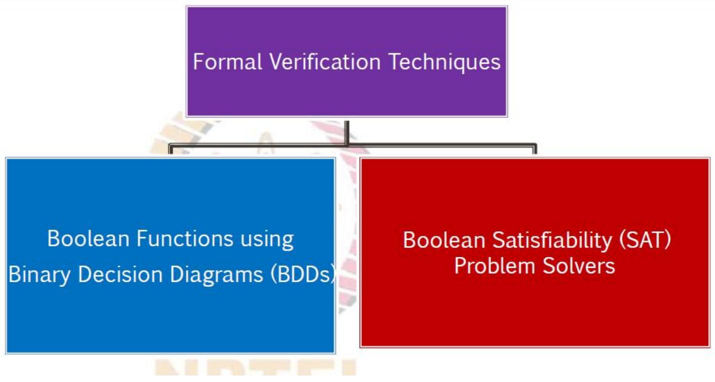
VLSI Design Flow: RTL to GDS

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

Lecture 23
Formal Verification-II

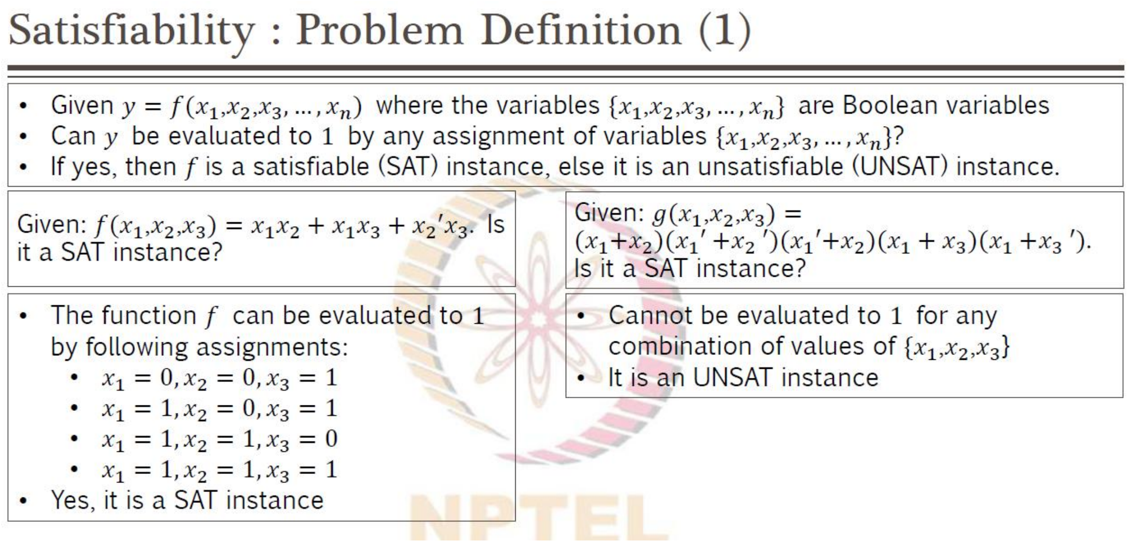
Hello everybody, welcome to the course VLSI Design Flow: RTL to GDS. This is the 18th lecture. In this lecture, we will be continuing with formal verification. In the earlier lectures, we had seen that two formal verification techniques that made the formal verification tools scalable for VLSI applications were Boolean function representations using binary decision diagrams and Boolean satisfiability solvers which were quite efficient. So, in the last lecture, we had looked into the BDD representation and in this lecture, we will be looking at Boolean satisfiability problem solvers or tools that solve the Boolean satisfiability problem or they are also known as SAT solvers. Now, let us first understand what is the satisfiability problem.



