VLSI Design Flow: RTL to GDS

Dr. Sneh Saurabh
Department of Electronics and Communication Engineering
IIIT-Delhi

Lecture 4
Overview of VLSI Design Flow: II

Hello everybody, welcome to the course VLSI Design Flow: RTL to GDS. This is the fourth lecture, and we will continue with the overview of VLSI design flow. In the previous lecture, we looked into system-level design. We had discussed how the functionality at the system level is partitioned into hardware and software. Now, given a function that needs to be implemented in hardware, what are the next steps? The next step is to make a specification from that given hardware and then take it through the VLSI design flow. So, we will discuss that part of the system-level design in today's lecture.

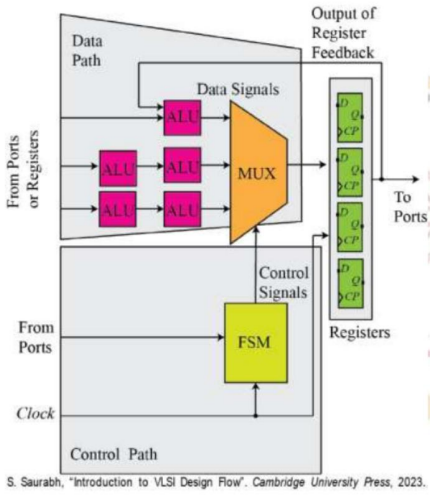
In particular, we will look at how a functional description at the system level can be translated to the RTL. And then, we will be looking into the concept of reusing RTL and behavior synthesis. Once we have partitioned a system into hardware and software, we take the hardware component and develop a functional specification for it. This functional specification can now be made in a high-level language, for example, C, C++, or MATLAB.

Why will we want to make a specification in high-level language? Because there, we have lots of flexibility and can do a lot of experimentation and analyze various trade-offs of the algorithms on the system behavior and other things. Therefore, we want to do an initial analysis or make a functional specification in a high-level language where we have lots of flexibility. But there is a downside to this also. The downside is that once we make a specification in a high-level language, we need to convert that specification to an RTL. And that creates what is known as an implementation gap. So we write a functional specification in C, C++, SystemC, MATLAB, or it can be a Metadata. Finally, we need to convert it into an RTL, typically in Verilog, SystemVerilog, or VHDL. So now this conversion needs to be done, and that is what we mean by the implementation gap is there.

Now, this is an important implementation gap. Why? When we describe functionality in a high-level language, we do not have information on timing, meaning a given task needs to be performed in which clock cycles. But when we want to get an RTL out of it, we need to describe the design in terms of how the data flows from register to register at various instants or clock cycles. So, an RTL carries timing information, i.e., which

operation of the function is being performed in which clock cycle. And we need to make important design decisions such as pipelining and resource allocation, and those kind of things that needs to be there in the RTL.

This timing information is missing in the functional description or a high-level language; therefore, we need to fill that implementation gap. Now, how to fill that implementation gap, we will see later. Before that, let us understand briefly what an RTL is and what is the structure of a general RTL. In this figure, we are showing the general structure of an RTL. So, in RTL, what we do is that we model the flow of data between registers, and therefore, RTL can also be called data flow descriptions.



S. Saurabh, "Introduction to VLSI Design Flow". *Cambridge University Press*, 2023.

It has two distinct parts: the data path and the control path. In the data path, major portion of the computation is done. For example, there are ALUs or arithmetic logic units which is doing a kind of computation. These units can be adders, multipliers, or logical units like AND gate, OR gates. These kinds of logical operations will be performed in the data path.

How this data moves is decided by the control signals generated in the control path. So, the FSM generates these control signals on the control path, and MUX passes the data based on these control signals. And we have registers from which data or information can go to the arithmetic logic units, and so on. So, this is the general structure of an RTL. An important point to note here is that in an RTL, there can be computation, which can be in series. For example, if we consider this path, the first ALU computes, then the second ALU computes, and so on.

