VLSI Design Flow: RTL to GDS

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

Lecture 3
Overview of VLSI Design Flow: I

Hello everybody, welcome to the course VLSI Design Flow: RTL to GDS. This is the third lecture. In this lecture and in the following five lectures, we will be taking an overview of VLSI Design Flow, and then in the rest of the lectures of this course, we will be looking at the various design tasks in more detail. We have taken this approach of first taking an overview of VLSI Design Flow so that later on when we look into the various design tasks in more detail, we can appreciate the linkages between one design task and the other. We can also understand the impact of one design task on another and how we can optimize a given design task such that, down the flow, other tasks benefit from it. So, taking an overview and then going into the details of each task helps us understand the entire design flow in detail.

So, in this lecture, we will cover these topics:

1. An overview of how the design flow is structured.
2. We will look at an important concept, which is known as abstraction.
3. We will look into pre-RTL methodologies, and
4. Hardware-software partitioning.

Let us first look into the top-level problem we are trying to solve in VLSI Design Flow. The top-level problem is that we have some idea of implementing a system, and that system is doing some useful work. For example, it is doing some processing, or some computation, or it may be, say, controlling a robot or doing some useful work for us. So that is the starting point of chip design, or that is the idea we are having. At the end of the flow, we want a chip that delivers that task, and that chip should be manufactured in bulk. The fabrication of the chip should be such that the whole process is profitable for the person undertaking this venture. Now, this is a complicated task. It can be accomplished in many ways. Some of them can lead to profitable chip design, some may not, and some can actually fail. So, to tackle this complicated problem, what strategy do we follow? We follow the strategy of divide and conquer. We break the entire chip designing task and chip manufacturing task into multiple steps and carry out those steps sequentially. So let us look into at a broad level what these tasks are. So, given an idea and its implementation to a chip within this flow, the design undergoes many transformations.

Nevertheless, we can identify a few milestones through which a design goes through. What are these milestones? The first one we can say that it is the RTL. RTL is a design

representation in a register transfer level. It is typically written in Verilog or VHDL language. And then we take this RTL, and it goes through the design process.

Finally, we get a layout, which we say is in the form of a graphical database system, representing a layout. So, from the idea to the chip, we have identified two important milestones. One is the RTL and the GDS. Based on these milestones, we can divide the entire process flow into three parts. The first part is from the idea to the RTL.

In this step, we take a high-level idea concept of a product and represent the hardware portion of the idea into an RTL. We can call it as idea to RTL flow, or we can also call it system-level design. And once we have an RTL, then comes the RTL to GDS flow. So, in this, we take an RTL and take it through various stages of logical and physical design. What does logical and physical design mean? We will look into that in the later part of this course.

At the end of the RTL to GDS flow we get a layout that is represented in GDS form. And once we have got the layout, then it goes to the foundry for fabrication. This is called the GDS-to-chip process. It takes a GDS, prepares the masks for a given GDS, fabricates, tests, packages the chips, and then it goes to the market. So, in today's lecture, we will look primarily at this part.

In later lectures, we will take an overview of this RTL to GDS flow. Then we will also go into GDS to chip processes flow, we will be taking an overview, or we will be looking into them very briefly. But the main part of this course is this RTL to GDS flow, where a lot of effort is needed not only in the implementation of the design but also in the verification of the design. We have taken a top-level view of the VLSI design flow and how an idea is converted to a chip. Now, let us look into an important concept, which is known as abstraction, and how it relates to VLSI design flow.

So, what is abstraction? Abstraction is hiding lower-level details in a description. So, as the design moves through the VLSI design flow, details go on adding to the design, and as a result, abstraction decreases. Why does abstraction decrease? Because abstraction means hiding lower-level details and keeping only the thing that is required at the top level. So, as the design moves through the VLSI design flow, more details are added, and abstraction decreases.

