VLSI Design Flow: RTL to GDS

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

Lecture 16
RTL Synthesis- Part II

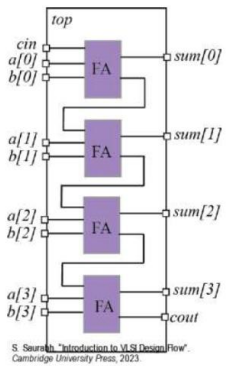
Hello everybody, welcome to the course VLSI Design Flow: RTL to GDS. This is the 13th lecture. In this lecture, we will be continuing with RTL synthesis. In the last lecture, we saw that the RTL synthesis task is broken down into smaller tasks such as parsing, elaboration, and then the translation of the RTL constructs to logic gates and then RTL-specific optimization. So, in this lecture, we will be continuing with looking at some more Verilog constructs and how they are translated to logic gates, and we will also look at some RTL-specific optimizations. So, let us start with looking at how for loop is synthesized in Verilog.

```
module top(a, b, cin, sum, cout);
      input [3:0]a, b; input cin;
      output [3:0]sum; output cout;
      reg [3:0]sum; reg cout; reg carry;
      reg [2:0]idx;

      always @(*) begin
         carry = cin;
         for(idx=0; idx < 4; idx=idx+1)
         begin
            {carry, sum[idx]} =
               a[idx] + b[idx] + carry;
         end
         cout = carry;
      end

endmodule
```

```
{carry, sum[0]}=a[0]+b[0]+cin;
{carry, sum[1]}=a[1]+b[1]+carry;
{carry, sum[2]}=a[2]+b[2]+carry;
{cout, sum[3]}=a[3]+b[3]+carry;
```



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

So, a for loop can model repeated logic structure, and whenever an RTL synthesis tool wants to synthesize a for loop, what it does is that it expands or unrolls the loop, and after unrolling whatever RTL constructs it gets, it instantiates logic gates corresponding to those. So, in each iteration, some RTL constructs will be formed after unrolling the loop, and then whatever the RTL construct is encountered in each iteration, logic gates corresponding to that RTL construct are instantiated in the design. So, now if the tool wants to unroll the for loop and instantiate the hardware, what it must know during synthesis that how many number of iterations are there. So, it needs to know the number of iterations at the time of synthesis.

So, the number or the loop count should be a constant, then only the RTL synthesis tool can synthesize it. If the loop count is a variable quantity, then the RTL synthesis tool cannot infer that how many times the loop needs to be unrolled and how much hardware it needs to put in the logic. Therefore, the for loop in which the number of iterations is fixed or known at the compile time or during the synthesis that kind of for loop are synthesizable in Verilog. So, let us take an example. Suppose there is a for loop, this for loop is there and in this there is a variable index (idx) and it varies from 0 to 3.

So, whenever index<4, then this loop is executed, and then when it reaches 4, then this loop is terminated. So, this for loop continues for 4 iterations. So, the number of iterations is fixed in this case, 4. If it was a variable, then this for loop would not have been synthesizable because during synthesis, the tool cannot infer that how much hardware to infer. So, for this for loop, what the tool will do is that it will unroll it.

So, it will start with, say, putting the index equal to 0. So, before the loop starts, carry is taking a value of cin and within this loop, what the statement is executed is carry, sum[idx], idx initial value is 0. So, the carry and sum[0] is being assigned a value a[0], a[idx] and idx is 0 for the first iteration + b[0] + carry. Now carry's initial value is cin. So, the first iteration, the unrolled form of this or the RTL construct that is formed after unrolling is this and then the index is increased to 1 and then if the index is increased to 1, then the RTL construct will be carry, sum[1] = a[1] + b[1] + carry.

Now, what is this carry? Carry from the previous iteration, and similarly, when the index value is 2, then it will be expanded, and for the third iteration also, it will be expanded. Now, this is the equivalent representation for this for loop. Now this RTL constructs that are there in this unrolled form of for loop, we can identify this thing as a full adder and similarly all these expression on the RHS is a representation of a full adder. So, in this case, the full adders will be instantiated inside the design. So, the implementation can look something like this.