So, this is a kind of serial computation. And there are some ALUs, for example, if we consider this ALU and the other one. So, these two ALUs are performing computation in parallel. So, in an RTL, there can be processing, which is done in serial, for example,

these two, and there can be processing of data, which can be done in parallel like these two.

So RTL needs to be modeled in a language or framework where both kinds of modeling are allowed, that is, sequential modeling as well as concurrent modeling. Hardware description languages provide that facility. RTL is typically modeled using Verilog, VHDL, or system Verilog languages. We have discussed that there is an implementation gap when we describe a design in terms of functional specification in a high-level language and the RTL, which describes design in a way in which data moves from register to register. Now, how do we fill this implementation gap? There are typically three methods by which this implementation gap can be filled.

The first one is, of course, the manual coding. Given an algorithm, we can code an RTL manually and describe which arithmetic operation or which kind of computation is done in which clock cycle and add the timing information and other details in the RTL manually. That is one way of doing things. The other is IP assembly, which is reusing an existing RTL. This is typically used in SoC design methodologies.

And then there is the third way, using an automatic tool. So, there are tools known as behavior synthesis tools that can automatically generate an RTL for a given description of a design in a high-level language. So, we will look at the manual coding of a design in Verilog in later lectures. For now, let us look into this IP assembly method or reusing the existing RTL method and the behavior synthesis method to fill the implementation gap. Now, reusing RTL is especially popular in SoC design methodology.

Now, what is SoC? SoC is a system on a chip. So, what is a system-on-chip? A system-on-chip is a complete system built on a single chip. Earlier, what we used to have is that we used to have various components, for example, separate processors, separate memory, and various other components separately for a system, and we used to integrate them over a board, and connect them and get the required functionality.

But what happened is that with the increasing level of integration, now, in a given chip, we can embed all those things together inside one chip. That chip in which various components are integrated together is known as a system-on-chip or SoC. Now, what an SoC can consist of? It can consist of processors, hardware accelerators, memories, peripherals, analog components, and RF devices. These are a few examples. There can be many other things that are connected using structured communication links. These SoCs can also contain embedded software.

Now, what is the merit of SoC design methodology or SoCs is that it improves productivity. Why? Because what we do in SoC design methods is that we do not design individual components. We reuse the existing components. Reusing the existing components means that if we have already designed a processor, we take that processor

and put it in our design. Then, we take, say, a memory component, which was pre-designed, and put it in the SoC.

So, the designer's task now is not designing a complicated chip in which each individual component is designed but only integrating those components. So, the task of designing reduces to integrating various components together at the system level and, of course, verifying. This is a simpler task than designing each component at the individual level. And that is how we improve productivity in SoC-based design. It also lowers the cost because we take less time to design and reuse many things.

So we do not spend much effort in already done things. Therefore, it saves cost in designing. And it increases the features. Since we can easily integrate many things together on the same chip, we can have many features that were not possible earlier. So, the SoC design methodologies allow us to implement very complex features or complicated functionality within a single chip.

These are the merits of SoC design methodology, and therefore, these days, a lot of industrial designs are designed using SoC design methodologies. An integral part of SoC design methodology is intellectual properties or IPs. Now, what are IPs? IPs are pre-designed and pre-verified subsystems or blocks. For example, it may be a processor, or it can be a memory unit, or it can be peripherals, etc. Now, these are pre-designed.

This pre-designing can be done either in-house by a company or a subsystem, or an IP can be purchased from third-party IP vendors. So, it can be developed internally, or it can be purchased from the IP vendors. Now, what is the content of intellectual property? It can be hardware blocks, for example, processor, memory, and interface, or it can be software, like real-time operating systems, device drivers, and so on. So, the content of IPs can be varied depending on what the IP is.

Even some IPs can have verification components. For example, what is the communication protocol for a given IP, and so on. Those things are there, already designed, verified, and encapsulated in a format so that it can easily be instantiated in another design. These verification IPs very much help in or ease the verification effort or reduce the verification effort. Now, while designing using the IP method or SoC design method, an important concern or important aspect is how to share the information. Why is this important? Typically, the IP is made by another group or company, which is taken from them and then put or integrated into another design. The designer who is designing or integrating the component may not have worked on that IP earlier and may not have the details of how to use it.