So, if we look into the three parts of the design flow that we just discussed, the abstraction is very high in the idea to RTL flow. Here, we are representing a design in terms of system and behavior. Later on, as the design moves through various stages of RTL to GDS flow, the abstraction decreases, and the design gets transformed first from idea to RTL to logic gates, then transistor, and finally to the layout. Once we have got the layout, then there is no abstraction. All the details required for fabrication are already there, and then the actual implementation starts and the design in the fabrication step is represented by mask and integrated circuits or the final integrated circuit. Now, why do we do abstraction or abstract out information in VLSI design flow, especially at the

earlier stages of design? To understand that, we should first see what are the important considerations for a design task.

There are two important considerations. The first one is optimization. Optimization means choosing the right combination of design parameters to obtain the desired QoR by trading off some other QoR measures. The optimization is basically trying to find suitable parameters for the design so that we get a better quality of results or figures of merit. That is one part of a design task.

Another part of the design task is the turnaround time. Now, what is the turnaround time? Turnaround time is the time taken to make changes in a design. Ideally, we want to make these changes as quickly as possible. Why do we want to do that? Because we want to make design of a chip as soon as possible so that the chip can be fabricated and then it goes out to the market and succeeds in the competition in the market. So, the sooner we bring a product, the more likely it will succeed; therefore, turnaround time is very important, especially in the semiconductor industry.

Now, when we do abstraction, what benefits do we get? When we do abstraction at the higher level, the details in the design are less. So, at the higher level, abstraction is greater, and therefore, a large number of solutions can be analyzed in less time. So, when we remove details of a design, then what happens is that only some parts of the design that are actually important, maybe the functionality or other things, are important, and only a few implementation details are available at the top level. So, when we keep only a few details of a design, it allows us to explore the design space more easily in terms of runtime by the tools and a human designer effort. Why? Because when the details are fewer, a human designer can look into more details. And when there are fewer entities, the designer can focus on those entities and optimize them manually or give hints to the tool and get a better QoR.

As a result, optimization at a higher level of abstraction is expected to be better. So if the scope of optimization is very high at the top level because there are fewer details, we can try out many options and then choose the best of them. Therefore, the scope of optimization is very high at the system level design or at the idea to the RTL flow, which is the first part of the flow that we discussed. And the turnaround time will also be low because the details are less. We need to add or change fewer details in the design; therefore, turnaround time can be quicker.

In the RTL to GDS flow, what happens is that as the design flow progresses, more details are added, and therefore, making changes becomes difficult. The scope of the optimization decreases because we can try out fewer options. And when we go into the last part, which is the fabrication, there is no optimization. Only if some corrections are required, those are made, and those are very costly. Now, to illustrate the various levels of abstraction, let us take an example. Now, consider that the functionality is represented in two ways.

First, in terms of logic formula, this is one level of abstraction in which we are representing a function in terms of logic formula (Representation “A”). For example,

F = (A + B)'

Now, the same function can be represented in a layout also, in terms of standard cells and a NOR gate or a standard cell, which is delivering a NOR function, and we are making a layout and placing it and making a connection out of it (Representation “B”). So, these are two different levels of abstraction. Now, let us understand which of the above representations has greater abstraction.

Since in the logic formula, there is less detail; it only defines the functionality. The details about how it is implemented in terms of layout and other things are missing, the level of abstraction is higher for the first representation. So, in the first representation, the abstraction is greater, and a smaller turnaround time will be required in the first representation. Why? Because if you want to make a change in the implementation, suppose we want to represent F = (A + B)′ as F = A' . B'.

This kind of transformation can easily be done at the logic formula level. However, what if we want to do a similar transformation in the layout? Then we might need to change the NOR gate to, say, an AND gate, and then we need to have inverters as well, and then we need to place these gates and make the connections on the layout.

So, the turnaround time required to make changes in the layout representation or "B" representation will be much higher than in the "A" representation. Now, the question comes in which case the accuracy in evaluating different options (FoMs) will be higher. So, we understand that in the logic formula, the details are missing, meaning that we do not know the details of how the NOR gate is implemented. So, we cannot compute the delay, area, and other things. However, if the layout is there, then we can estimate the area, delay, and other figures of merit much more accurately. So, greater accuracy in evaluation will be in the "B" representation. So, what we see is that as the design moves from a higher abstraction level to a lower abstraction level, the details are added. As such, when details are added, the scope of optimization reduces, but the estimate of the figures of merit becomes more accurate.