Formal Verification Techniques
Boolean Functions using Binary Decision Diagrams (BDDs)
Boolean Satisfiability (SAT) Problem Solvers

Suppose we are given a function y=f(x1, x2, x3, …, xn) where x1, x2, x3 to xn these are Boolean variables and we are asked the question that can y be evaluated to 1 by any assignment of the variables x1, x2, x3, …, xn. So, all these variables can take only binary values meaning 0s and 1. Can we find a combination of values for these x1, x2, xn such that the y can be evaluated to 1. If yes then f is said to be a satisfiable instance or SAT instance and if you are not able to find or it does not exist, if there does not exist any assignment of these Boolean variables such that y can become 1 then we say that y is an unsatisfiable instance or unsat instance. Now, let us take an example and understand it.

Suppose we are given a function f(x1, x2, x3)=x1x2+x1x3+x2'x3. Now, is it a SAT instance? We can easily see that if there are three product terms. Now, if any of these product terms are made 1 then f can be become 1. So, there can be many possible cases for example, if we take x1=1 and x2=1 this product term will be 1 and f will be 1. So, there the function f can be evaluated to 1 by many assignments for example, x1=0, x2=0 and x3=1.



Satisfiability : Problem Definition (1)

- Given $y = f(x_1, x_2, x_3, \dots, x_n)$ where the variables $\{x_1, x_2, x_3, \dots, x_n\}$ are Boolean variables
- Can y be evaluated to 1 by any assignment of variables $\{x_1, x_2, x_3, \dots, x_n\}$?
- If yes, then f is a satisfiable (SAT) instance, else it is an unsatisfiable (UNSAT) instance.

Given: $f(x_1, x_2, x_3) = x_1x_2 + x_1x_3 + x_2'x_3$. Is it a SAT instance?

- The function f can be evaluated to 1 by following assignments:
  - $x_1 = 0, x_2 = 0, x_3 = 1$
  - $x_1 = 1, x_2 = 0, x_3 = 1$
  - $x_1 = 1, x_2 = 1, x_3 = 0$
  - $x_1 = 1, x_2 = 1, x_3 = 1$
- Yes, it is a SAT instance

Given: $g(x_1, x_2, x_3) = (x_1 + x_2)(x_1' + x_2')(x_1' + x_2)(x_1 + x_3)(x_1 + x_3')$. Is it a SAT instance?

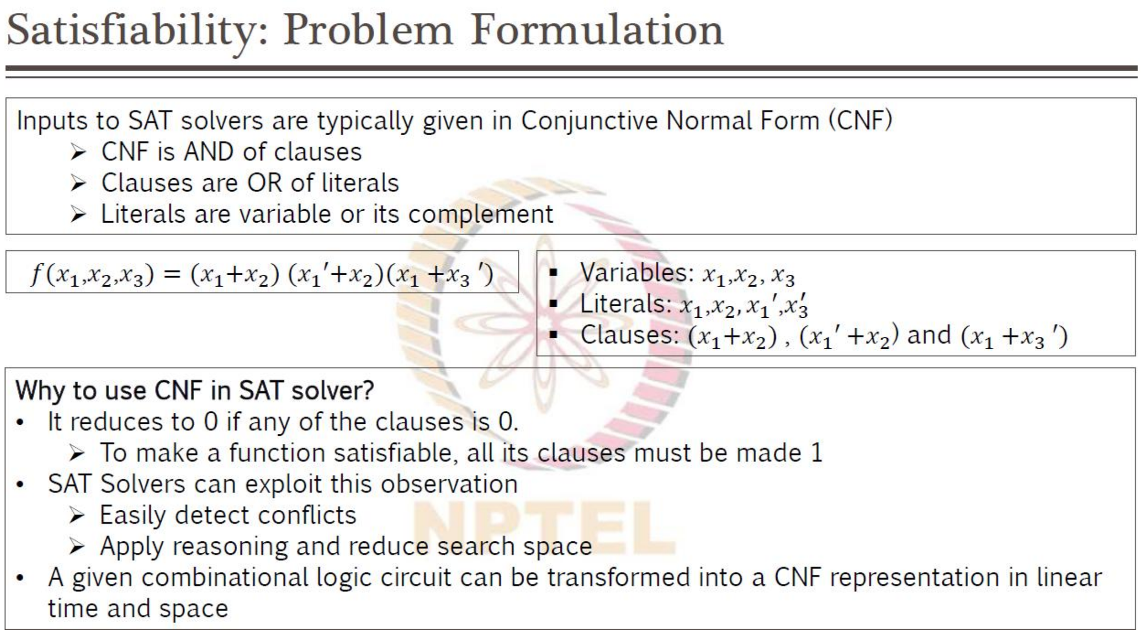
- Cannot be evaluated to 1 for any combination of values of $\{x_1, x_2, x_3\}$
- It is an UNSAT instance

