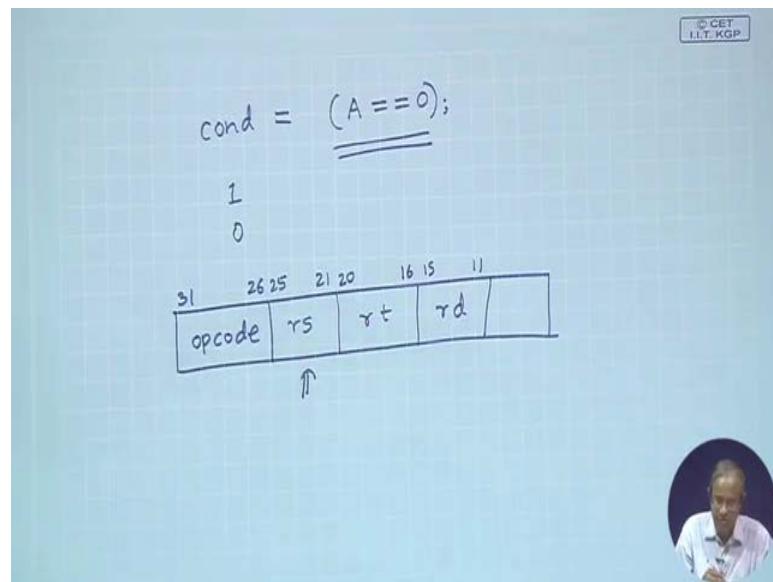


```
always @ (posedge clk1)          // IF Stage
begin
    if (HALTED == 0)
        begin
            if (((EX_MEM_IR[31:26] == BEQZ) && (EX_MEM_cond == 1)) ||
                ((EX_MEM_IR[31:26] == BNEQZ) && (EX_MEM_cond == 0)))
                begin
                    IF_ID_IR      <= #2 Mem[EX_MEM_ALUOut];
                    TAKEN_BRANCH <= #2 1'b1;
                    IF_ID_NPC    <= #2 EX_MEM_ALUOut + 1;
                    PC           <= #2 EX_MEM_ALUOut + 1;
                end
            else
                begin
                    IF_ID_IR      <= #2 Mem[PC];
                    IF_ID_NPC    <= #2 PC + 1;
                    PC           <= #2 PC + 1;
                end
        end
    end
```

So, now let us look at the different stages of the pipeline. This is the instruction fetch stage. Well, in instruction fetch stage, it is triggered by positive edge of clk1. Well, we do this only if halted is not set, if a halt instruction is already set the halted flag that means, there is no need of fetching any further instruction. So, we execute this block only when halted this flag is equal to 0, this flip-flop.

Now, what do you check here? First in this if statement we are checking whether there is a branch which is being taken. How we are checking it, you see we are first checking whether the opcode, you recall in the instruction register the opcodes were bit number 26 to 31. So, we are checking whether the opcode is equal to BEQZ. Well, in the parameter, we have already defined a bit pattern for BEQZ. So, it is BEQZ equal to 0 and the condition flag equal to 1.

(Refer Slide Time: 15:08)

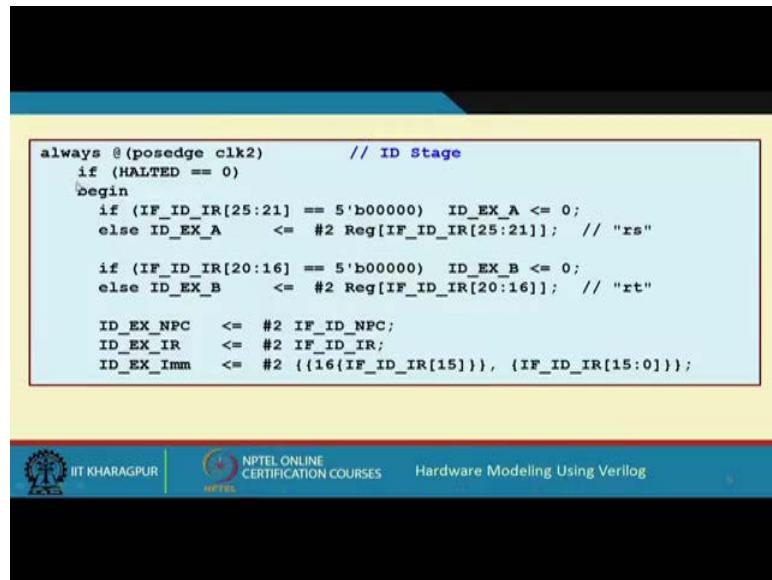


So, what this condition flag will represent, this condition flag will actually represent, this condition, this register A you recall the output of the first register that you are fetching, this equal to equal to 0, this is the expression you are assigning to cond, cond is a single bit variable. So, if this is true, cond will get 1; if this is false, cond will get 0. So, if the register value that is fetched in A is 0, then cond will be 1. So, this condition says that if it is a branch equal to zero instruction and it is actually equal to 0 that means, cond is 1. So, you have to branch or it was a branch not equal to zero instruction and cond was equal to 0 that means, it was the register was not equal to 0 that means, cond is 0. If either of these is true which means we have to jump.

In that case, what we do, so this instruction register we do not fetch from PC, but rather we fetch it from the address we just calculated in EX_MEM. You see for the branch instruction as it said at the end of the EX stage the address of the branch is already calculated and stored in EX_MEM_ALUOut. So, you take the address from there that memory location you load in IR. And you set the taken branch flip-flop to 1, these variable to 1 indicating that we are actually taking the branch; and this is the address from where you are fetching the instruction. So, now my PC or the NPC which is pointing to the next instruction must be this plus one, just the next address and else if it is not a branch, branch is not taken then it is the normal sequence of execution, you fetch the instruction from PC, then increase PC and NPC both by 1. This is the instruction fetch stage.

Then lets come to instruction decode. So, in the instruction decode you recall, we basically do three things. First one we actually decode the instruction well which we are not showing in the Verilog code because there is implied in the case statements. Secondly, we are pre-fetching to source registers and thirdly we are sign extending the 16-bit offset, let us see how we are doing that.

(Refer Slide Time: 18:00)



```

always @(posedge clk2)          // ID Stage
begin
    if (HALTED == 0)
        begin
            if (IF_ID_IR[25:21] == 5'b00000) ID_EX_A <= 0;
            else ID_EX_A      <= #2 Reg[IF_ID_IR[25:21]]; // "rs"

            if (IF_ID_IR[20:16] == 5'b00000) ID_EX_B <= 0;
            else ID_EX_B      <= #2 Reg[IF_ID_IR[20:16]]; // "rt"

            ID_EX_NPC     <= #2 IF_ID_NPC;
            ID_EX_IR      <= #2 IF_ID_IR;
            ID_EX_Imm     <= #2 ({16{IF_ID_IR[15]}}, {IF_ID_IR[15:0]});
        end
    end
end

```

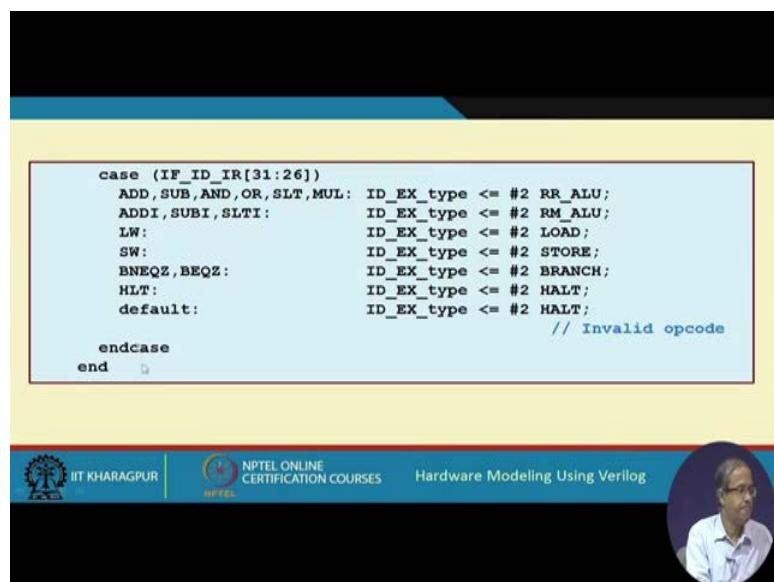
The screenshot shows a portion of a Verilog code editor. The code is contained within a red-bordered box. It defines an always block that triggers on the posedge of clk2. Inside, it checks the HALTED flag. If HALTED is 0, it performs several assignments based on the values in the IF_ID_IR register. Specifically, it checks bits 25:21 and 20:16. If either is 5'b00000, it assigns ID_EX_A or ID_EX_B to 0. Otherwise, it uses a #2 delay to read from a register indexed by the corresponding bits of IF_ID_IR. It also initializes ID_EX_NPC, ID_EX_IR, and ID_EX_Imm. At the bottom of the slide, there are logos for IIT Kharagpur and NPTEL, along with the text 'NPTEL ONLINE CERTIFICATION COURSES' and 'Hardware Modeling Using Verilog'.