Now, let us look at pre-RTL methodologies. Pre-RTL methodologies are the design tasks that we undertake before getting an RTL. In this part of the VLSI design flow, we decide the various components required for a system.

These components may be hardware or software, and we also decide how these components interact with each other. Therefore, this design step or pre-RTL methodologies is also known as the system-level design. In this, when we have an idea, and we want to get a product out of it. What we do is that initially, whenever there is an

idea, the first thing is that we evaluate whether that idea is worthwhile or not. So, we check for market requirements, analyze financial viability, and look into the technical feasibility. If all these things are positive, then we move to the next step, which is creating a specification for the product.

So when we define a specification, we will decide on the features that we need for the product and the PPA or the figures of merit that are relevant to the product. For example, at what clock frequency should it work, or what power dissipation will be allowed for the chip, cost, and other kinds of figures of merit? Then we also decide the schedule of making this product and the time to market. So, the time to market is the time it takes for an idea to get implemented and the product coming out to the market. We want to keep it as small as possible.

Once we have the specifications and the features that need to be implemented for a product, the next step is hardware-software partitioning. So, in this hardware-software partitioning step, we identify various components that will be required for the system. So, at the broad level, these components can be either hardware or software. Now, we need to decide which component needs to be implemented in software and which component needs to be implemented in hardware, and this step is known as hardware-software partitioning. Once we have decided on the hardware portion of the specification that needs to be implemented, it goes through the IC design and manufacture phase in which we build the hardware.

Similarly, the software portion of the specification goes through the software development process; and the executables and the firmware, device, drivers, apps, etc are the final results of the software development. Once both hardware and software are available, there will be some components that already exist, which means we can directly reuse them at the system level. When all of them are ready, we do a system-level integration. We combine all these components together, we do some validation and testing, and at the end of this, the product comes out of it. So, this is a very top-level view of system-level design.

So now we will look into a few of these steps or system-level design steps in more detail. So, the system-level design varies depending on the type of products we are designing or the type of functionality we are implementing. For example, whether it is a processor or a signal processing chip or maybe some controller or other kind of design, the design task undertaken in system-level design will vary. However, let us look at a few design tasks common to all the system-level designs or widely used.

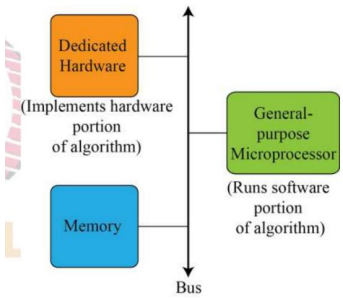
So, let us look into the first part, hardware-software partitioning, which is one of the integral parts of system-level design. So what is hardware-software partitioning, and why do we want to do it? So, the motivation for doing hardware-software partitioning is to

exploit the merit of both hardware and software. So now hardware has got some merit, it has also got some demerit, and software has got some merit and also demerit. Now, when we combine hardware and software and get a system out of it, we want the advantages of hardware to be utilized in the system, and advantages of the software also to be utilized, and as a result, we get a product of excellent quality. So, what are the advantages of hardware, and what are the advantages of software? Let us understand that first. The good thing about hardware is that it can deliver a very high performance, meaning it can deliver a very high speed.

It can give you a very high speed because hardware can be made such that there are parallel circuits, and all of them can work concurrently, and as such, a given task can be completed much more quickly. If we look into the software, it is running on a processor and assuming that the processor is only running in a sequential manner taking one instruction, doing the processing, and giving out the result. So, there will be a kind of sequential operation for the software, but for hardware, a lot of parallelization can be done, and therefore, the biggest merit of hardware is that it can deliver a very high performance. Now, what are the good things about the software? The good thing about software is that the development of the software is much easier because it is less complicated than making hardware or designing the hardware. Also, the risk involved in software is much lesser because if we find any error in the software, we can simply debug it and make changes in the code, recompile it, and we are done.