If we take this assignment then this term the last term will become 1 and therefore, f will become 1. Similarly there are other assignments. Even if there was one assignment we would have said that f is satisfiable. So, since in this case there are four such assignments for which the value of f comes out to be 1 we say that this is a SAT instance. Now, even if there existed only one assignment, such that f was 1 we would have said that the f is a satisfiable instance. Now, let us take another function g(x1, x2, x3)= (x1+x2)(x1'+x2')(x1'+x2)(x1+x3)(x1+x3').

Now is it a SAT instance now to verify it we can simply draw a truth table for example, we can say that x1, x2, x3. Now there are three variables and therefore, there are 2 to the power 3 or 8 combinations for example, x1 can take 0, x2= 0 and x3= 0 or x1=0, x2=0 and 1 and so these are four possible cases. Similarly, there will be four other cases in which we will have 1, 1, 1, 1 here and the x2x3 will take 00, 01, 10 and 11. So, these are the eight possible cases now for each of them we can say what will be the value of g for example, if we take x1, x2, x3=0 what will happen now if any of this term becomes 0 then g will be 0. Now in this case x1+x2 takes a value 0 so this one will be of course, 0 and in here also x1+x2 is 0 so this one will also be 0 and we can say that these two will always be 0.

Similarly, if x1'+x2' this becomes 0 then also this function will become 0 for example, if we take this where x1=1 and x2=1 this one will become 0 and this one will also become 0. Similarly, we can evaluate for all the cases and it turns out that if we do it for all the cases we will get that for all cases the value of g will be 0. Now if that happens for a function then we say that it is an unsat function. So, in this case g cannot be evaluated to 1 for any combination of values of x1, x2, x3 and therefore, it is an example of an unsat instance. So, this is the basic problem of satisfiability.

So, given a function whether we can evaluate it to 1 or not, if yes then we can say that this function is sat otherwise it is unsat. Now, what are inputs to a sat solver? So, sat solvers typically takes input in terms of conjunctive normal form or CNF representation. So, this is a special type of representation of a Boolean function. What it is we will just see. Now what is CNF? CNF is AND of clauses and what is a clause? Clauses are OR of literals. So, a function represented in CNF is AND of various clauses, basically so AND of clauses, 1 clause, 2 clause, 3 clause and so on and these are ANDed together and each clause is a OR of literals meaning that we have plus and we will have literals here and so on.



Satisfiability: Problem Formulation

Inputs to SAT solvers are typically given in Conjunctive Normal Form (CNF)
- CNF is AND of clauses
- Clauses are OR of literals
- Literals are variable or its complement

$f(x_1,x_2,x_3) = (x_1+x_2)\,(x_1'+x_2)(x_1+x_3')$

- Variables: x_1, x_2, x_3
- Literals: x_1, x_2, x_1', x_3'
- Clauses: (x_1+x_2), $(x_1'+x_2)$ and (x_1+x_3')

Why to use CNF in SAT solver?
- It reduces to 0 if any of the clauses is 0.
  - To make a function satisfiable, all its clauses must be made 1
- SAT Solvers can exploit this observation
  - Easily detect conflicts
  - Apply reasoning and reduce search space
- A given combinational logic circuit can be transformed into a CNF representation in linear time and space

So, what are literals we have seen earlier that literals are variable or its complement. So, let us see an example of a function represented in CNF. So, this is a function which is represented in CNF. So, we have variables in this as x1, x2, x3 and what are the literals literals are x1, x2, x1' and x3' these are the literals that appear in this Boolean function and what are the clauses? This is 1 clause, this is another clause and this is third group. So, there are 3 clauses in this. Why? these clauses are OR of literals.

