VLSI Design Flow: RTL to GDS

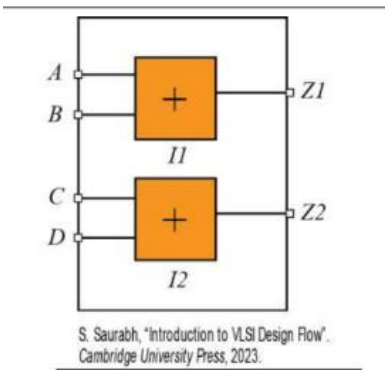
Dr. Sneh Saurabh
Department of Electronics and Communication Engineering
IIIT-Delhi

Lecture 11
Hardware Modeling: Introduction to Verilog-I

Hello everybody, welcome to the course VLSI Design Flow: RTL to GDS. This is lecture number 9. In the earlier lectures, we looked into the overview of VLSI design flow. From this lecture onwards, we will go into each design task in detail. For the RTL to GDS design flow, the RTL is considered the starting point, or when we get an RTL, we say that we have started the design flow. In the earlier lectures, we have seen that an RTL is modeled using hardware description languages.

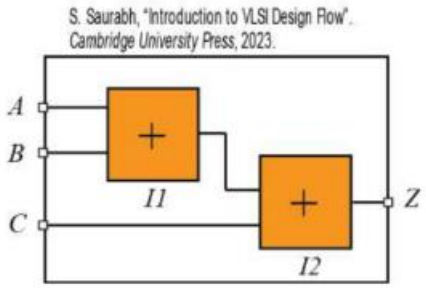
Now, among many hardware description languages, Verilog is the most popular. So, in this lecture, we will look into some basic concepts related to Verilog. Specifically, in this lecture, we will first look into the features of hardware description languages. Then, we will look into various constructs of Verilog or basic syntax and semantics of Verilog. Now, let us first understand some distinct features of hardware description languages compared to other high-level programming languages.

So, several features of hardware description languages are the same as other high-level programming languages such as C and C++. However, the hardware description languages such as Verilog and VHDL have distinct or some special features, and these features are required because we want to model hardware, and hardware has some features that need to be captured or modeled in a realistic way using these languages. So, what are these features? The first feature hardware can offer is concurrency, meaning that the computation can be done in parallel. Let us take an example. Suppose this is a simple circuit. It has two adders.



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

The first one adds A + B while the other adds C + D. Now, these two hardware components can work in parallel, meaning that while A or B is changing, the first adder will produce a result, and Z1 will be produced. If, say, one of the inputs out of C and D is changing, then at the same time, the other adder will also produce the correct outputs, parallel or concurrently. This is something distinct to hardware that is not typically there in other high-level programming languages because, in other high-level programming languages, the model is that we have a CPU, and we have instructions, and instructions go to the CPU, and they are executed in a sequential manner. But in hardware, we can do computation concurrently, and the hardware description languages must have syntax and semantics to distinguish between parallel and sequential operations. Note that in hardware, a sequential operation can also be there.



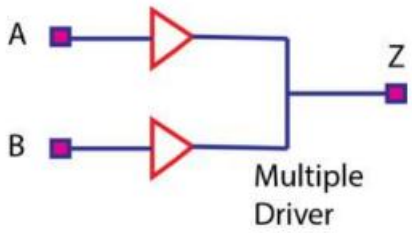
For example, consider this circuit. In this, there are two adders. The first one is adding A + B. The result is produced here as A + B, and the second adder is adding (A + B) +C, and the result A + B + C is produced by the circuit. But this second adder can do the computation only when the first adder has done the computation.

So, for example, if A changes, then the change should be first reflected by the first adder, and once the first adder has done or updated its computation, then only the second adder can produce the result at the final output, i.e., A + B + C or the correct value or the updated value of A + B + C will be captured at the output. So, the hardware can be organized in a sequential manner, and it can also be organized in a parallel manner. So, the most important feature of a hardware description language is that it should have syntax and semantics to distinguish parallel and sequential operations.

The second thing that is required for hardware modeling is the notion of time. So, we need to describe the behavior of the system with respect to time. There may be delays in the signals and those kinds of things we need to model or capture in the hardware description languages. And, of course, the notion of concurrency and sequential operations also requires some notion of time. And for modeling hardware, we also need to create waveforms, especially for synchronous circuits. In a synchronous circuit, there is a clock, and we want to model our circuit such that some operation happens at each

edge of a clock. We need to have something that allows us to model this clock in an easy manner and captures the operations that are controlled by the clock in a realistic manner.