But if there is a problem with the hardware, we may need to respin it. We may be required to redesign the hardware, we may need to change the mask and other things; and therefore, it may be costly and time-consuming. Another good thing about software is that a lot of customization can be done. You can make a code that can do many things based on the requirements, and it is easy to customize in the case of software. However, for hardware, you need hardware changes and other things which are more difficult to implement.

The development time of the software is much less compared to the hardware because hardware design is a more complex task, as we will be seeing in this course in the subsequent lectures. Now, to understand how we do hardware-software partitioning, let us understand or take a look at a typical system in which both hardware and software coexist. This figure shows such a kind of system. In this, we have a general-purpose microprocessor over which the software part is run. So, one instruction is running, and the processor gives the result, then it is utilized again, and so on.



So, the software is running sequentially on a general-purpose microprocessor. Then, there is dedicated hardware that performs some tasks. This hardware that we are talking about that we can implement in a system is also known as a hardware accelerator, which can give the result very fast. In this figure, we are showing only one dedicated hardware, but there can be multiple dedicated hardware, and all of them may be working in parallel. As a result, the whole system will be giving the result very fast.

There can be memory, and all these things can interact through or share information through some bus. So here, a simple bus is shown, but these connections can be very complicated in a chip or system realized on a chip. So, the figure that I have shown here is just for illustrative purposes. I have just shown you a kind of system where hardware and software are both working together. So, if the hardware usually runs as parallel circuits, we can have very good PPA, and that is why these are known as hardware accelerators, and they can be implemented in full custom IC, ASIC, or FPGA.

This dedicated hardware can be implemented in various design styles, and then the software usually runs sequentially on the general purpose processor. Now, the hardware-software partitioning step tries to map the functionality that should go into this software part running on a general-purpose microprocessor and the functionality that should go into this hardware part or multiple such hardware or different kinds of hardware which will be combined onto the system. Now, let us look into an example of how partitioning can be done. Let's consider a video compression algorithm. We can divide this algorithm into two parts.

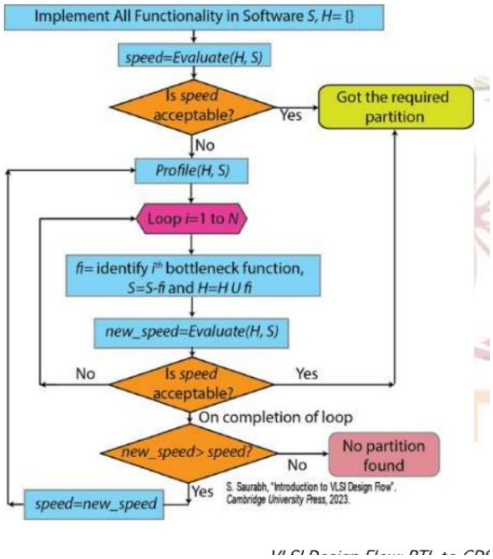
The first part is the one which is computing discrete cosine transform. So, assume that for the time being that this computation is done many times, and this is the bottleneck for the algorithm. It takes a lot of time to compute a discrete cosine transform. Then, other computations, for example, frame handling, etc., are also being done. But the point is that here we are assuming for the time being that DCT computation part of the algorithm is bottleneck, meaning that a lot of runtime, say out of 100 percent runtime 80 percent, is going into computing DCT.

Now, in this case, how do we do a hardware-software partition? What we can do is that the DCT part can be computed separately on a hardware or a hardware accelerator. It can be computed using parallel circuits, and that circuit can be optimized so that the performance can be 1000 times faster than software doing a similar kind of computation on a general-purpose computer or processor. It can be several orders of magnitude faster

and more energy efficient. We can optimize other QoR measures, for example, energy efficiency or power dissipation required for the computation of DCT and other things. Those things can also be optimized in the hardware that we design. The other part is frame handling and other computations that can be delegated to the software part, and that software will be running on a general-purpose processor in the same system. So, software running on the general-purpose microprocessor gives the benefit that it provides more flexibility.