So, here again we start by checking the halted flag. If HALTED is zero only then you do it; if HALTED is set you skip. Well, here when you are fetching the registers, you make a special check, you recall I mentioned that R0 is a register, special register, which is assumed to always contain the value zero, nothing else can be written to R0. So, here we are checking, you see bit number is 21 to 25 in the instruction register, this represents the first register operand. Just you recall again in the instruction register, the first 6-bits is the opcode. So, it is bit number 31 up to 26. Then you have register source 25, 21 then you have rt, rt is 16 to 20; then you have rd, rd is 11. So, here we are accessing rs, bit number is 21 to 25.

So, we check whether this is 0 or not, zero means we are trying to access R0. So, if it is R0 then we do not access register bank you straight away assign zero to this A variable otherwise A is assign to access the register with this register number. So, whatever 5-bit register number is there that is rs that will be loaded into A, ID_EX_A.

Then you load the second operand, which is in bit number 16 to 20, same thing. If it is 0 you load the value 0 to B otherwise you read from the register this is your rt field, load it into B. Then the other routine things this NPC, you simply forwarded because you need it later IF_ID_NPC will go to ID_EX_NPC; IR also you forward to the next latch stage and immediate here you are doing the extension, sign extension. So, in the IR, bit number 0 to 15 indicates the offset. So, this will be the last 16-bits and the highest 16-bits, you take bit number 15 replicated 16 times I said, the sign bit will be replicated 16 times and then concatenate, this will become a 32-bit quantity. So, immediate is created by sign extension.

(Refer Slide Time: 20:59)



```

case (IF_ID_IR[31:26])
    ADD,SUB,AND,OR,SLT,MUL: ID_EX_type <= #2 RR_ALU;
    ADDI, SUBI, SLTI:        ID_EX_type <= #2 RM_ALU;
    LW:                      ID_EX_type <= #2 LOAD;
    SW:                      ID_EX_type <= #2 STORE;
    BNEQZ,BEQZ:              ID_EX_type <= #2 BRANCH;
    HALT:                    ID_EX_type <= #2 HALT;
    default:                  ID_EX_type <= #2 HALT;
                                // Invalid opcode
endcase
end

```

The slide also features the IIT Kharagpur logo, NPTEL Online Certification Courses logo, and a portrait of a man.

Then you also do another thing. You check the opcode again, this is the opcode bit number 26 to 31. Depending on the opcode type, you set the value of this type. This type you recall we have already defined 3-bit codes to the different types and using parameter instead of writing 000001, we can write it in a much more readable way like RR_ALU, RM_ALU, load, store etc. And these are the instruction, these opcodes also have been declared using parameter. So, you understand now what is the advantage of using parameters, our code becomes much more readable. So, now you can actually see by looking at the code that well yes we are looking at the ADD instruction otherwise seeing the opcode look at the table that well this opcode means ADD, we have to refer every time but this makes your code more readable, right, fine.

So, if it is either ADD, SUB, AND, or SLT or MUL, you call it as a register-to-register ALU type. If it is ADDI, SUBI, SLTI call it as RM type. If it is LW, SW, BNEQZ or BEQZ, it is branch, HLT is halt. And by default if it is something else by chance then also you make it halt, there is an invalid opcode case. Just one thing this ID stage is triggered by clk2.

(Refer Slide Time: 22:42)

```

always @ (posedge clk1)          // EX Stage
begin
    if (HALTED == 0)
        begin
            EX_MEM_type  <= #2 ID_EX_type;
            EX_MEM_IR   <= #2 ID_EX_IR;
            TAKEN_BRANCH <= #2 0;

            case (ID_EX_type)
                RR_ALU: begin
                    case (ID_EX_IR[31:26]) // "opcode"
                        ADD:   EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_B;
                        SUB:   EX_MEM_ALUOut <= #2 ID_EX_A - ID_EX_B;
                        AND:   EX_MEM_ALUOut <= #2 ID_EX_A & ID_EX_B;
                        OR:    EX_MEM_ALUOut <= #2 ID_EX_A | ID_EX_B;
                        SLT:   EX_MEM_ALUOut <= #2 ID_EX_A < ID_EX_B;
                        MUL:   EX_MEM_ALUOut <= #2 ID_EX_A * ID_EX_B;
                        default: EX_MEM_ALUOut <= #2 32'hxxxxxxxxx;
                    endcase
                end
            end
        end
    end
end

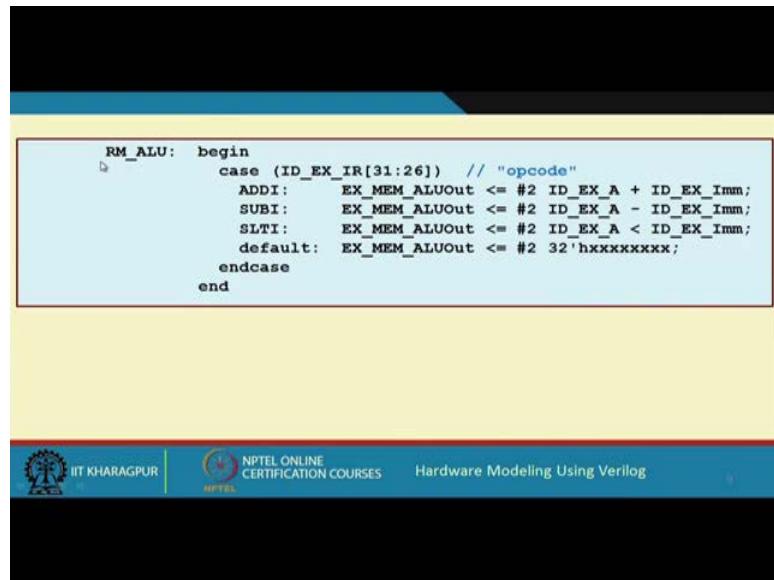
```

Then next comes the EX stage which is again triggered by clk1 and the way we started is the same, we check the halted flag. So, if it is equal to zero only then we do it otherwise you skip it. First we forward a few things, well this EX type, whatever type you have calculated earlier in the last stage will be forwarded to the next stage if IR is also forwarded. And taken branch you see, taken branch this variable was set in the IF stage, right if you see earlier. So, in the IF we had set this variable here. Because of this the next instruction that we will see later that we will not be allowed to write into the WB stage, but all instruction after that again should be allowed to continue because there will be starting to fetch from the correct address because already program counter has been updated.

So, now we are resetting the taken branch back to zero here. Now, here depending on the instruction type we are carrying out the operations. There is a case on type ID_EX type. So, if it is register-register ALU then again we use a case statement on the opcode, this is opcode. So, it is ADD, SUB, AND, OR, SLT or MUL, you specify what you are doing.

If it is ADD, ID_EX_A and ID_EX_B are added, result is stored in ALUOut. If it SUB, subtract; AND; OR; if it is set if less than you just do a comparison less than, if it is true then 1 will be stored in ALUOut, if it is false 0 will be stored in ALUOut. And MUL is multiplication. And if it is an invalid opcode, then you just load this undefined xxx in ALUOut.

(Refer Slide Time: 24:50)



The screenshot shows a Verilog code editor with a yellow background. A red box highlights a portion of the code. The code defines a process for RM_ALU based on the value of ID_EX_IR[31:26]. It includes cases for ADDI, SUBI, and SLTI, and a default case. The ADDI case adds ID_EX_A and ID_EX_Imm. The SUBI case subtracts ID_EX_Imm from ID_EX_A. The SLTI case compares ID_EX_A and ID_EX_Imm using a less-than operator. The default case loads an undefined value (32'hxxxxxxxxx). The code ends with an endcase and an end keyword.

```
RM_ALU: begin
    case (ID_EX_IR[31:26]) // "opcode"
        ADDI: EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_Imm;
        SUBI: EX_MEM_ALUOut <= #2 ID_EX_A - ID_EX_Imm;
        SLTI: EX_MEM_ALUOut <= #2 ID_EX_A < ID_EX_Imm;
        default: EX_MEM_ALUOut <= #2 32'hxxxxxxxxx;
    endcase
end
```

Next if it is registered to memory RM_ALU, this type then again you look at the opcode, these are the three kinds of RM_ALU instructionsm add immediate. Here will be adding a with the immediate data. Subtract will be subtracting immediate data from A. SLTI will be compared in less than A and Imm, rest is same.