So, these kinds of features should be there in the hardware description language. Then, in hardware description language, since it is capturing hardware and in hardware, we are modeling the electronic circuits, we also need to consider the strength of the signals. For example, if we have, say, two drivers for this signal Z, one driver is A, and the other is B. Now, when, say, A and B are producing output, there can be, in some situations, a conflicting value. Now, if one is producing, for example, 0, and the other is producing 1.



Then, the strength of the signal may decide its final value, Z. For example, if 0 is weak and 1 is strong, Z will probably get a value of 1. So, there should be some way of resolution of the conflict when there is a conflict in the operation of values at a signal. The fourth thing that is required for a hardware description language is that it should have bit true data types, meaning that when we model our hardware, there can be, say, a data bus, and this data bus can contain, say, 32 signals, or 32 individual signals. So sometimes we want to do the operation with the entire bus. For example, if there is one bus, the value of this bus is written from one buffer to another, then we can just say that A = B in the sense that A is a bus and B is also a bus. Or, in some cases, we want to do an operation with each individual bit. For example, we want to mask some bit out of this bus, or we want to take a group of them and do some operations. So, this behavior of buses, the behavior of individual bits, also needs to be modeled. So, in the sense that we need various levels of abstraction of the signal, maybe one abstraction level is a bus, and the other abstraction level is a bit of that bus. So, the hardware description language should support various kinds of abstraction levels of signals. Now, in the early 1970s, there was a lot of research regarding hardware description languages, and many hardware description languages were proposed.

Finally, two hardware description languages that became popular were Verilog and VHDL. So, currently, these are the most popular languages which are used for modeling hardware. So, let us look into them briefly. Verilog was created in 1983-84 at Gateway Design Automation and the word Verilog is derived from two words, 'verification' and 'logic'. So 'Veri' is from the word 'verification' and 'log' from 'logic'.

Now, this Verilog word signifies that Verilog was initially started for the purpose of verification using simulation. The main purpose of this very invention of Verilog was to do very fast simulations. And in the early 1980s, there was a simulator, Verilog XL, which could do a very fast simulation at that time compared to other existing technologies. And this was the first reason why Verilog became popular.

Then, a tool called Design Compiler from Synopsys could actually do logic synthesis using the Verilog language. This was developed, and the concept of using hardware description language to model a circuit and then automatically transfer that hardware description model or HDL model to a logic or netlist. This concept was new at that time, and it was efficiently being done by this design compiler tool, and this made it further popular. Another reason why Verilog became popular was because of its simplicity. Simplicity in terms of syntax and semantics, as we will see in today's lecture.

So, the syntax of Verilog is very much similar to C, and this resonates with the engineers who already know C, and that is why Verilog became very popular. It was first standardized by IEEE in 1995 and then again in 2001 with more updates. Later on, for verification purposes, more features were required, and to support that features from other languages were borrowed. A language known as System Verilog, which is a superset of Verilog and with added functionality for design verification, was proposed and standardized in 2009. Currently, Verilog and System Verilog are the most popular hardware description languages for design and verification, and according to one estimate, almost 80% of the chip designing or digital circuit designing is done using Verilog or System Verilog. Another popular language is VHDL. So, VHDL is a short form for VHSIC hardware description language.

Now, VHSIC stands for "very high-speed integrated circuit". So, this language VHDL was initially started as a documenting language for integrated circuits in the 1980s, and it was very popular at that time. Since it was more for documenting purposes, it is a very verbose language, and there is strict type-checking in VHDL language. VHDL was initially standardized by IEEE in 1987 and recently in 2019. So, VHDL is also a popular language for modeling hardware. Now, let us go into the various features of Verilog languages.

So, before going into the features of Verilog language, let me show one dialogue from the novel "Through the Looking Glass" by Lewis Carroll. So, there is a dialogue between Humpty Dumpty and Alice. So, Humpty Dumpty says, 'When I use a word,' Humpty Dumpty said in a rather scornful tone, 'it means just what I choose it to mean - neither more nor less. Then Alice says, 'The question is whether you can make words mean different things.' So, Humpty Dumpty replies, 'The question is which is to be the master - that's all.'