So, IPs contain information related to the structure, configurability, and interfaces of the system, and the challenge is how to package this information so that the IP integrator

who is using that IP in the SoC can understand how to configure that IP and connect that IP. This is a challenge because the person or group who is making the IP is different, and the group who is using that IP is different. This problem becomes more difficult because, typically, uniform standards are not followed in this kind of package, and this is a challenging task. And the IP vendors or one who is creating the IP must be careful about how they are sharing the information, and similarly, the one who is using those IPs must be careful of how they are using the IP in their designs, and they are not violating the assumptions that are made by the IP vendors or the one who has designed the IP.

So, now, in the integration of IPs or how the SoC design is made, the major task for a designer is to instantiate various IPs and then make a connection out of them. That is the major task because we assume that IP is already there, it is already designed and verified, and available to us for use in our design. So, as a designer of an SoC, the major task is to instantiate those IPs and make the connections between them. Now, how do we do it? So, to do this integration, what an IP integrator or the one who is designing an SoC can do is they can create Metadata. So, in these Metadata we define the top-level IP models, bus interfaces, ports, registers, and the required configuration. These need to be filled in by the IP integrator. And there are some standards for filling in that information; for example, there is an IP-XACT, SystemRDL, XML, or a spreadsheet that can be used for filling in this information about these Metadata, like what are the IP models that need to be there in your design, what are the bus interfaces, what are the ports, what are the registers, and what is the configuration of each of the IPs.

Earlier, this kind of integration used to be done manually, but those were error-proof. Now, there are generator tools that produce an SoC-level RTL with the instantiated IP rather than manually integrating those IPs or writing an RTL for the integration of those IPs. What we can do is that we can use a generator tool. The generator tool will read the Metadata about which IP to use, how to connect, and other things and automatically produce an RTL with the instantiated IP.

A generator tool can also produce a verification environment and low-level software driver. This can greatly help the designer and allow us to avoid mistakes in IP integration. Nevertheless, configuring IPs can be challenging because IPs can have configuration parameters such as bus width, power modes, and communication protocols. So now, when making or integrating an IP in our design, what to choose for the bus width, what to choose for power models, and so on. So there, IP assembly involves choosing the set of configuration parameters, and this task is challenging because there can be too many parameters and simply too many combinations of parameters that can be chosen.

Out of them, which one is optimum? That is a difficult question to figure out. And the other problem is that there can be inconsistency between different IPs. For example, if we choose one IP with a width as 8 and other as 16 it may be incompatible for communication. So, when different IPs are interacting with each other, the configuration

should also be consistent. That is another thing that we must ensure or check while doing IP assembly.

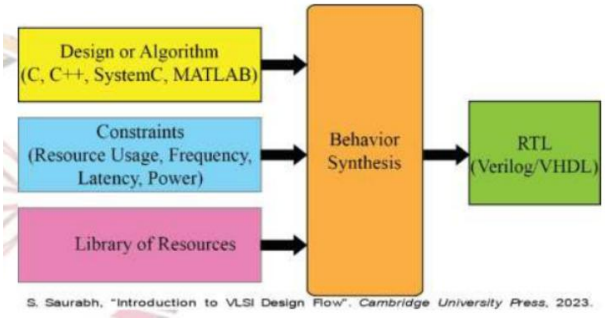
Another thing is that how do we connect different IPs? Earlier, what we used to do is we used to connect different IPs using ad-hoc bus-based connection. It was okay till the time the connections were not very complex, but with these modern SoCs, the level of complications between the connections between IPs has increased tremendously. As such, the ad-hoc method of manually connecting or making bus-based connections does not work, and now what we do is that we follow a structured network approach, which we borrow from computer networks theory and apply it in VLSI. These structured networks inside SoC are known as network-on-chip or NoC.