So, there will be full adder, and full adders the sum[0] will be the output of it and what it will be taking input cin, and b[0] and a[0]. Now whatever the output that is generated that is this carry out (cout) and that goes as input in the second full adder. This is the second full adder, represents the second RTL constructs, the third one represents the third one and the fourth one corresponds to this. So, the output that is the carry out that is generated that goes as carry in the next stage and so on. So, this is how the for loop will be synthesized in Verilog.

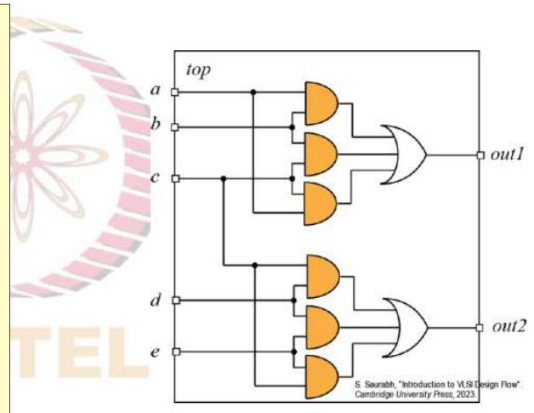
```
module top(a, b, c, d, e, out1, out2);
      input a, b, c, d, e;
      output out1, out2;
      reg out2;

      function MAJOR3;
            input A, B, C;
            begin
                  MAJOR3 = (A&B)|(B&C)|(C&A);
            end
      endfunction

      assign out1 = MAJOR3(a, b, c);

      always @(*) begin
            out2 = MAJOR3(c, d, e);
      end

endmodule
```



Now let us look into how functions are synthesized. So, functions synthesizes to combinational logic block with one output and that output can be scalar or vector depending on what the function is returning, whether it is returning a scalar value or a vector value. So, let us take an example. Suppose there was a function there was a function MAJOR3 which was returning value, MAJOR3 and on the RHS we have this expression. So, corresponding to this expression, combinational logic gates will be instantiated, and this MAJOR3 function is called twice, once here and once here.

So, out1 will be driven by whatever this is producing, and similarly, out2 will be driven by whatever is produced by this function. So, the synthesized design or netlist will look something like this. So, this logic structure corresponds to this RHS expression, and of course, similarly, this logic structure corresponds to this RHS. Now, what the inputs are to this logic structure are decided by what parameter or what values we are passing to the functions. So, this is how a function is synthesized.

Now, let us look at how various operators of Verilog are synthesized. So, in earlier lectures, we had seen that Verilog supports various kinds of operators, for example, logical operators, bitwise operators, arithmetic operators, and relational operators. Now among these operators the logical operators and bitwise operators, these are very easy to translate to the corresponding logic gates. So, what the RTL synthesis tool does is that it directly translates logical and bitwise operators to their corresponding logic gates and then optimize them subsequently. So, in the VLSI design flow after RTL synthesis we carry out logic optimization.

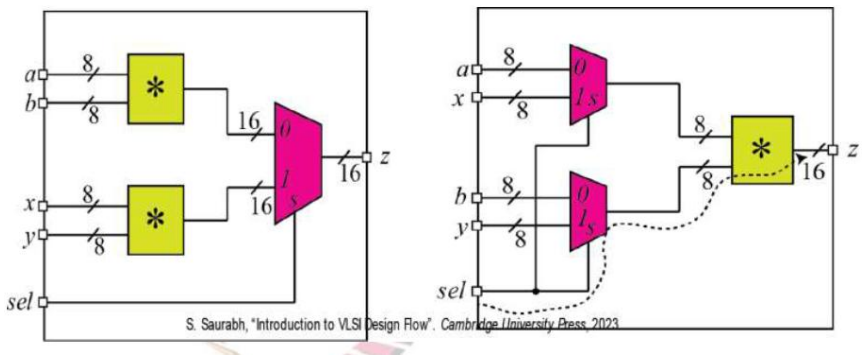
So, in logic optimization, these logical operators and the bitwise operators will be optimized much more efficiently. So, not much effort is spent during RTL synthesis to optimize these logical and bitwise operators. However, the arithmetic and relational operators those are resources or those are circuit elements which consume large area, and the implementation of them is also not trivial. So, a direct translation of these arithmetic

and relational operators may yield an unoptimized area or timing and other figures of merit. So, the RTL synthesis tool treats arithmetic and relational operators differently than logical and bitwise operators.