So, this dialogue aptly describes the role of words in conveying ideas, and it also describes the challenges and opportunities we have in defining words and what they mean. With this idea now, let us look into how the words in Verilog are chosen to describe hardware. So, the word should be chosen such that it can describe the hardware very realistically and in such a way that it does not lead to confusion, meaning that it has to describe the hardware and the features of the hardware unambiguously. So, in today's lecture, we will look into the features, syntax, and semantics of Verilog. We will be looking at how these features, these Verilog constructs or Verilog features, are used in simulations and synthesis in the next couple of lectures. So, in this lecture, we will introduce the constructs of the Verilog language, and then what is the interpretation of these constructs we will look at in subsequent lectures.

So, a Verilog language considers an RTL file or a given file as a stream of lexical tokens. If there is a file, it contains words, characters, etc. So, the Verilog language considers it as a stream of tokens, and the rules for the lexical tokens are similar to the C programming language, and it is case sensitive. This is another point we should note that Verilog language is case sensitive. Now, what can be the tokens? So, in a given file, what could be the tokens? So, tokens can be of various types: they can be white spaces, comments, keywords, operators, identifiers, numbers, and strings. Now, we will be looking at each of these tokens in more detail to understand what is allowed by the language and what the interpretation is.

So, a white space can contain the characters of spaces. For example, I write a word, say white space. So, between these two words, there is a space, and this is a white space. So, white space can contain characters or spaces. It can also have tabs, new lines, and form feeds. And what is the role of white space? It is used as a separator for tokens.

So, we break the Verilog file into various tokens with the help of white spaces. What are the comments in a Verilog file? So, anything starting with a double slash // or anything that is enclosed within /* and */ are considered as comments. Nested comments are not allowed. This is a comment. If we need a one-line comment, we just put it after // and add whatever the comment is. And if we want a block of comments, then we can start with /*, give our comment, and then end it with */. So, this is a block.

What are the keywords in Verilog? Keywords are basically reserved words for the Verilog language, and these reserved words are always in lowercase as per Verilog standard or Verilog syntax. For example, the keywords are module, input, output, initial, begin, end, always, endmodule, etc. We will be looking at those keywords as we proceed in the lecture.

Then there are operators. Operators are predefined sequences of one, two, or three characters used in an expression. For example, !, +, -, &&, ==, !==, etc. So, we will be

looking into these operators in more detail later in this lecture. Then, we should understand what identifiers are in Verilog. Identifiers are unique names given to an object so that it can be referred to in the Verilog code. We want to give a unique name to modules, ports, nets, registers, and functions.

So, whatever RTL entities are there or design entities are there, we want to give a name to them so that we can refer to them later in the code for various purposes. What are the rules for the identifiers? The first rule is that it should start with an alphabet, a small letter (a to z), or a capital letter (A to Z), or it can start with an underscore (_). Subsequent characters can be numbers (0-9) or $ in addition to alphabets and underscore. For example, I name a module as a Mymodule_top.

This is an identifier. So, we can reuse this module in our code wherever required to refer to it. Similarly, Register_123 is an identifier. These identifiers are case sensitive, meaning that if I write Net_1, this is a separate identifier, and if you write net_1, this is distinct from the first one. So, the identifiers are case sensitive, and the maximum size of the identifier allowed by the language is 1024, meaning that the tool that supports Verilog language must support at least 1024 characters in an identifier. The tools can support more than that, but at least they must support 1024.

So, it means that a designer writing a Verilog code can use identifiers of size less than 1024 because we are not sure whether all the tools used in the design flow will support sizes more than 1024. So, to be on the safe side, we should always restrict our identifiers to less than 1024 characters. Now, there can be something known as escaped identifiers. Any character can be used in an identifier by escaping the identifier. It means that in addition to the characters that are allowed in an identifier, if we want to add/use other characters, we can use it. But in that case, we need to follow a rule that if we want to use special characters in our identifier, the identifier should be preceded with a \ and ended with a white space, meaning a space, tab, etc. So, these are known as escaped identifiers. For example, suppose I wanted to use the characters (,), +, *.

These characters are not allowed in an identifier, but if we want to use it to add more readability to my code, I should start the identifier with a backslash and end with a space. For example, \net_(a+b)*c . The space is not visible here. There is a space after this. So, the tool will treat the name as net_(a+b)*c. It will remove the backslash and also the ending white space. This is just for the understanding of the tool that an escaped identifier is starting and ending. So, whatever is between them, the tool will treat it as an identifier name as such without the preceding backslash and the ending space.

Then what are the numbers in Verilog? They can be integers or real numbers. When we write numbers in our code, we should write in a format that is convenient to us, and also it is readable for others to read the code. Note that once we write the code, it is not only

for us. Our code that we will be writing may be used by some other group or will be debugged by some other group.