(Refer Slide Time: 25:17)

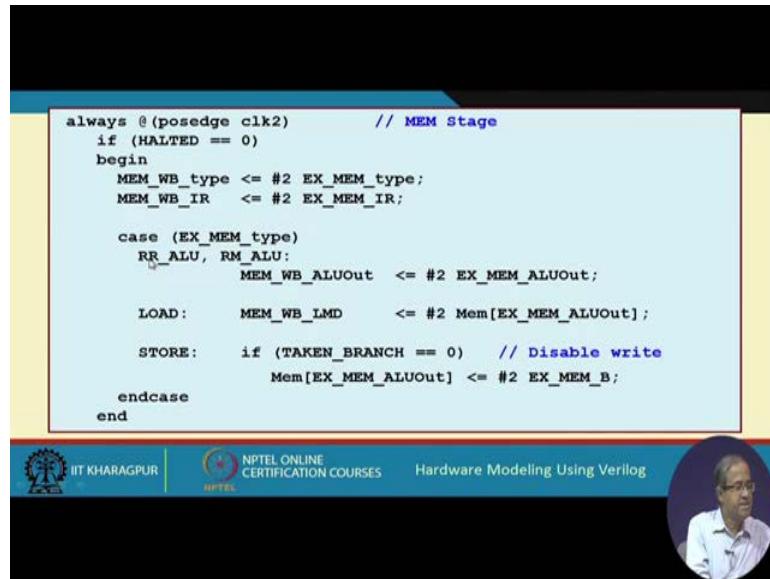
```
LOAD, STORE:
begin
    EX_MEM_ALUOut <= #2 ID_EX_A + ID_EX_Imm;
    EX_MEM_B      <= #2 ID_EX_B;
end

BRANCH: begin
    EX_MEM_ALUOut <= #2 ID_EX_NPC + ID_EX_Imm;
    EX_MEM_cond   <= #2 (ID_EX_A == 0);
end
endcase
end
```

Then if it is load or store, then you are calculating the address of the memory, address of the operand. What you do you simply add the value of A and Imm. So, this will give you the address of the memory, this is stored in ALUOout. And the values of B is forwarded because this will be requiring for store instructions later. And if it is branch then you see this is where you take the decision branch, you calculate the target address of the branch. How you do it? the value of the program counter NPC is added to the offset Imm. So, here you are calculating the target address if you have to take a branch and also you are evaluating this variable cond, right.

Now, you see this ALUOut and cond, which you are calculating in the EX stage, let us again go back to the IF stage. You are actually using them in IF stage, you see, you are looking at this cond and this branch address, these are being used here. So, now you can understand from EX_MEM how they are coming? they are actually getting calculated here, fine. Let us move on to the MEM stage, this is again triggered by clk2. So, again if this halted is zero only then you execute this. So, what you do type is forwarded, IR is also forwarded. And here again there is a case statement, well let us make it uniform, here I did not put this #2, I given a delay just for notational purpose. Let us use the same delay here #2, fine.

(Refer Slide Time: 27:14)



```
always @ (posedge clk2)      // MEM Stage
begin
    MEM_WB_type <= #2 EX_MEM_type;
    MEM_WB_IR  <= #2 EX_MEM_IR;

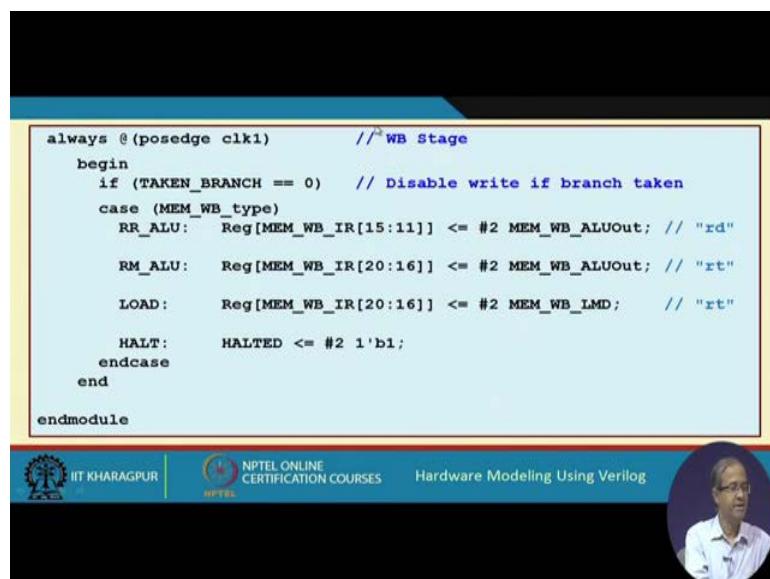
    case (EX_MEM_type)
        RR_ALU, RM_ALU:
            MEM_WB_ALUOut <= #2 EX_MEM_ALUOut;

        LOAD:   MEM_WB_LMD    <= #2 Mem[EX_MEM_ALUOut];
        STORE:  if (TAKEN_BRANCH == 0) // Disable write
                  Mem[EX_MEM_ALUOut] <= #2 EX_MEM_B;
    endcase
end
```

The screenshot shows a portion of a Verilog code editor. The code is for the Memory Write Back (WB) stage. It includes logic for handling different EX_MEM types (RR_ALU, RM_ALU), performing loads (LOAD), and stores (STORE). For stores, it checks if the taken branch is 0 or 1 to either disable the write or update the memory location. The code uses the EX_MEM_ALUOut signal from the previous stage and updates the EX_MEM_B signal.

So, if it is a register to register RR to RM, you do nothing just to forward ALUOut to the next latch. If it is load, the address of memory is already in ALUOut, we have calculated in the last type EX, we access memory and store it in LMD. And store, well in store you check whether taken branch is 0 or 1; if it is 1, you do not do a write, disable write; if taken branch has been 0 only then you do the write, whatever is there in B that you store in the memory location.

(Refer Slide Time: 27:54)



```
always @ (posedge clk1)      // WB Stage
begin
    if (TAKEN_BRANCH == 0) // Disable write if branch taken
    case (MEM_WB_type)
        RR_ALU: Reg[MEM_WB_IR[15:11]] <= #2 MEM_WB_ALUOut; // "rd"
        RM_ALU: Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_ALUOut; // "rt"
        LOAD:  Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_LMD; // "rt"
        HALT:  HALTED <= #2 1'b1;
    endcase
end
endmodule
```

The screenshot shows a portion of a Verilog code editor for the Write Back (WB) stage. The code uses the EX_MEM_ALUOut signal from the previous stage to update memory locations for reads ("rd") and writes ("rt"). It also updates the HALTED signal. The code is enclosed in an endmodule block.

And lastly in the right back or WB stage here trigger by positive edge of clk1 again. Here we check if taken branch is 0 only then you do it. If taken branch is 1 means someone is taking the branch, you disable the writes here, you should not write into the register, this registers, this instruction are actually dummy instruction, they should be ignored. So, what we do if it is RR_ALU, you recall there the destination register is store in bit numbers 11 to 15 in the instruction register. So, ALUOut will be stored here, that is rd. If it is rn, it is stored in rt bit numbers 16 to 20. And if it is load again the data is in LMD, it will be stored in rt again 16 to 20. And if it is the halt instruction it is here in the WB stage, you are setting this flag halted to 1, so that after you have halted, you have set the flag to 1, all further instructions will not do anything else it will the machine will basically stop.

So, with this we come to the end of this lecture, where basically we have translated the micro operation that we discussed in our last lecture to Verilog. Of course we have done a few things that are very specific to pipeline implementation like those two additional flags we have maintained and we have check those flags at the beginning of every stage whether that flag is active or deactive, if necessary you skip some stage. So, in the next and final lecture, what we shall be discussing, we shall be looking at some example programs that you want to run on this processor we have designed, and let us see how it works, this we shall be seeing in our next lecture.

Thank you.

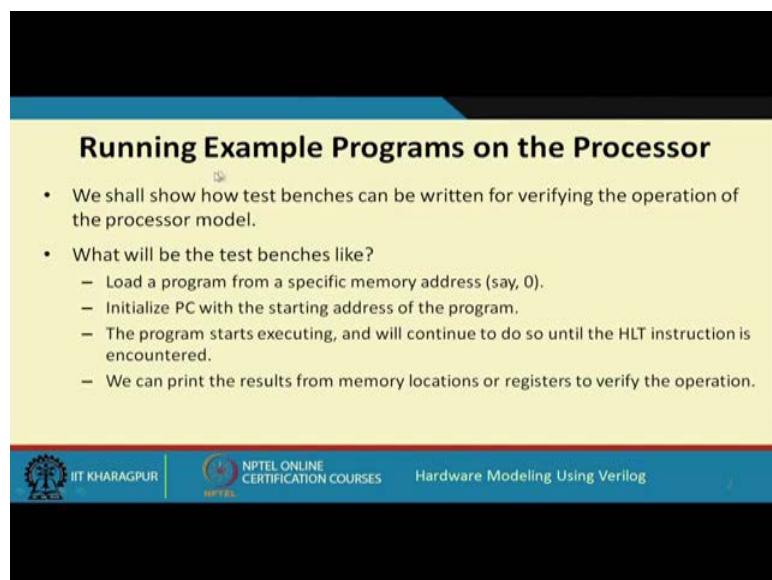
Hardware Modeling using Verilog
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 41
Verilog Modeling of the Processor (Part 2)

So, in the Verilog implementation of the processor that we had discussed during our last lecture, you recall we had considered only a small subset of instructions from the MIPS32 instructions set.