So, what the tool can do is that it first translate these arithmetic and relational operators to some internal representation or data structure may be a graph representation in which say operator and operand relationship is preserved. So, this internal representation will allow better or more efficient optimization. Subsequently what this RTL synthesis tool will do is that it will perform operator level optimizations. So, what are these operator-level optimizations? We will see in the subsequent slides, and then it will map those operators to a specific implementation? What this mapping means, we will see in the subsequent slides.

```
if (sel == 1'b0)
      z = a*b;
else
      z = x*y;
```

Assume: *a, b, x, y* are of 8 bits and *z* is of 16 bits.



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

So, first, let us look into an optimization or RTL-specific optimization, which is known as resource sharing. So, to understand it let us take an example and understand with the help of it. Suppose we are given a piece of code in which there is a select line which if takes a value of 0 then z = a*b or a multiplied by b is produced else z is written with x*y or x multiplied by y. And let us assume that a, b, x and y all are of 8 bits and z is of 16 bits. Now, given these things, if we directly translate this code snippet to logic gates or to computational elements, circuit elements, then it will look something like this for corresponding to a*b: we will have a multiplier, and the inputs will be a and b, there will be another multiplier in which the inputs will be x and y.

And then, depending on the value of select, either a*b is selected; if the select = 0, then a*b will be selected, or if select = 1, then x*y will be selected. So, this is a direct representation of the code snippet to a circuit implementation. However, note that in this case we are using 2 multipliers. So, 1 multiplier and 2 multiplier and multipliers are very costly in terms of area and and also power and other figures of merit. So, what we try to do is that in our implementation, if possible, we can reduce the number of multipliers. If we are able to reduce the number of multipliers, then probably we can save on area, and other figures of merit can also improve.

So, first let us look into that can we implement the same functionality with the help of only 1 multiplier rather than 2 multipliers. So, such an implementation is shown. So, what we are doing is that we use multiplexer while selecting the operand for this multiplier. So, whenever select = 0 then a line is selected and b line is selected and the output z becomes a*b. If select = 1 then what happens is that x line is selected and y line is selected and z gets the value of x*y.

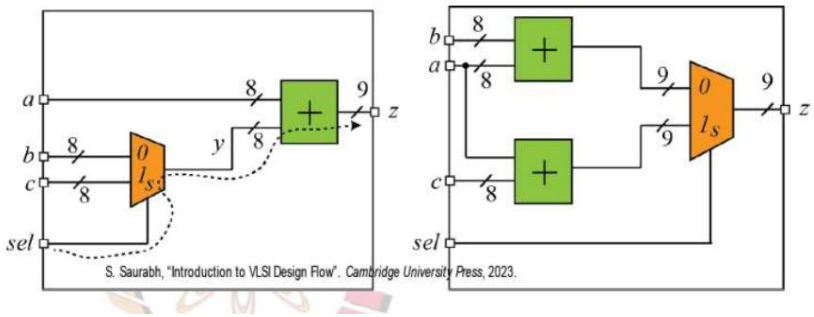
So, the selection happens first, and then we do the multiplication, and as a result, we are saving on the number of multipliers. Note that typically the multiplier will take much greater or significantly greater area than the multiplexer. So, increasing the number of multiplexers will not lead to much area increase, but the decrease in area because of reducing the number of multipliers will be more significant, and therefore, this circuit implementation, the second one, is expected to be more optimal in terms of area. However, we should note that there can be some problem in timing in the second implementation. In this circuit, there can be some problem in timing if you are not careful.