So whenever we write code, we should look into the readability part, meaning whether our code is understandable to the other person who will be using the code. So when we write code, we should use a convenient and readable representation, but tools internally will convert it into a sequence of bits. So when we write, we can use decimal, hexadecimal, octal, binary notation, or any format we can use for integers, and then the tool internally will convert it into a sequence of bits. We can write an integer either in the traditional format like, say, 169, -123, etc. This is allowed. Or if we choose to define the base, size, and value, then we can use this format: -<size>'<base><value>. What is this format? The first character is -. It is for the negative sign and is optional. Then, we have the size, which defines the number of bits; the default is 32, and the base defines the base of the number. It can be binary, for which we use the character b or B, or it can be octal, which can be defined by o or O; for decimal, we can use d or D; and for hexadecimal, we can use h or H.

So, we have to define this base using these characters only. And the final thing is the value of the number. Let us take a few examples. If we write 1, the internal representation will be a series of 0s, and then the final bit will be 1 (0000... 0001). So internally, since we have not specified the size, it will be treated as 32-bit, and that is why all these zeros will be added. Now, if we write 1'b1, there is no negative sign, so it is a positive number. Then, 1 defines the size of this number, so it is a 1-bit number, and b defines that it is a binary number. It is not octal or decimal. 1 defines the number (value), so 1'b1 is a Boolean number 1. Then the third number is 8'ha1, 8 means that it is 8 bits, h means that it is a hexadecimal number, and a1 is the number. So, a in hexadecimal is 1010 in binary, and 1 is 0001, so this will be the internal representation 1010 0001.

Now, for 6 'o71, the size is 6 bit, it is an octal number, for octal 7 is equal to 111 in binary, and 1 is 001, so this is the internal representation of the number 111 001. Now, there can be complicated scenarios for writing the integers, so there are some rules. For hexadecimal, octal, and binary constants, x/X represents the unknown or don’t care value, and z/Z/? represents the high impedance value. Now, when a high impedance is don’t care, it is better to write a ? so it distinguishes that it is a don’t care and not a high impedance or it is high impedance, which should be treated as don't care. For example, this is the number 8'b100z00?1. Here ? is the same as z; internally, it will be treated the same as z or high impedance value.

Now, when the size is smaller than the value, the leftmost bits from the value are truncated. It means that suppose in 6'h88, size 6 is smaller than the value it has. The value here for hexadecimal 88 is 1000 in binary for the first 8 and 1000 for the next 8. But the given size is only 6 bits and not 8 bits. But it has got 8 bits, so what the tool will do is it

will simply truncate the leftmost bits, so the 2 leftmost bits will be truncated, and the internal representation will be 001000. Now, another situation can be when the size is greater than the value, then left most bits are filled with either 0 if left most bit value is 0 or 1, or it will be filled with Z if the leftmost bit is Z or it will be filled with X if the left most bit is X. So if the size is greater than the value, it needs to add some number. So, these rules define how to add these numbers. So, for example, we take a number that should be represented in 8 bits, and it should be in binary, and the value is given as 11. So, in that case, since the leftmost bit is 1, the other leftmost bits that are not specified are taken as 0. In another case, if we say 8'bz1, z1 is only 2 bits. So, what are the other bits that will be padded or will be added before this number on the left-hand side? All will be Z because the leftmost bit was Z in this case, i.e., zzzz zzz1. Also, we can add an underscore in the middle of the number to enhance the readability. For example, if your number has a very large number of bits, then we can break it into smaller parts by adding an underscore.

For example, for an 8-bit number, we can write it as two 4-bit numbers separated by an underscore, and this underscore will be ignored by the tool and will be treated as a number without an underscore. This underscore is just for readability. Now, negative numbers are internally represented in 2's complement. For example, if the number is, say, -8'd6, and we want to represent it in 2's complement. So first, we have to take the binary equivalent of the number 6, which is 110. The size is 8 bits, so we have to pad the numbers with bits to make it 8 bits. Then, we take the 1's complement, that is, 11111001, and then add 1 to it to get the 2's complement. It becomes 0101_1111. So this is the internal representation of -8,d6.

In Verilog, the real numbers can be represented in decimal form like we usually do. For example, we can write the real number as 3.14159, or we can choose the scientific notation like 2.99E8, where we have mantissa and exponent part. Internally, real numbers are represented in IEEE standard for double precision floating point numbers. Verilog also supports strings, and a string is a sequence of characters enclosed by double quotes and contained on a single line.