Now, these subsets of instructions have been chosen quite thoughtfully in the sense that there are many programs, which you can possibly write even using this small instruction set. Now in this lecture I shall be showing you three example test benches, which actually search the purpose of executing small programs in the MIPS32 assembly language. Let us go through them and see how it works.

(Refer Slide Time: 01:20)



Running Example Programs on the Processor

- We shall show how test benches can be written for verifying the operation of the processor model.
- What will be the test benches like?
 - Load a program from a specific memory address (say, 0).
 - Initialize PC with the starting address of the program.
 - The program starts executing, and will continue to do so until the HLT instruction is encountered.
 - We can print the results from memory locations or registers to verify the operation.

IT KHARAGPUR | **NPTEL ONLINE CERTIFICATION COURSES** | **Hardware Modeling Using Verilog**

So, the title of a lecturer is Verilog modeling of the processor the second part. Now here we shall actual be demonstrating by showing you how we can run some programs on the processor. Program means programs written in the MIPS32 assembly language, we are not talking a Verilog here. We have already designed a processor a CPU, now our next task is to write a program that will be running on that CPU, right.

Now, the test benches that will be writing, there will be having sudden kind of structure. The structure will be as follows the example programs that we shall show will be starting from a specific memory location. Well, we shall be assuming it is starting from address 0. And before we start execution we will be initializing PC with 0; that means, the first instruction that will be fetched will be from address 0, which is the first instruction.

And after we have started it then clock1 and clock2 are automatically coming. So, instructions will be fetched from memory one after the other automatically, you do not have to do anything else in the test bench. Only the initial thing we have to load the instructions in the memory, set program counter to 0 and that is it, right. And at the end we can print the result wherever the results are available. They may be in some memory locations or they may be in some registers, we shall see.

(Refer Slide Time: 03:01)

The screenshot shows a presentation slide with a yellow header bar. The title 'Example 1' is centered in the header. Below the title, there is a bulleted list of steps:

- Add three numbers 10, 20 and 30 stored in processor registers.
- The steps:
 - Initialize register R1 with 10.
 - Initialize register R2 with 20.
 - Initialize register R3 with 30.
 - Add the three numbers and store the sum in R4.

At the bottom of the slide, there is a footer bar with the following logos and text:
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog

Now, let us look at three examples one after the other. The first example is a very simple one, this says that we add 3 numbers 10, 20 and 30 in decimal, they are all stored in processor registers. So, how you write the program? That we initialize register R1 with 10, register R2 with 20, R3 with 30 then add them up and the result is stored in R3 or R5, in R4 or R5. In fact, we are storing in R5 not R4, first we are storing in R4, and then we are storing it in R5. Yeah, first 2 numbers we are adding storing in R4 the third number also adding, we are storing in R5.

(Refer Slide Time: 03:58)

The screenshot shows a table comparing Assembly Language Program and Machine Code (in Binary). The table has two columns: 'Assembly Language Program' and 'Machine Code (in Binary)'. The rows list the following instructions:

Assembly Language Program	Machine Code (in Binary)
ADDI R1,R0,10	001010 00000 00001 0000000000001010
ADDI R2,R0,20	001010 00000 00010 00000000000010100
ADDI R3,R0,25	001010 00000 00011 00000000000011001
ADD R4,R1,R2	000000 00001 00010 00100 00000 000000
ADD R5,R4,R3	000000 00100 00011 00101 00000 000000
HLT	111111 00000 00000 00000 00000 000000

At the bottom of the slide, there are logos for IIT Kharagpur and NPTEL, and the text 'NPTEL ONLINE CERTIFICATION COURSES' and 'Hardware Modeling Using Verilog'.

Let us first look at the program, how the program looks like. The program will look like this, you forget this for the time being, look at this program. What does the first instruction do? This add immediate R0 and 10 result goes to R1, which is as good as saying that we are initializing R1 with the value 10. Second instruction same thing R0 is always 00 and 20 are added in R2, R 2 equal to 20, similarly here R3 equal to 25.

Then in the 4th instruction we are adding R1 and R which are having values 10 and 20 and storing it in R4. So, R4 will be getting the value 30, then in the last instruction this 30 is added to R3, R3 is 25. So, the result is 55, 55 should go to R5 and then a halt.

Now, side by side we are showing the machine code because ultimately what we have to load in memory are the machine code of the instructions, right. Because the processor will be fetching the instruction those are the machine codes, ok.

Let us see how you have done that. Just look at the first instruction. This add immediate you consult the opcode table, the opcode for ADDI is 001010, this 6 bit is the opcode, I have shown some gaps to indicate clearly. Then the 2 source registers. Sources here there is only one source register R0, sources R0 it is all 0, indicating R0, then the destination. Destinations is R1, 00001 it is 1. Then 16-bit the immediate data, here it is 10. So, in binary 10 is 1010, you see this is 10 in 16-bits.

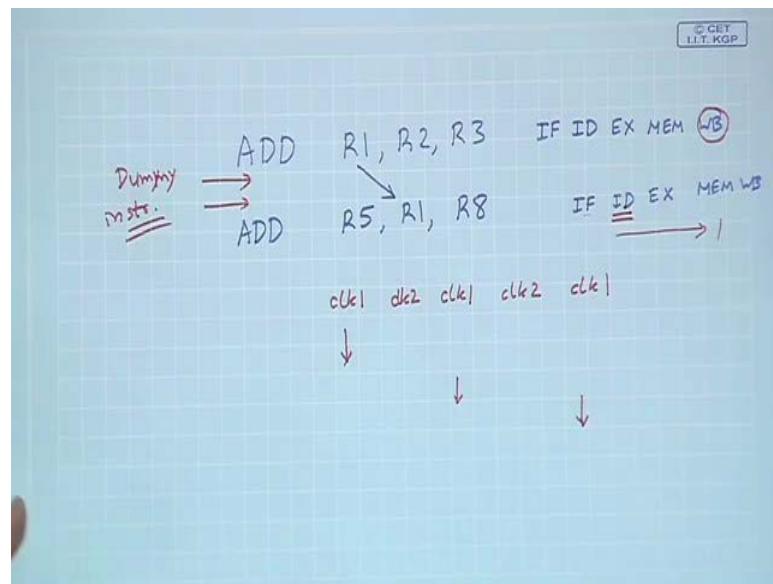
Second instruction similarly this is add immediate (ADDI), this is R0. This is 2, 10 which is R2 and this is 20 you see 10100 in binary mean 16 and 4, 20. Similarly this one is R3 and this one is 25, then there are register-to-register, I mean instructions two of them at R1 and R2 are added to R4. So, this is the opcode for ADD all 0s, this is R1, 1; R2, 2; R4, 1, 00, 4. And as I said for these instruction the last bits will be they are for some other purposes will be keeping them 0s. Similarly, here R3 and R4, this is R4 and R3, this is 4, this is 3, the target is 5. And HALT this is the opcode all one and the remaining bits are not used, set it all to 0.

So, actually when you will be loading them in memory you can actually group them 4 bits at a time and find the hexadecimal code, like the first one it would be how much 0010 is 2; 1000 is 8; 0000 is 0 and 0001 is 1. So, 2, 8, 0, 1 and here it will be 000a. In this way you can calculate the opcode of all the instructions in hexadecimal.

So, when you actually load the instructions in memory we shall be showing the hexadecimal codes, that you can directly get from this table or you can use the instruction format to create the instruction encoding. And find out what the opcode is coming for the 32-bit instruction is coming. So, you can convert it to hexadecimal after that, ok.

So, after we have done this, there is one point that I would like to just mention here. That I mentioned when we are discussed pipelining that instruction pipelining that there is something called hazard, because of data dependencies you can have something called data hazard.

(Refer Slide Time: 08:28)



Like suppose let us say, let us take an example suppose I have an instruction ADD R1, R2, R3. The next instruction is ADD, let us say R5, R1, R8.

So, the first instruction is producing the result in R1, which is used by the second instruction. So, if you look at the instruction execution cycle. Instruction fetch (IF), instruction decode (ID), execution (EX), MEM and WB. You see this R1 is getting stored only during the WB stage. And in the next instruction is trying to read the registers in the ID phase, so obviously it will be reading a wrong value

So, it has to wait at least for 2 cycles before it can read, right, this is one thing. So, if you really have a pair of instructions like this and the hardware is not automatically taking care of this, then you may have to insert some dummy instructions in between. So, what the dummy instructions will do? It will simply serve to delay this instruction, this will be delayed by 2 steps or 1 step whatever so that you can. So, by the time this instruction reaches ID this write back should be complete.