Another challenge of the integration of IPs is verification. When IPs are already there and we have integrated them, integration is a simpler task. After integration, we also need to verify that we are getting the required functionality. This verification needs to be done throughout our design or for the entire design. This verification is a difficult task because there is a huge functional space over which we need to verify. There are too many configuration parameters and many modes of the circuits or the IPs that it can work on and so on.

Then, we have to verify that under all conditions in which our IPs can work, we are getting the required functionality, and therefore, this is a challenging task. Added to that, if we consider that we also need to test or verify if the hardware portion of SoC is working correctly with the software portion, then our verification challenges increase. So these are some of the challenges that we need to consider while doing SoC design.

Now, the other way in which we can convert functionality in a high-level language to an RTL is behavior synthesis. So, behavior synthesis is the process of converting an algorithm that is not timed, meaning it does not carry the timing information, i.e., which operation needs to be performed in which clock cycle that information is not there, to an equivalent design in RTL, which is fully timed. Meaning that it is saying how the computation is done in which clock cycle, and so on. While doing this transformation, it must satisfy the specified constraints of resource usage and latency, etc. that we will see.

So, the framework of behavior synthesis is shown in this figure. We give the behavior synthesis tool the given algorithm, which may be in C, C++, SystemC, or MATLAB. Then we give the constraints of resource usage, frequency, latency, power, and the library of resources, meaning that if we have adders, multipliers, and other entities in our design and if the behavior synthesis tool uses them, then what will be the cost implications of each of these entities. And at the end, what the behavior synthesis will give us? It will give us an RTL model, typically in Verilog or VHDL. So behavior synthesis is also called high-level synthesis.



Now, what are the cost metrics that need to be considered in behavior synthesis? An untimed algorithm can be implemented in many ways. So, for the behavior synthesis tool, there are a lot of options because there are a lot of design decisions it needs to make, as there are many possibilities in which a given algorithm can be implemented. Now, out of those implementations, which one is better, and which one is worse? The behavior synthesis tool needs to evaluate the cost of each implementation to do that kind of analysis and pick the one with the lowest cost.

So, what are those cost metrics? So, the first one is the area, which is easy to estimate. It can be measured simply from the number of circuit elements. The number of circuit elements in the RTL that will be used can give us the kind of area measure for the RTL generated by the tool. Now, at a broad level, we can ignore the area of the wires, and we can just consider the area of the elements, logic elements, arithmetic units, and registers, etc.

Then, the second cost metric can be latency. What is latency? It is the number of clock cycles required before results are available. So if we make a change at the input, then after how many clock cycles that result is available at the output, that is called as latency. And then the third important thing is at what maximum clock frequency a circuit or a design can operate, and it can be measured using worst-case combinational delay. This requires a little more understanding of the timing.

So, we will be discussing this in the next slide. And what could be the other cost measure? The other cost measures could be power dissipation, and throughput, and there can be many other considerations that behavior synthesis tools account for while generating a solution. Now, let us understand what is the maximum clock frequency and how it can be estimated for a given design. So, to get a full understanding of this, we need to go into the static timing analysis, which we will be doing later in this course. But we can get a crude estimate of what factors the maximum clock frequency of a synchronous circuit will depend on. In this course, we will be mostly looking at synchronous circuits.

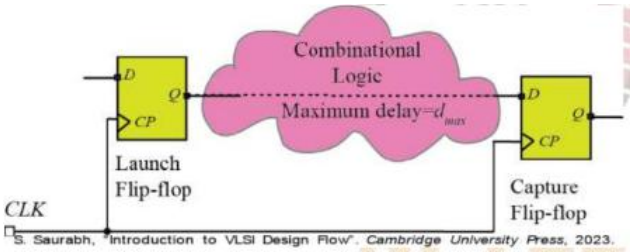
So, let us first assume that there is a synchronous circuit, and for that, let us look into a few definitions, and then we will look into how the maximum clock frequency can be determined. So, in a synchronous circuit, we define what a path is. So, the path is a sequence of pins through which a signal can propagate. For example, if you have an AND gate, then an inverter, then a NAND gate, and so on. So, the sequence of pins will

be: input pin of AND, output pin of AND, input pin of inverter, output pin of inverter, input pin of NAND, output pin of NAND, and so on. So, this sequence of pins forms a path. The sequence of pins through which a signal can propagate is a path.