For example, later on, a frame handling algorithm and other things need to be changed. Those can be changed just by the software modifications. No change in hardware will be required. So this is just an example that, given an algorithm, how can we partition into the hardware part and the software part? We typically choose the critical part, which is a bottleneck at a system level to implement in hardware and the part where we require a lot of flexibility in the algorithm or where the change can be there that we can implement in the software. Now, hardware-software partitioning can be done in many ways.

For illustrative purposes, let us look into one way in which hardware-software partitioning can be done to illustrate the challenges and what decisions need to be made during the hardware-software partitioning step. So, let us assume that the objective of hardware-software partitioning is to find a minimum set of functions that need to be implemented in hardware to achieve the desired performance. So, we say that initially, an algorithm is given to us, and that algorithm is implemented entirely in software. Now, out of all the functions implemented in software, we want to partition it or move a subset or set of functions to hardware, and what that subset should be, we need to find out. So, in this figure, I am showing you a flow chart for how hardware-software partitioning can be done.



Initially, we are representing here two sets: one is S, and the other is H. So the set S contains all the functions implemented in software, and H contains all the functions implemented in hardware. So initially, all the functions of the algorithm are in S, and H is an empty set, and everything is implemented in the software. Then we are also given an

acceptable performance P so the performance should be better than P; it should not be worse than P. So, by hardware-software partitioning, we want to attain that the performance should come above a given threshold P. Then there are parameters of the algorithm N, which decides the maximum number of functions to be moved to hardware in each iteration.

What is the use of N? We will see later. So, the input is the set of functions that need to be implemented in a given algorithm, and all are initially in the software. In the end, the output that we want from this software-hardware partitioning is a set of function H that needs to be implemented in hardware. Initially, H is empty, but as this algorithm progresses, the H will be populated, or that set will get filled, and it will be filled with a minimal set of functions such that the performance becomes above the given threshold. Now let us see how to achieve this in hardware-software partition. So when we do hardware-software partitioning, we need to measure the performance. So, in this pseudo code or this flow chart, I am showing the task of measuring the performance using the function Evaluate(H, S).

So, this algorithm first measures the speed or current performance of the partition. In the initial stage, H is empty, while S contains all the functions of the given algorithm. Now, if the speed is already acceptable, i.e., greater than P, then we need not do any partitioning. In that case, everything can be implemented in the software part. So, we have the required partition, and nothing needs to be done. If the performance is below the threshold P, then what we do is a step which is known as Profile(H, S).

What this step is doing is measuring the frequency or duration of each function call. Suppose we had all the functions of the algorithm implemented in software. Now, for each of the functions, we do a step known as profiling, and profiling is running the given test cases using a profiler. A profiler is a tool that measures how much run time each function takes in a given executable**.** So, initially, everything is in the software. We do a profiling of the code to find out which functions are taking more time.

For example, there are four functions: f1, f2, f3, and f4. So, the profiler will give information that f1 is taking, say, 10 percent of the time, f2 is taking, say, 70 percent of the time, f3 is taking, say, 10 percent of the time, and f4 is also taking 10 percent of the time. This information will be given by the profiler. Now, using the profiler information, we can identify the bottleneck function. So clearly, in this case, f2 is the bottleneck function. So in each iteration, what we do is that we identify i-th most severe bottleneck function fi.

So, in this case, the first bottleneck function is f2. So once we identify the i-th most severe bottleneck function, what we do is that we assume that fi is implemented in hardware. So now, in H, we will add an element which is f2. So f2 will be added to the set H, that is what it is doing: $H = H \cup fi$, that is, we are adding the element f2 to H. And what we are doing simultaneously: $S = S - fi$, meaning that we are removing f2 from the set of functions in S. So, it indicates that this function f2 has moved to hardware rather than software.