Now, another thing you remember that we are not using a single clock, we are using clk1, clk2 alternately, right, in the stages. So, the first instruction is fetched here. Instruction fetches done only during clk1, the second instruction is fetched in clk1, third instruction is fetched in clk1. So, if you delay it by one instruction that will serve your purpose. So, just one dummy instruction if you include that should be ok.

Now, in this example we do not have a problem here because this ADDI is calculating R3, and this ADD is not using R3, but here I have a problem. You are computing R4, this R4 is being used here, right, ok.

(Refer Slide Time: 11:03)

The screenshot shows a computer screen with a Verilog test bench window. The window title is "TEST BENCH". The code inside the window is:

```
module test_mips32;
    reg clk1, clk2;
    integer k;

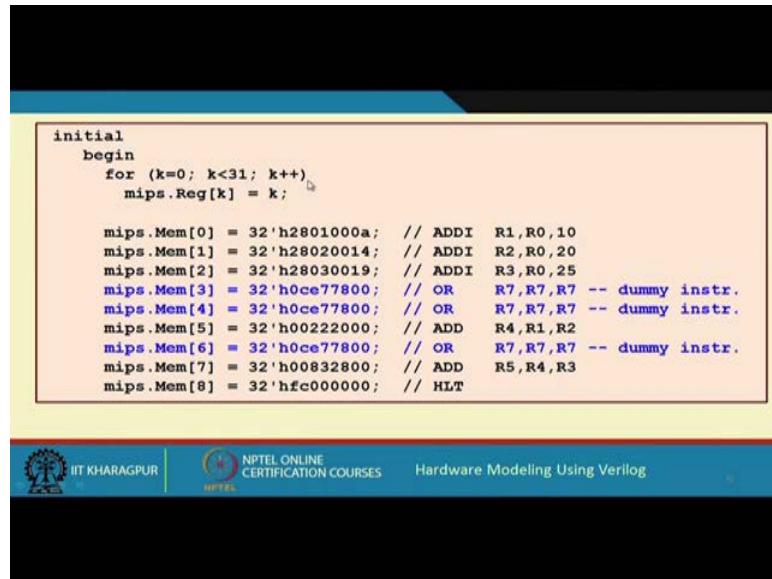
    pipe_MIPS32 mips (clk1, clk2);

    initial
        begin
            clk1 = 0; clk2 = 0;
            repeat (20)                      // Generating two-phase clock
                begin
                    #5 clk1 = 1; #5 clk1 = 0;
                    #5 clk2 = 1; #5 clk2 = 0;
                end
        end
end
```

At the bottom of the screen, there are logos for IIT Kharagpur and NPTEL, and the text "NPTEL ONLINE CERTIFICATION COURSES" and "Hardware Modeling Using Verilog".

So, this we shall see in the test bench how we have handled this. Let us now come to the test bench, this is the first part of the test bench where we have instantiated our processor we call it MIPS, only 2 parameters, declared as reg and variable k. Here we are generating the 2 phase clock, this we have also seen earlier how we can generate 2 phase clock, we have the same code. Well 20 clock cycle is sufficient for this program. So, we have repeated it for 20.

(Refer Slide Time: 11:39)



```
initial
begin
    for (k=0; k<31; k++)
        mips.Reg[k] = k;

    mips.Mem[0] = 32'h2801000a; // ADDI R1,R0,10
    mips.Mem[1] = 32'h28020014; // ADDI R2,R0,20
    mips.Mem[2] = 32'h28030019; // ADDI R3,R0,25
    mips.Mem[3] = 32'h0ce77800; // OR R7,R7,R7 -- dummy instr.
    mips.Mem[4] = 32'h0ce77800; // OR R7,R7,R7 -- dummy instr.
    mips.Mem[5] = 32'h00222000; // ADD R4,R1,R2
    mips.Mem[6] = 32'h0ce77800; // OR R7,R7,R7 -- dummy instr.
    mips.Mem[7] = 32'h00832800; // ADD R5,R4,R3
    mips.Mem[8] = 32'hfc000000; // HLT
```

The screenshot shows a software interface for hardware modeling using Verilog. At the top, there's a toolbar with various icons. Below the toolbar is a menu bar with options like File, Edit, View, Insert, Tools, Help, and a Language dropdown set to Verilog. The main area is a code editor containing the provided Verilog code. At the bottom of the screen, there's a footer bar with the IIT Kharagpur logo, the NPTEL Online Certification Courses logo, and the text "Hardware Modeling Using Verilog".

Then we load our program in memory, before that we initialize the registers, k0 to 31, reg k equal to k, which means reg 0 is 0; register 1 is 1; register 2 is 2; register 3 is 3 just some initialization.

Now, here you see we have inserted some dummy instructions. Now as I said these dummy instructions are not required, you need only this, but actually here we have this inserted 2 dummy instructions. Now what is the dummy instruction we have used? We used an instruction like OR R7, R7, R7. What does that mean? You do a logical OR with R7 and R7, we have initialize R7 with 7, right. So, you do OR 7 with 7 result will be 7 itself, store 7 back to R7. So, result will not change, but it will consume one clock cycle, ok.

So, like this wherever you feel there is a doubt there is a hazard data dependency you can insert some dummy instruction like this. So, here I just shown. So, here we have inserted 2 dummy instructions and 1 dummy instruction here. The other instructions are just as we had shown earlier. And mips.Mem[0] we are loading the hex code of the first instruction, Mem[1] address 1, second instruction 2.

(Refer Slide Time: 13:18)

The screenshot shows a Verilog simulation interface. On the left, the Verilog code is displayed:

```
mips.HALTED = 0;
mips.PC = 0;
mips.TAKEN_BRANCH = 0;

#280
for (k=0; k<6; k++)
    $display ("R%1d - %2d", k, mips.Reg[k]);
end

initial
begin
    $dumpfile ("mips.vcd");
    $dumpvars (0, test_mips32);
    #300 $finish;
end

endmodule
```

On the right, the "SIMULATION OUTPUT" window displays the register values:

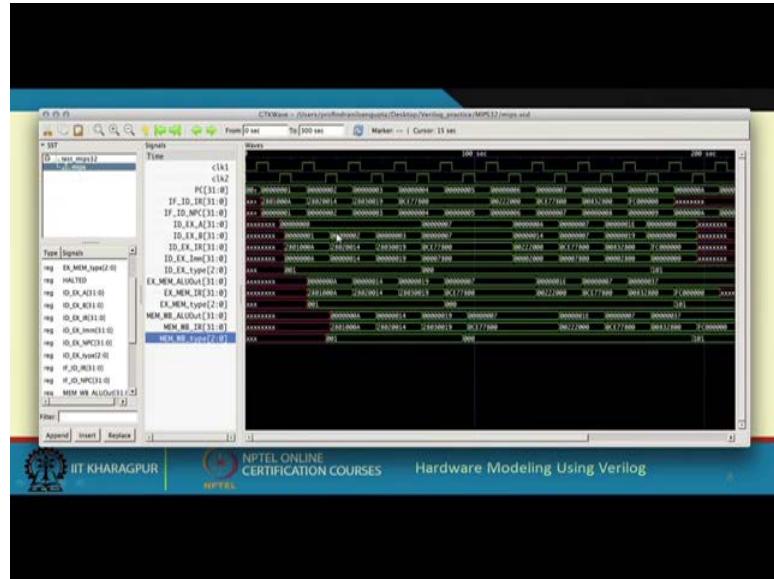
R0	-	0
R1	-	10
R2	-	20
R3	-	25
R4	-	30
R5	-	55

In this way there are 9 memory locations, right. And before we actually start execution, we reset the HALTED flip-flop to 0, TAKEN_BRANCH to 0 and set PC to 0. *****

So, after this when the clock starts, clock will start after a delay of 5 the first one. So, the instruction you start executing well. And after sufficient delay we are displaying the values of R1, not R1, k equal to 0 to 5, R0, R1, R2, R3, R4 and R5.

So, we are displaying R register number dash, the value in the register, let us see. So, if you run this, the simulation output will come like this, as part of this loop, for loop. So, R0 is 0, R1 is 10, 20, 25 which we have already loaded. R4, 10 and 20 was added, R4 is 30 and R5, 30 and 25 is added, it is 55. So, the results are correct.

(Refer Slide Time: 14:39)



Now, in a similar way you have also captured the wave form in a file MIPS.vcd. So, if you see the timing diagram, the fonts quite small, I do not know whether you are able to see it clearly or not, but you can see we have listed all the relevant signals, the clocks, PC, then IF_ID_IR, MPC, ID_EX_A, ID_EX_B, IR immediate and so on.

So, you can see that how things are going on, this is instruction type EX_MEMORY, it is 001 initially, we had the add immediate instruction, this is of RN type. You look at the ALUOut, first operation was add immediate R0 with 10, you see. ALUOut contains 10, after adding 0 and 10, 00, ALU is 10.