Now, what is a combinational path? So, before explaining the combinational path, let me explain the difference between a combinational circuit element and a sequential circuit element. Combinational circuit elements are those circuit elements whose output depends on the current inputs only. For example, the AND gate, OR gate, XOR gate, multiplexer, and decoder are combinational circuit elements because whenever we make any change to the input, the output can change. Then, there are sequential circuit elements in which the output depends not only on the current input but also on past sequences of it. For example, for flip flops or latches, the output will depend on the past value that it got at the D pin whenever the clock came last. So the combinational circuit elements are AND gate, etc., and sequential circuit elements are flip flops and latches.

Now, what is a combinational path? So, a combinational path is a path that does not contain any sequential circuit element, such as a flip-flop. So flip-flops, latches, or any other sequential circuit element cannot come in a combinational path. This means that only combinational circuit elements, for example, AND gate, OR gate, or those kind of combinational circuit elements, exist in a path, then we say that it is a combinational path.

Then, there is a concept of sequentially adjacent flip-flops. Now if the output of one flip-flop is fed as an input to other flip-flop through a combinational path only, then we say that it is a sequentially adjacent flip-flop. For example, consider this flip-flop. The output from this Q pin (of launch flip flop) goes to this D pin (of capture flip flop) through the combinational logic. So, this pink cloud shown here consists of combinational circuit elements only, and therefore, any path from this Q pin to the D pin will pass through a combinational path only. Therefore, if the output of one flip flop, in this case, Q pin, is fed as an input to the other flip flop through a combinational path, then we say these two flip flops are sequentially adjacent.



The one which is launching the data that is known as the launch flip-flop, and the one which is capturing the data is known as the capture flip-flop. So this is the capture flip-flop, and this is the launch flip-flop. In a synchronous circuit, the data that is launched at a particular clock cycle that should reach the D pin of the sequentially adjacent flip flop by the next clock cycle or next clock edge.

So, that maximum time allowed for the data to move from this Q pin to the D pin is one clock period. So, we can write that the clock period should be greater than the delay of

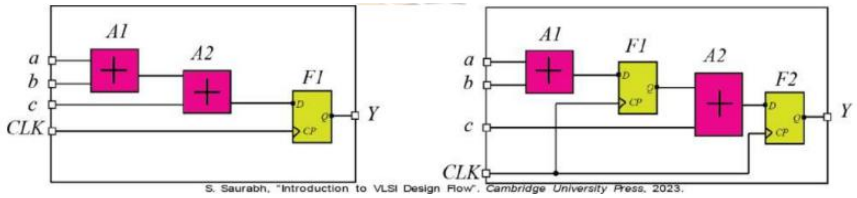
the critical path: $T_p > d_{max}$. What is d_{max}? d_{max} represents the maximum delay between Q and D. Note that between Q and D, there can be multiple paths. Out of these paths, the one that is exhibiting the maximum delay is d_{max}.

So, for a synchronous operation, we want that by the time the next clock edge comes, the data should start from here and reach the D pin. And therefore, we have the constraint that T_p, that is, the clock period, should be more than the d_{max} or the maximum delay on the data path. And the clock frequency is the reciprocal of the time period, and therefore, $f_{max} < 1/d_{max}$.

Therefore, we can say that the path deciding the clock frequency is the combinational path with the largest delay. The one that has the largest delay will determine the maximum clock frequency at which a synchronous circuit can operate because if we increase the clock frequency more, then the clock edge will reach very early at the capture flip flop, and the data will not be captured synchronously at the capture flip flop.