So, in the earlier case, in the first implementation from the select line to the z line, we do not have a multiplier. We have only the multiplexer, and therefore, the delay from the point select to z is not that much because only one multiplexer is on the path. However, if we look into the second implementation from the select line to the z line, we have not only the multiplexer but also the multiplier; either we see this path or this path in both of them, we will have a multiplier in its path. So, if this circuit was a part of a bigger circuit where the arrival time at select line was high meaning that this signal was already delayed and if we are adding a multiplier further in its path then what can happen is that this path from the select through the s line of the multiplexer and then finally, ending on the z pin through the multiplier that can become critical therefore, while doing this optimization the tool needs to be careful such that the path through the select line does not become critical and violate timing constraints. Now, let us look into another type of optimization which is known as speculation or it is also known as resource unsharing because it is kind of opposite of resource share.

```
if (sel == 1'b0)
      y = b;
else
      y = c;

z = a+y;
```

Assume: *a, b, c, y* are of 8 bits.



So, let us understand this optimization with the help of an example. So, suppose a part of code is given if select = 1'b0, then y = b. Else y = c, and subsequently, z can take a value,

either a + b or a + c depending on what the value of the select line. Now, let us assume that a, b, c, y all are of 8 bits. Now, if we make this assumption and translate this code directly into a circuit, it will look something like this.

So, we will have one adder corresponding to this expression. So, this adder will get one operand as a and the other operand as y and y can take a value either b or c depending on what the value of the select line is. Now, given this, let us assume that the path through the select line is critical, meaning that we want to reduce the delay of this path. Now, from this select line to the z line, we have a multiplexer and also an adder. Now, the adder is a circuit element that consumes or exhibits a large delay, and therefore, if we can remove this adder from the critical path, then the timing of the design can improve.

So, let us understand that how we can remove this adder from the critical path. So, one of the ways it can be done is shown in this figure. So, what we are doing here is that we are selecting the result based on the select line, and we are precomputing a + b, and using the other adder, we are computing a + c. And then, depending on the select value of the select line, we are either selecting a + b if sel = 0 and a + c if sel = 1. So, this code, this circuit, and this circuit are the same, but what improvement we get in the second implementation?

In the second implementation, what we have done is that we have removed the adder from the critical path, but what penalty did we incur? The penalty was that we now have two adders, and adders are costly circuit elements in terms of area and power dissipation. And therefore, in the second implementation we have actually traded of area and power dissipation to improve the timing of the critical path. So, it decreases the delay of the critical path by employing more resources. So, it performs addition irrespective of the sum being used. So, in this case what we are doing is that we are getting the result a + b and a + c without knowing that whether we need a + b or a + c.

And therefore, this is an example of speculation or eager computation. We are computing a + b and a + c without knowing that finally what will be selected. So, it is a kind of eager computation. Now, once optimization is done at the operator level, then the tool will implement those arithmetic operators using some predefined modules. So, what the tool will do is that the tool maps arithmetic operators to predefined modules implementing those operators.

So, an RTL synthesis tool typically has internally a set of implementations for various arithmetic and logical operators, for example, +, -, *, /, or ==, >, >=, <, <=. For all these kinds of arithmetic and relational operators that are allowed by Verilog, the RTL synthesis tool will internally have an implementation. And these implementation are typically parameterized module and during RTL synthesis what will the tool do it will instantiate these internal modules and choose the right set of parameters while

implementing. So, let us take an example and understand. Suppose, we are given this module in which there are two operators these arithmetic operators a + b and there is a relational operator c >d.