Next one was 20, which is 14 in hexadecimal means 20, 3rd one as 25, you see it was 19, 25. Then there is some dummy OR instructions, forget it 7. Then first two numbers were added 10 and 20, 30 you see, 1e is 30; then again there is a dummy OR instruction 7, then lastly, the last two are added 55 which is 37 hexadecimals. So, this way you can track all the other signals in clocks systematically they are going on, right.

So, at the end you see when, this, here halted signal is not shown, but at the end you will see after everything is done the halted signal will be activated and instruction you stop, you see here it had, it started to become undefined that mean stopped here. So, you see that with the small example that we are indeed able to write a program and run the program on our processor which were designed in Verilog, let us take another example.

(Refer Slide Time: 16:38)

Example 2

- Load a word stored in memory location 120, add 45 to it, and store the result in memory location 121.
- The steps:
 - Initialize register R1 with the memory address 120.
 - Load the contents of memory location 120 into register R2.
 - Add 45 to register R2.
 - Store the result in memory location 121.

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

Well in this example we are using a memory location, memory. So, what we are doing? We are loading a word which is already stored in memory location 120. Then we are adding 45 to it and storing the result back in memory location 121. So, the steps are very simple which are illustrating with the program, I am showing the program straight away.

(Refer Slide Time: 17:06)

Assembly Language Program	Machine Code (in Binary)
ADDI R1, R0, 120	001010 00000 00001 0000000001111000
LW R2, 0(R1)	001000 00001 00010 0000000000000000
ADDI R2, R2, 45	001010 00010 00010 0000000000101101
SW R2, 1(R1)	001001 00010 00001 0000000000000001
HLT	111111 00000 00000 00000 00000 00000

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

So, the program is like this. Because we are storing the number in memory address 120, we are initializing a register with that address 120. This ADDI R1, R0, 120 means we are

loading 120 in R1, right, 0 and 120 are added, store in R1. Then you are loading from that address 0(R1) means 0 plus R1, this R1 contains 120.

So, that memory will be fetched and the data will be stored in R2. Then we are adding that number with 45, result in R2 and we are storing R2 in 1(R1) which means R1 contains 120, 120 plus 1 that means 121. So, instruction encoding is very similar, this add immediate I have already showed.

let us see this LW, this LW opcode is this 001000. Here source is R1 you see, this is 1(R1); destination is R2 this is 2, and offset is 0, you see all 0. This add immediate, this is add immediate R2, R2, this is 2, this is 2, 45, this is 45; store, this is store, this R1, R2 and 1, this is R2 this is R1 and 1 this is 1, halt, it is just halt.

So, here you see that there are lot of data dependencies. This add immediate is generating R1 which is used in the next instruction. Here you use here you are generating R2 which is again used in next instruction. This add immediate also is generating a new value of R2 which is used in the next instruction. There are lot of data dependencies.

So, let us look at the test bench again. The first part of the test bench is identical, same we are instantiated it and the clock generation logic. Then this is the program.

(Refer Slide Time: 19:15)

The screenshot shows a Verilog testbench window. The code area contains the following Verilog code:

```
initial
begin
    for (k=0; k<31; k++)
        mips.Reg[k] = k;

    mips.Mem[0] = 32'h28010078; // ADDI R1,R0,120
    mips.Mem[1] = 32'h0c631800; // OR R3,R3,R3 -- dummy instr.
    mips.Mem[2] = 32'h20220000; // LW R2,0(R1)
    mips.Mem[3] = 32'h0c631800; // OR R3,R3,R3 -- dummy instr.
    mips.Mem[4] = 32'h2842002d; // ADDI R2,R2,45
    mips.Mem[5] = 32'h0c631800; // OR R3,R3,R3 -- dummy instr.
    mips.Mem[6] = 32'h24220001; // SW R2,1(R1)
    mips.Mem[7] = 32'hfc000000; // HLT

    mips.Mem[120] = 85;
```

The simulation results window below shows the state of registers R0 through R3 and memory locations 0 through 120 over time. The memory values correspond to the assembly code above, with the final value at R2 being 121 (85 + 45).

You see I told that there is a data dependency between all these pairs of instructions. So, we have inserted a dummy instruction in between every pair of them. So, here the

dummy instruction we have used the OR R3, R3, R3. So, this opcode you can actually verify. So, I will give it an exercise for you that OR R3, R3, R3 is actually this is your 0c631800 and this will be our total code. So, our useful code is 5 instructions and we are adding 3 dummy instructions, it becomes 8. So, it is loaded from memory location 0 up to memory location 7. And we said that some data is already loaded in memory location 120, let us load a value 85 there, 85 is already stored in 120, memory 120.

(Refer Slide Time: 20:14)

The screenshot shows a Verilog simulation environment. On the left, the Verilog code is displayed:

```

mips.PC = 0;
mips.HALTED = 0;
mips.TAKEN_BRANCH = 0;

#500 $display ("Mem[120]: %d \nMem[121]: %d",
               mips.Mem[120], mips.Mem[121]);
end

initial
begin
  $dumpfile ("mips.vcd");
  $dumpvars (0, test_mips32);
  #600 $finish;
end

endmodule

```

On the right, the simulation output is shown in a box labeled "SIMULATION OUTPUT". It displays the memory values:

```

Mem[120]: 85
Mem[121]: 130

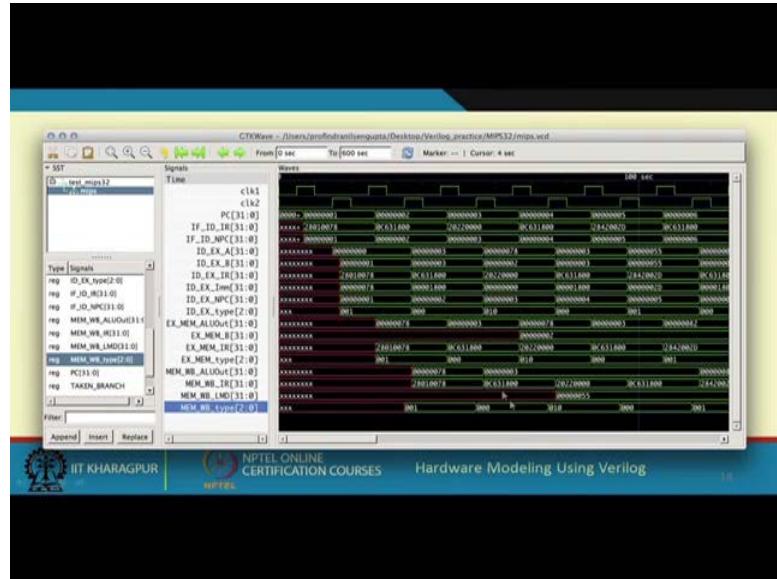
```

At the bottom of the interface, there are logos for IIT Kharagpur and NPTEL, along with the text "NPTEL ONLINE CERTIFICATION COURSES" and "Hardware Modeling Using Verilog".

Then we again initialize the PC to 0, halted to 0 and taken branch to 0, and at the end after sufficient delay we are displaying the value of memory location 120 and 121, these two things were pending, ok.

Let us see the output, this is how the output is coming MEM[120] : 85, MEM[121] : 130. So, actually 45 is being added to this, you see 85 plus 45 is actually 130, right.

(Refer Slide Time: 20:51)



So, here also the result is correct. So, again we showed the waveform here, the timing diagram.

So, here you can again analyze and say how it works. Just look at this LMD, for load instruction, this LMD is used. We see when the load instruction executed, this LMD becomes 55, 55 hexadecimal is 85. So, actually the data is getting loaded here. Then it will be added in that MIPS stores. So, so the whole thing you cannot see in the screen here I will be showing a part of it, right.

Similarly, you can see the PC the instruction showing fetch address 1, 2, 3, 4, 5, 6 sequentially. The IR you see the opcodes of the instruction, first opcode, second opcode, third opcode like this you can actually analyze what is going on in the pipeline, what Verilog code you have written you can actually see it step by step. And here I strongly suggest you look into this timing diagram very carefully, you run your own version and see exactly what is happening step by step, then it will be very clear to you, right.

(Refer Slide Time: 22:04)

Example 3

- Compute the factorial of a number N stored in memory location 200. The result will be stored in memory location 198.
- The steps:
 - Initialize register R10 with the memory address 200.
 - Load the contents of memory location 200 into register R3.
 - Initialize register R2 with the value 1.
 - In a loop, multiply R2 and R3, and store the product in R2.
 - Decrement R3 by 1; if not zero repeat the loop.
 - Store the result (from R3) in memory location 198.

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

Now, let us come to a slightly more complex program, which involves a loop. Loop means iteration. So, we are trying to compute the factorial of a number N . So, what you do suppose the factorial of a number N , I will give a value of N , let say N equal to 5. So, I take a variable initialize to 1 and multiply it with 5, decrement 5. 4 is it 0 or no multiply with 4, again decrement 3, multiply with 3, 2, 1, and when it reaches 0, I will stop. And whatever is this register contained at the end that is the value of the factorial.