Once we do that, then what we do is that we evaluate the new speed. Ideally, we want the new speed to be better than the earlier speed because hardware can probably do things faster compared to software, and therefore, the new speed will be greater. Now, if this performance is improving to such an extent that it crosses the threshold P, we get the required partition. If not, then the next function will be taken. In this case, f1, f3, and f4 all have equal percentages. So in the next one, maybe, say, f1 will be picked; anyone can be picked out of them, so the first appearing function out of the functions having an equal percentage will be taken. Now we move f1 also into H and then measure. Now, if the speed is acceptable, we are done. If not, we will continue moving a function from the software to the hardware side.

So, we measure the performance and check whether target performance P is met. So, it moves a maximum of N most critical bottleneck functions to the hardware. So it goes on doing it N times. Now, what happens is that in doing this N times, either we get the required partition, or the performance of the system improves to such an extent that it crosses the threshold P, or it may not deliver sufficient gain, and then we need to iterate it. Now, this iteration terminates when we are able to achieve the target P, which is one criterion, and then this algorithm will terminate. Another terminating criterion is that there is no improvement even after moving N functions from software to hardware.

When we are not getting an improvement, we say that no partition can be found. Why can we not get an improvement even when we have moved a function from software to hardware? There can be many reasons. One of the primary reasons can be communication. When we implement everything in software, everything goes to software. But when there is a partitioning between software and hardware, then there will be time taken in the communication between software and hardware. There will also be dependency on the data, and it may be that hardware is waiting for the data, and therefore, it cannot produce the result even though it is ready.

Therefore, because of the data dependency, there can be less gain, or even if we gain by implementing in hardware, that can be offset by the extra time taken in communication. Therefore, even if we move some function out from the software to the hardware, we may not get an expected gain, and in that case, we cannot find a suitable partition. This is a very simplistic method of hardware-software partitioning. This is just for the illustrative purpose. It takes a greedy approach to settle to some minima or get to a case in which we are able to meet the performance and incur the cost of hardware to the minimum. We are assuming here that the hardware is costly. Therefore, we do not want to move everything to the hardware. We want to move a minimum set of functions from the software to the hardware; that is what this algorithm is assuming.

What are the challenges in this kind of implementation or hardware-software partitioning in general? The first challenge is the performance estimation. When we move the function f from software to hardware, we evaluate the performance to see whether it is below the threshold or not. Now, the problem is that we do not have the hardware yet. We have just assumed that the software function has moved to hardware, but the hardware is still non-existent; the hardware has to be made, it has to be designed, and it has to be fabricated. Then only we can get a fair idea or get the exact numbers by how much performance is improved.

So, the non-existence of hardware will be the primary concern in hardware-software parts. Now, how do we try to tackle this problem? We tackle this problem by implementing the hardware either in, say, FPGA, which can be designed very quickly because the hardware is already there; we just need to program it, and therefore, we assume that the things have moved to FPGA, and based on that, we can make some estimate that if that function is moved from software to hardware how much gain do we expect. Even for ASIC, we can extrapolate that number to something and then check if we implement that function in ASIC whether we will get a sufficient gain or not. The second approach that can be taken is that given the function f1 is implemented in hardware; we take it through a quick design flow. For example, we do a behavior synthesis or high-level synthesis, implement it, do a placement, routing, and other things at a very crude level, and estimate or get a number out of it.

That can be another way to deal with this problem. The second challenge is regarding the verification of hardware and software. Now, earlier everything was implemented in software. Now we are saying something is implemented in hardware, something in software. Now, we also need to ensure that once these two combine together, they still give the required functionality, and we need to do that kind of verification. And that kind of verification is a difficult thing because the hardware is still non-existent.

So various approaches are taken to deal with this problem. For example, hardware-software co-simulation in which we assume some model of the hardware or timing model for the hardware, and using that timing model of the hardware and the existing software, we try to simulate it and get a fair idea of whether the system will work even when these two things are implemented together or combined.

So, this is the reference to which you can refer for more information on this topic, and this brings us to the end of this lecture. In this lecture, we have looked into an important concept of abstraction, and then we looked into the idea to RTL flow or the system-level design flow. In particular, we looked into hardware-software partitioning. In the next lecture, we will look into another aspect of system-level design, which is behavior-level synthesis. Thank you very much.