For example, this is a string "Hello". Now, internally, the string will be represented by its corresponding value, with each character being represented by an 8-bit ASCII equivalent of it. For example, the word or the string Hello will be represented internally as the ASCII equivalent of H, will represent H in 8 bits, then similarly, e will be represented as the ASCII equivalent of e, and so on. So, internally, the string will be represented by the ASCII representations of each of the characters in that string. Now let us look at what are the data values and data types in Verilog.

Verilog supports four-valued data. This is an important thing because typically, we associate digital systems with binary values, i.e., 0 and 1, but Verilog supports four-

valued logic or four-valued data, and what are they? So, the first one is 0, which is equivalent to Boolean 0 or false logic. Then, 1 is equivalent to a logic true or Boolean 1, and the third is x, which is an unknown value, and z, which is a high impedance state. So, these are four values that are supported for a signal or for data. And what are the data types in Verilog? So there are two primary types: the first one is net, and the second type is variable. Now let us look into what is a net. So, the net represents structural connections. Whenever we say net, it is a kind of wire, and it cannot store value. It has nothing to store value. It only connects entities in the design. So, these nets are like wires, and the keywords to define a net are wire, supply0, supply1, wand, and wor.

So, these are the keywords. For example, we can say that

wire w1, w2; or

wire w3=1'b1;

and then we can say that 'gnd' is the name of a wire, 'vdd' is the name of a wire, and the type is supply0, used to model logic 0 or something which is connected directly to the ground and supply1 is for logic 1 or it is used to model VDD, respectively. The second type of data in Verilog is variable. Now, variables are the elements that store value in simulation, and these can be declared using the keyword reg. So, it stores the last assigned value until it is changed by another assignment.

So, reg stores the last value until its value gets changed. An example is

reg r1, r2;

So these are variables, and this reg works similarly to a variable that we have in other programming languages; it stores value, and until and unless we change, it keeps the old value. So we can assign reg in procedural blocks. We will see later what these procedural blocks are. So reg can model flip-flops, latches, and also combinational elements. Now, this is important. So many times, new programmers in Verilog consider reg similar to register, and they think that it has something to do with memory or flip-flops. So, reg is a keyword that is there to define a variable. Finally, this variable will be synthesized to a flip-flop, combinational circuit, or latch based on how this reg is basically used inside the Verilog code.

It need not always be synthesized to flip-flop or memory element. It can also be synthesized into combinational elements. So, reg is not equivalent to a flip-flop in the hardware. That is what I want to point out because novice programmers make this kind of mistake. Now, Verilog also supports vectors and arrays. Now, nets and variables are 1 bit wide by default, and they are known as scalar.

Now, if the Verilog also supports vectors and arrays. We can declare a vector by preceding the declaration with a vector definition in the following format:

[⟨ left_range⟩ :⟨ right_range⟩]

So before the declaration or before the identifier, we can give this left range and right range separated by a colon in a square bracket, for example [7:0]. So, this will be defining a vector. So, the left range is the most significant bit, and the right range is the least significant bit. For example, we have:

wire [31:0]databus; or

reg [7:0]addressbus;

These are examples of vectors.

Now, in this vector, we can select a bit of a vector by specifying the address within the square bracket like

databus[4] = 1'b0;

or we can select a portion of a vector by specifying the range of MSB and LSB separated by a colon. For example, from this data bus, if we want to get the fourth bit, we just write the databus and then select within the square bracket the bit position databus[4]. Similarly, if you want to get part of a bus, then we can say

addressbus[3:0]

so it will give all the bits from bit positions 0, 1, 2, and 3. So, it will give the bits in that position in the address bus when we use this constraint.

Now, Verilog also supports arrays, and it can be used for grouping elements into multi-dimensional objects and so many times. For example, when we are trying to model memory, we can want to group together entities in some manner. To do that kind of grouping, we can use arrays. So, the difference between vectors and arrays in terms of syntax is that we specify their range after the identifier. For the vector, it was before the identifier. For arrays, it is after the identifier. For example, if we write

reg r[15:0];

then it is basically grouping together 1-bit register r and 16 of them starting from 0 to 15 in the group.

Similarly, we can have more than one dimension, so it is a two-dimensional array, which is defined here as size 10x10.

```
wire matrix[9:0][9:0];
```

So, this defines a matrix of size 10x10. So, we have seen a few constructs of the Verilog language in this lecture. We will look at more such constructs in the next lecture. Thank you very much.