So, actually we have done this. We have assumed that the number N is stored in memory location to 100. And the factorial of the number, let us say we want to store it in memory location 198.

(Refer Slide Time: 23:14)

Assembly Language Program		Machine Code (in Binary)
ADDI	R10, R0, 200	001010 00000 01010 0000000011001000
ADDI	R2, R0, 1	001010 00000 00010 0000000000000001
LW	R3, 0(R10)	001000 01010 00011 0000000000000000
Loop:	MUL R2, R2, R3	000101 00010 00011 00010 00000 000000
	SUBI R3, R3, 1	001011 00011 00011 0000000000000001
	BNEQZ R3, Loop	001101 00011 00000 111111111111101
	SW R2, -2(R10)	001001 00011 01010 111111111111110
	HLT	111111 00000 00000 00000 00000 000000

IIT Kharagpur | NPTEL Online Certification Courses | Hardware Modeling Using Verilog

So, again we have done it step by step fashion, but I will be explaining with the program, it is here. This is our program. See what we have done? In the first instruction we are initializing register R10 with 200, where the value of N is stored. And this R2 is initialized to the value 1, where our factorial will be finally stored, we will be multiplying with R2, ok.

Now, we are loading the number, R10 contains the address 200, 0(R10) loaded, store it in R3. So, R3 contains the number N, this is a loop. So, these 3 instructions we repeat, what we do, multiply R2 and R3, store the result in R2.

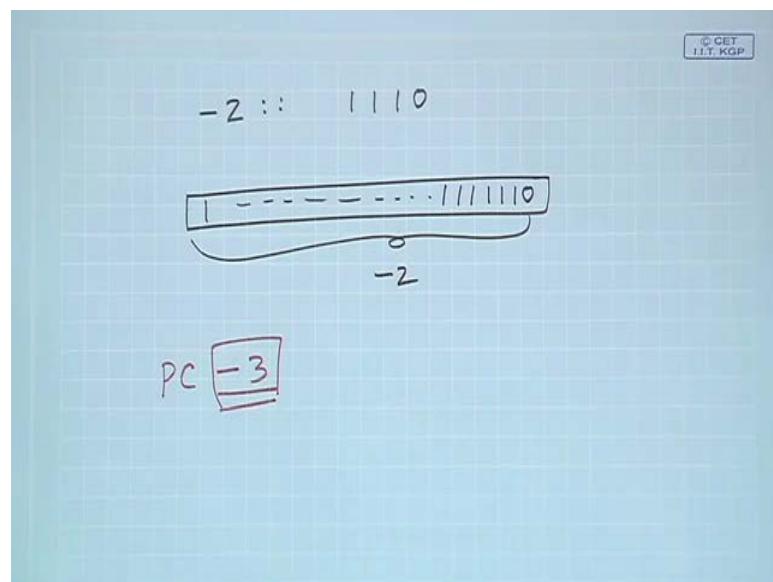
So, if let us say this R3 was 5, we multiply 5 with 1, store it in R2, R2 becomes 5, subtract 1 from R3 and store it back in R3. So now, R3 becomes 4. You check not equal to 0, is R3 is 0 or not 0, if it is not equal to 0 go back to loop. So, this 4, not equal to 0, go back to loop, multiply 4 with 5, 20 stored in R2, subtract 1 from R3, again it becomes 3 not equal to 0, again go back to loop. So, 20 multiply by 3, 60. So, like this it will go on, right. Here multiply number one by one, and as soon as R3 become 0, you will be coming out of the loop and you will have to store the result in 198 memory location, right.

Now, R10 contains memory location they are just 200. So, we write like this SW R2, -2(R10), which means R10 minus 2, in this address we want to store it. So, you can see the instruction encoding it is very similar, but only one thing I want to tell you, say add

immediate, we have already seen how to encode, load also I have said let us see this load you see once more, R10 this is 10 source 3, this is your R3, right. Now multiply similar this is multiplying R2, R3, R2; R2, R3, R2; subtract immediate R3, R3 and this is 1, ok.

Let us look at the store instruction in particular I would like to concentrate with the offset minus 2. You see here we have specified the offset as minus 2, and in the instruction encoding how offset is 16-bits.

(Refer Slide Time: 26:15)



So, how much is -2 in 16-bits? Well for those of you who are familiar with 2s complement number you may be knowing that in 4-bits representation -2 can be represented as 1110, this is -2

Now, if I want to extend it to 16-bits, I just mentioned I told you the rule for sign extension. So, whatever number you have you take the sign, it is 1, you replicate 1 as many times you want. The value of this number will still be minus 2, right. So, we have done the same thing you see -2 is this all 1s and the last as 0.

Similarly, let us look at the branch instruction. BNEQZ as an opcode of this, R3, this is 3, it does not require the second operand. So, it is second register it is 0. And loop is here. So, what should be stored here, let us see. Here the memory address is what if this is 00,1,2,3 loop is 3, right, 4, 5, this is 5. So, when I am executing this already PC has been implemented, PC contains 6. So, from 6, I have to go back to 3.

So, 6 to 3 means it should be -3. So, actually what I have to do, whatever is the value of the program counter I have to subtract 3 from it. So, this -3 will be my offset. So, in this branch instruction the loop here whatever I wrote this is the 2s compliment of -3.

So, the PC is pointing here -1, -2, -3 it will be jumping here after this, right, fine.

Let us now come to the test bench. The first part is same, then these are the instructions. You see we have inserted some dummy instructions again in between one here, one here, one here.

(Refer Slide Time: 28:22)

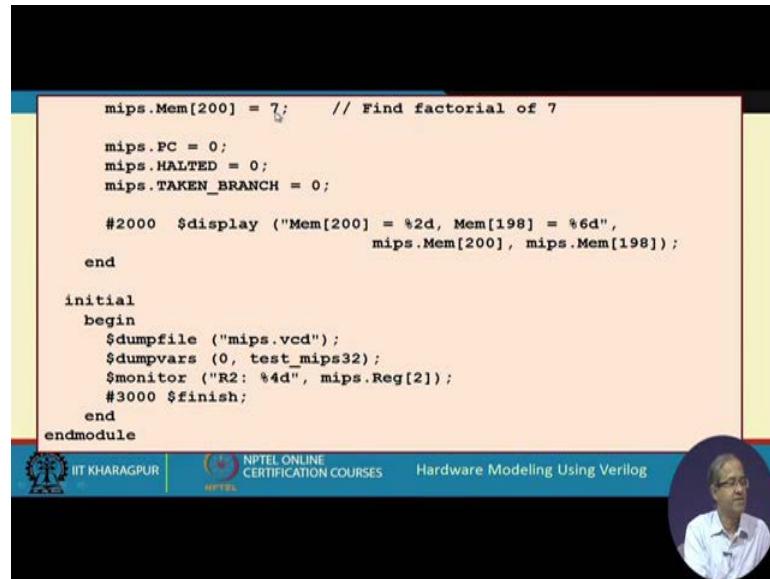
```
initial
begin
    for (k=0; k<31; k++)
        mips.Reg[k] = k;

    mips.Mem[0] = 32'h280a00c8; // ADDI R10,R0,200
    mips.Mem[1] = 32'h28020001; // ADDI R2,R0,1
    mips.Mem[2] = 32'ho0e94a000; // OR R20,R20,R20 -- dummy instr.
    mips.Mem[3] = 32'h21430000; // LW R3,0(R10)
    mips.Mem[4] = 32'ho0e94a000; // OR R20,R20,R20 -- dummy instr.
    mips.Mem[5] = 32'h14431000; // Loop: MUL R2,R2,R3
    mips.Mem[6] = 32'h2c630001; // SUBI R3,R3,1
    mips.Mem[7] = 32'ho0e94a000; // OR R20,R20,R20 -- dummy instr.
    mips.Mem[8] = 32'h3460ffffc; // BNEQZ R3,Loop (i.e. -4 offset)
    mips.Mem[9] = 32'h2542ffff; // SW R2,-2(R10)
    mips.Mem[10] = 32'hfc000000; // HLT
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog | 18

So, wherever there is a dependency we have inserted a dummy, you see here, of course here it was not really required, R2, here R3 and R3 we have inserted. Here R3 and R3 here also we inserted and the opcodes have been coded. So, here we need 11 instructions in total. So, as usually we have initialized register to something, but there is not only required, but we have done it.

(Refer Slide Time: 29:04)



```
mips.Mem[200] = 7;      // Find factorial of 7
mips.PC = 0;
mips.HALTED = 0;
mips.TAKEN_BRANCH = 0;