So, this is just an approximate description of how the maximum clock frequency for a circuit can be calculated. But we will look at the detail of these calculations in lectures of this course when we discuss static timing analysis. Now, let us take an illustration of behavior synthesis how behavior synthesis is done. So, suppose we are given an algorithmic description:
Y = a + b+ c.
Now, this is an algorithmic description there is no description about when the plus operation is carried when the another plus operations is being carried out.

Now, for this, it can be implemented in many ways, and the cost metrics we will consider is, say, latency, the area or the circuit elements that are used in the circuit, and the maximum delay of the combinational path. So, one of the implementations can be where we have two adders. a + b is performed by adder A1, then the result goes to the next adder, and we are getting the result, which is going to the D flip-flop, and whenever the clock edge comes, the result is available at Y. And for this implementation, we can compute the cost. The resource usage is two adders and one register. The latency is one clock cycle because if we make a change in the input, the change in the output will be available in the next clock cycle. So, latency is one clock cycle in this case, and the worst case delay or the longest combinational path delay will be, say, the path in which you have two adders delay.



S. Saurabh, "Introduction to VLSI Design Flow", Cambridge University Press, 2023.

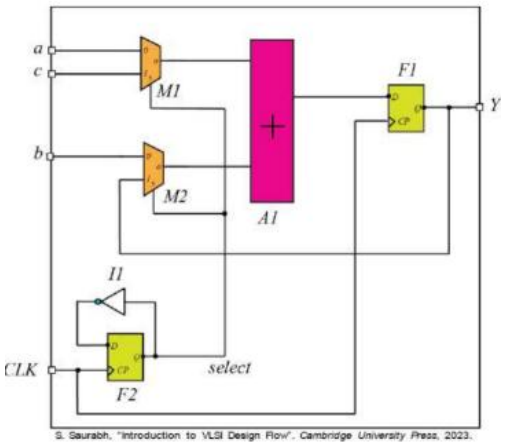
Now, another implementation can be in which we added a flip flop in the middle. Now, as a result of that, what changes can happen? First of all, the area increases by one flip-flop, or the area of one register increases. Then, the latency increases to two clock cycles.

Why? Because if you make a change in input ‘a’, then the effect on Y will be after two clock cycles. Because two flip-flops are there, and therefore, latency increases from one clock cycle to two clock cycles. But what happens to the worst-case delay or the longest-path delay? So, the longest path delay is of one adder between sequentially adjacent flip flops or one adder from input to thefirst flip flop.

So, in this case, the worst-case delay is one adder. So, in the earlier case, the worst-case delay was a delay of two adders. Now in this case, we have the delay of only one adder, and therefore, this circuit can operate at a higher clock frequency. Why? Because we saw that:

$f < 1 / d_{max}$.

Now, in this case, d_{max} is related to only one adder compared to two adders, and therefore, f will be greater for the second case. Now, let us look into another implementation of the same thing in which we have used only one adder.



S. Saurabh, "Introduction to VLSI Design Flow", Cambridge University Press, 2023.

Now, the adder is being used to compute a + b in one clock cycle. Whenever select is equal to 0, then a and b come to the adder input, and you get the result as Y = a + b. Whenever the clock comes, the output is a + b, and this is fed to M2. And whenever the next clock edge comes, what happens is that since Q is connected to D through an inverter, this Q will toggle every clock cycle. So in one clock cycle, it will be 0, then in the next clock cycle, it will be 1, then 0 and 1, and so on.

So, in the next clock cycle, what will happen is that this select signal will take a value of 1. So when 1 is taken, the 1 lines of the multiplexers are selected, and in that case, what operation is performed by the adder is: (a + b) + c. So, in the next clock cycle, we will have a + b + c. So, the adder is used in the first cycle to compute a + b, and in the next cycle to compute a + b + c, that is, c + Y.

So, inputs to the adders are controlled by multiplexers. Multiplexers get select signals from the control circuitry. So, what is the benefit of this implementation? The benefit is clear that we are using only one adder. Now, typically, what happens is that adders, multipliers, and this kind of arithmetic unit are more costly in terms of area, power, and other things. So, if we reduce the count of these arithmetic units, we can save on area.