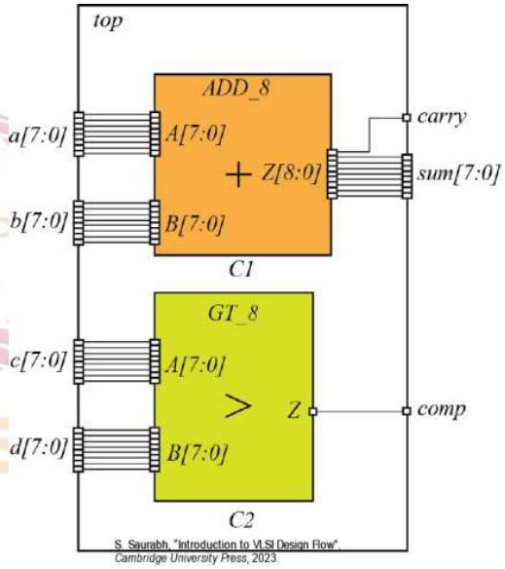
```
module top(a, b, c, d, sum, carry, comp);
      input [7:0]a, b, c, d;
      output [7:0]sum;
      output carry, comp;

      assign {carry,sum}= a + b;
      assign comp = (c > d);

endmodule
```

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

- Exact implementation tool-specific (logical function same)
- Architecture switching in later stages possible



Now how it will be implemented in a form of a circuit. So, the tool internally will have an adder; if adder implementation is there and the tool will simply instantiate it, it will instantiate this adder, and while instantiating, it will choose the parameter value. For example, the tool will have a parameterized module that is implementing an adder, and in this case, since the width of a and b are 8 bits, it will set the parameter value correspondingly. So, it may be setting the value as 8 for the parameter. So, the internal module, tool might have an internal parameterized module with the name add, and inside this module, there will be a parameter, for example, whose value is the width, and this parameter value is overridden based on what is required in the instantiation.

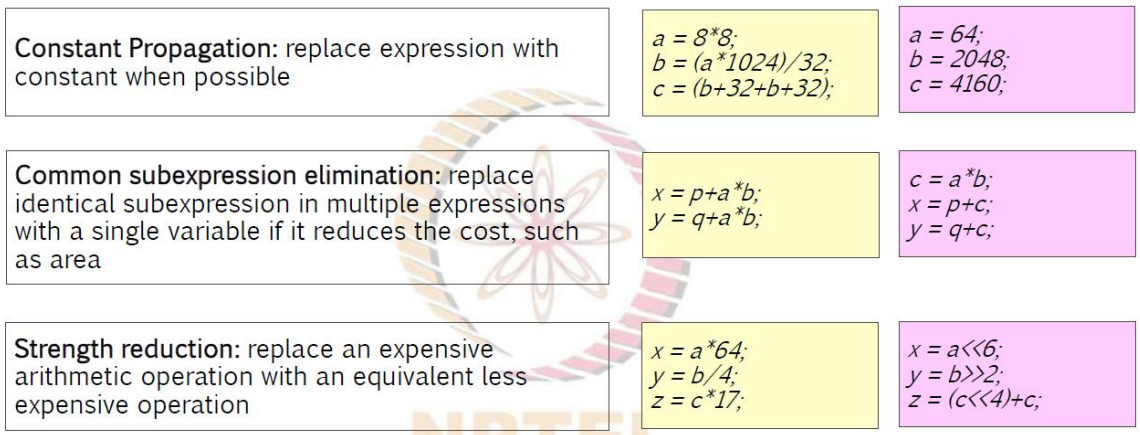
In this case, it will be overridden by the value 8. So, what the name of this module will be that is tool specific it is tool specific and also the implementation of these modules will be tool specific. So, it may happen that some RTL synthesis tool implement in some particular way this adder and another implementation tool will be implementing the same thing in a different manner. But functionally, the adders implemented by any tool should comply with what the addition operator means. So, functionally, it will be the same, but the internal implementation can be different.

So, corresponding to the other operator that is c>d, another module will be instantiated, and the name of this module will be tool-specific. Another important point to note is that there can be many implementations of the adder. For example, if we think of an adder, then the adder can be implemented as a ripple carry adder, or it can be implemented as a carry look ahead adder. So, the RTL synthesis tool might have both the representation or

both the implementations within itself. Initially, it may put, say, a ripple carry adder. It will instantiate a ripple carry adder for this because it does not know the timing of the circuit. Later on when it does the logic synthesis and also timing analysis and it discovers that this particular adder is coming in a critical path and we need to improve its timing or reduce its delay.