#2000 $display ("Mem[200] = %d, Mem[198] = %d",
                mips.Mem[200], mips.Mem[198]);
end

initial
begin
    $dumpfile ("mips.vcd");
    $dumpvars (0, test_mips32);
    $monitor ("R2: %d", mips.Reg[2]);
    #3000 $finish;
end
endmodule
```

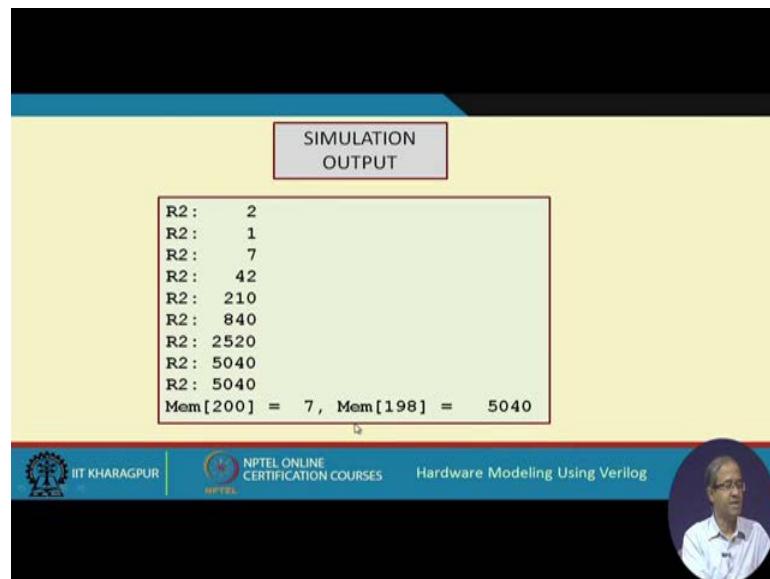
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Hardware Modeling Using Verilog



And suppose we want to calculate the factorial of 7. So, in memory location 200 we are storing 7.

So, as usual we have initialized PC to 0, HALTED to 0, TAKEN_BRANCH to 0. And at the end after sufficient delay we are displaying the contents of memory location 200 and memory location 198. And we have also monitored so that whenever it changes it will be printed the value of R2 because you recall R2 contains the value of the product, you are multiplying it into R2, right.

(Refer Slide Time: 29:43)



SIMULATION OUTPUT

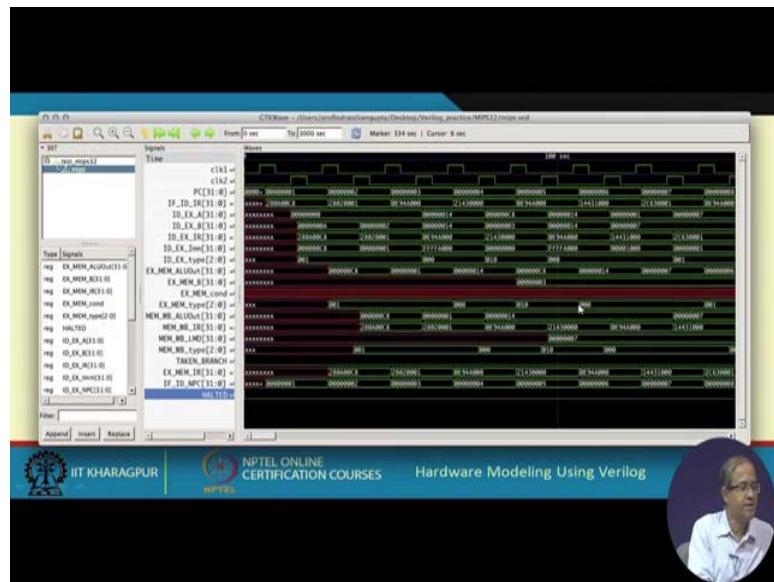
```
R2: 2
R2: 1
R2: 7
R2: 42
R2: 210
R2: 840
R2: 2520
R2: 5040
R2: 5040
Mem[200] = 7, Mem[198] = 5040
```

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



So, if you run it the simulation output comes like this you see, at the end you see Mem[200] is 7, Mem[198] is 5040, you can verify factorial 7 is actually 5040. And R2 well this initially you are initializing all register k to k, initially it was 2 after that 1 multiply with 7, multiply with 6, multiply with 5, 4, 3, 2 you get final 5040 like this.

(Refer Slide Time: 30:18)



And in a similar way If you look at the timing diagrams you can see step by step what is happening.

Now, I will leave it as an exercise for you because this is quite complex and the font size it is also not quite large. So, I am not sure whether you are able to see it very clearly, but actually you can trace for the, this is the first part of the timing diagram, this is the second part of the timing diagram. Still you can say execution is going on, it is not finished. Well, when finished will be done and HALTED is still 0, HALTED will become 1 after that, ok.

(Refer Slide Time: 30:55)

Point to Note

- We have not considered the methods for avoiding hazards in pipelines.
- For the examples shown, we have inserted dummy instructions between dependent pairs of instructions.
 - So that data hazard does not lead to incorrect results.
- Also, we have modeled the processor using behavioral code.
 - In a real design where the target is to synthesize into hardware, structural design of the pipeline stages is usually used.
 - The Verilog code will be generating the control signals for the pipeline data path in the proper sequence.

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

So, just one thing I would like say here, see we have taken three very simple examples, but as I said the instruction set or the instruction that we have selected they are powerful enough. Well, if you want let us say, you can try and write a program to compute the GCD of 2 numbers, that is quite possibly this instruction set. You can try and write a program to sort a list of N numbers, using any sorting algorithm like bubble sort or insertion sort. It is possible, even with the small instruction set.

So, what I mean to say is that, this instruction set even though it is very small, we have chosen the some of the essential instruction which are required for condition checking, arithmetic, looping and so on. So, that you can write meaningful programs using this. Well of course, if you want to extend say by adding more instructions, you can always make changes to a Verilog code you can add more instructions, ok.

Now, there are a few points we would like to mention here. That we talked about hazards, but in the design that you are presented we have not considered any method for avoiding hazards in pipeline. What I have assumed is that if there are hazards we are inserting dummy instructions. So, it is the users responsibility to do that.

Well there are many instances where this is actually done, but not the user, the compiler. Suppose there is a C compiler for the MIPS machine, someone has written a code in C and when the C compiler is generating machine code, it automatically checks whether there are any dependencies between consecutive instructions. If So, it will first try to

move some instructions around if possible if not it will insert automatically such dummy instructions So that the final code will be executing correctly on the pipeline, right

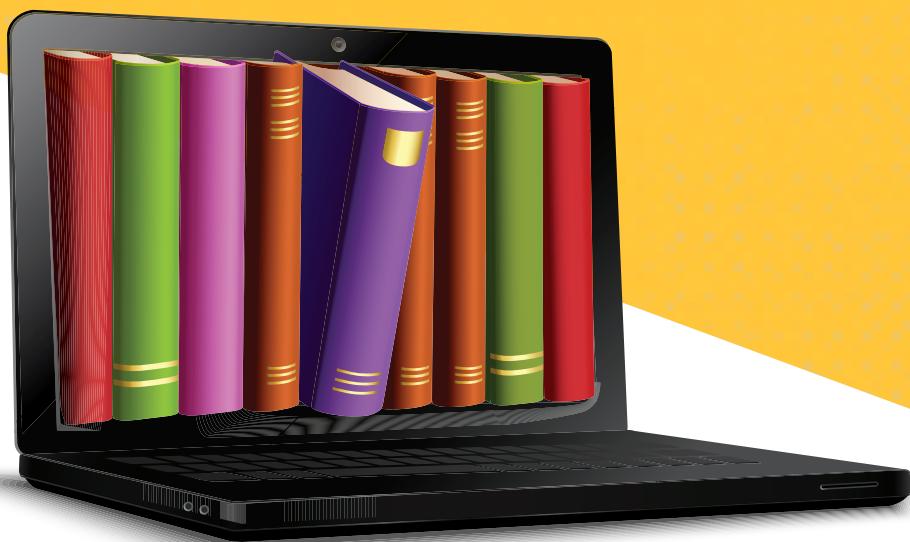
So, this is what I mentioned for the examples where we inserted dummy instructions such that results obtained are correct. And another important thing is that the processor model that we have considered is purely behavioral model. So, we have done or we have written the code using if then else kind of statements, which is quite similar to say program to which we write in high level language. But in an actual real design where you are actually seriously wanting to synthesize your design, it is always better to go for structural code. All the pipeline elements like the registers, ALU's, register banks they will all be part of your data path, we have already seen earlier how to separate the data path and control path.

So, once you have the date path components, you will also have to identify the control signals to activate them. Now inside your main pipeline Verilog code you will be just activating the control signals one by one. That is what will be there in the code, right. Just like the control path or the controller you are designing those earlier examples, ok.

So, this is what is mentioned here. So, with this we come to the end of this lecture, in fact, this is the last lecture of the series. So, we have seen various features and structures of the Verilog language. We have looked at a number of examples as well. So, we hope with this background you will be able to create some more meaningful and serious designs, which will be helpful in the domain that you are working in.

Thank you.

**THIS BOOK IS
NOT FOR SALE
NOR COMMERCIAL USE**



(044) 2257 5905/08



nptel.ac.in



swayam.gov.in