But what is the downside? Now, we will have the latency of two clock cycles. In one clock edge, it produces a + b, and in the next, it produces a + b + c. So, every two clock cycles, it is producing the result. So the latency will be two clock cycles, and the throughput, that is, how many computations are done in one clock cycle, is also reduced in this case. But what benefit are we getting? The benefit is in terms of area and possibly in terms of power also. And what is the worst-case delay in this case? It is an adder plus multiplexer. So whichever is the longest path, from 'a' to Y or 'b' to Y or 'c' to Y. So, one of these paths will be the critical path, deciding the clock frequency.

Now, if we assign area and delay to these circuit elements, then we can easily compute the cost of each of these implementations. For example, for the implementation of RTL-1, we can say that the total area is the area of 2 adders, that is, 200 + 200 = 400, and 1 register that is 12, so the total area = 412.

Similarly, we can compute the area for other other implementations. We can compute the latency, and we can compute the delay for each implementation. And then what the behavior synthesis tool will do is that out of these three kinds of timed implementation for the algorithmic description: Y = a + b + c and there can be other implementations also, (which we have not considered, we have chosen only three for illustrative purposes.), the behavior synthesis tool will choose the best possible implementation satisfying the constraints.

Now, what would be the best one? It will depend on the constraint, what objectives we have, or what kind of design we want to implement. For example, if we want that area to be minimized. So, in that case, the tool will choose the RTL-3, the third implementation in which the area is the minimum. But then the downside will be, of course, the throughput and the latency. If we want to minimize the latency, then we will probably use the first implementation, i.e., RTL-1. And if you want to maximize the clock frequency, then we can probably choose the RTL-2, which has got the minimum delay. So, depending on objectives, the behavior synthesis tool can produce one of these outputs among all the evaluated outputs.

Now, what are the merits of behavior synthesis? So, the merits of behavior synthesis are that it allows automatic exploration of different possible implementations. For example, for a human designer, it may not be that easy to think of all three implementations; at least the third implementation was non-trivial. And therefore, those kinds of implementation are difficult for a human to come up with. So, the solutions produced by the behavior synthesis tool will be after a more exhaustive search of the design space than the handwritten RTL. It reduces the design effort because it is done by an automatic tool, and it has less chance of introducing errors compared to handwritten RTL. So, these are the benefits of behavior synthesis.

And what are the challenges or demerits of this? The first is related to physical design. During system-level design, the tool is not knowing much about the implementation over the layout and other details. And as such, some solution can be produced that can later be

found to be bad in terms of physical design or so. For example, the behavior synthesis tool can share a resource too much. So, a lot of signals come to a shared resource, and there can be congestion around that shared resource. And that congestion will be found only during physical design. So, the solution that may be produced by the behavior synthesis tool may actually lead to some problem down the design flow because of the incomplete information it has.

The other is the lack of readability. The machine-generated RTL code after behavior synthesis is not easy for a human to read and understand. So, if we want to make incremental changes or make some debugging in that RTL generated by the behavior synthesis tool, it is difficult. So that is a big demerit. Another problem is that if you make small changes in the algorithmic behavior, the RTL can change widely. A small change in the algorithm can lead to a very big change in the generated RTL, and therefore, making small adjustments to the algorithm and other things becomes really difficult with the behavior synthesis tool. There are verification challenges ensuring that the algorithmic description and the RTL are both actually matching in terms of functionality; that is, again, a difficult challenge that needs to be tackled by the designer who is using the behavior synthesis tool.

So, these are some of the important references that you can look into for more details on the topics that I covered in this lecture. So, in this lecture, we have looked into that from given function how we can get an RTL. So this completes the first part of the Overview of VLSI Design Flow, in which we have looked into the system-level design. So, we have discussed very briefly what system design is. And then, in the next lecture, we will be looking into the flow from RTL to GDS. That is the main topic that we will be discussing in this course. But first, we will be looking into an overview of RTL to GDS flow in the next lecture. So that is it for this lecture. Thank you very much.