Then probably, it can switch to another architecture, for example, carry look ahead adder and instantiate that in our design. So, those kind of architectural switching can happen later in the design flow. An RTL synthesis tool can also apply compiler optimization techniques on the RTL code. So, the compiler optimization techniques have been adapted by these RTL synthesis tool to work on RTL constructs. And these types of optimizations can be applied, say, on the parse tree or the internal model of the RTL.

And these compiler optimizations are applied in passes and in each pass what is done is that a specific type of transformation is turned to the RTL representation such that some figures of merit improves. So, this type of compiler optimization gives significant gains for the arithmetic operators. So, let us look into a few examples of this compiler optimization techniques that are applied to RTL. So, there is a compiler optimization technique known as constant propagation. So, in this, what we do is that we replace expressions with constants whenever possible.



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

For example, suppose it is written a = 8*8, 8 multiplied by 8. So, in this case, instead of using a multiplier, a can be directly inferred to be a constant, and its constant value is 64. Similarly, once one variable is known to be constant, then wherever it appears, for example, in this expression, b = a*1024 divided by 32. So, a is known to be a constant, that is its value is 64. So, now b can also be evaluated as a constant, and once b is known to be constant c is also evaluated to be a constant, and its value is 4160.

So, if we do constant propagation so we infer one variable to be a constant and that can actually lead to other variables being constant and as a result we may be able to optimize our RTL expression and use a simpler logic gates or circuit elements rather than a more complicated gates. Another optimization that is targeted for arithmetic operators during RTL synthesis is common subexpression elimination. So, what is done is that it replaces identical subexpressions in multiple expressions with a single variable if it reduces the cost such as area. For example, suppose x was p + a * b and y was q + a *b. So, a * b is a common subexpression and therefore, we can write c = a*b and then we use c in the expression x and y.

So, if we do this kind of transformation then instead of two multipliers which were required earlier will require only one multiplier and will be able to save a. So, this kind of common subexpression eliminations are done during RTL synthesis and this is more applied for arithmetic operators at this stage because for the logical operators and bitwise operators there are more powerful techniques which can discover this common subexpression and use them to optimize the circuit. So, those optimization techniques are are done during later phases of the logic synthesis, but in particular during logic optimization. So, when we will be looking at logic optimization we will be looking at how common subexpressions are discovered for Boolean expressions and how that can be optimized. So, at this stage common subexpression elimination is performed based mostly for the arithmetic operators.

And then we can do a strength reduction. So, in strength reduction what is done is that we replace an expression or expensive arithmetic operation with an equivalent less expensive operation. For example, suppose it is given that x = a * 64. Now, if you want to multiply a number by 2, what we can do is just shift left, and if we want to multiply a number by power of 2, in this case, it is 64, then we have to shift left that many times as the number of power. So, in this case, we can write x << 6 times. If we just shift left, we can obtain a*64. So, instead of using a multiplier, we can just use the shifting operation, and that will cost much less than the multiplier.

Similarly, if we want to divide by 2 or a power of 2, we can shift right. So, in this case, suppose we are trying to find b divided by 4. So, instead of using a divisor or a circuit element that implements division, we can just shift right by 2 times, and we get y = b / 4. Now, if we have an expression where say c * 17. So, in this case also what we can do is that we can consider it as c* 17 is nothing but c * 16 + c.

Now, c is multiplied by 16 can be can be evaluated by shifting left 4 times and then adding c. So, since the multiplication operator takes more resources or more area rather than using a multiplier, it is better to use shift left and an addition operator. So, if you want to go deeper into the topics that we discussed today, you can refer to these books and summarize what we have discussed in the last two lectures. So, in the last two

lectures, we saw that given an RTL represented in Verilog we can do a synthesis and obtain a netlist out of it with some RTL-specific optimizations. Now, in the next lecture, we will be looking into how more optimizations can be done on the netlist that is generated by the RTL synthesis. Thank you very much.