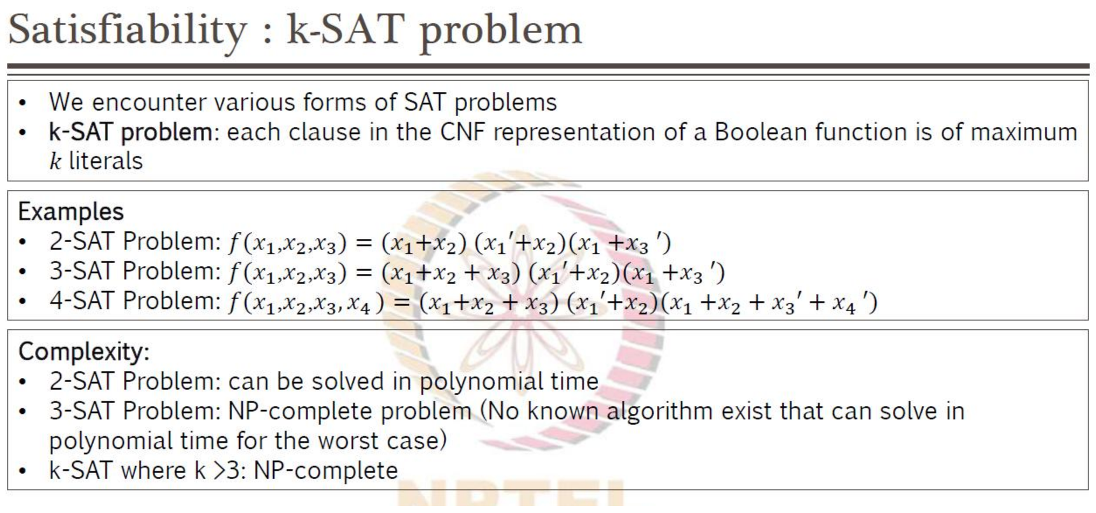
So, these are all OR of literals and we have finally, taken the AND, so dot is implicit. So, we have ANDed with. So, all these clauses are ANDed together and we got a function f in a CNF representation. Now, why to use a CNF in SAT solver why do we use a special representation of a Boolean function for a SAT solver? Because it reduces to 0 if any of the clause is 0. Now, in this if we represent a Boolean function in CNF if any of these clauses takes a value 0 then the whole function takes a value 0.

And therefore, what it means that to make a function satisfiable all its clauses must be made 1, all of them have to be made 1 then only the function is satisfied. If we cannot do that then the function is not satisfied or unsat and SAT solver can basically exploit this observation to easily detect conflicts on assignment. Meaning that it can say that ok when this variable takes 0, this variable takes 0 then the third variable cannot take 1 to make it satisfiable and so on. So, these kind of things and what is conflict we will see in more detail. So, those kind of conflicting requirement that if this variable takes this and this variable takes, this the third cannot take this or so on those kind of deductions can be done very easily by a SAT solver if a function is represented in a CNF representation.

And based on it the SAT solver can apply reasoning and reduce the search space and find a solution meaning that determine that whether function is SAT or an unsat in an efficient manner. And another thing is that suppose there is a combinational circuit. Now the combinational circuit is typically represented in terms of logic gates or maybe as say Boolean logic network or some other representation. Now if you are given a combinational logic circuit we can easily transform that combinational logic circuit into a CNF representation in linear time and space. So, there exists an algorithm which can transform a given circuit to a CNF representation in linear time and space meaning that it can transform it very efficiently. So, when we try to use a SAT solver for solving VLSI related problem what we do is that if we are given a combinational circuit we can easily transform that combinational logic circuit into a CNF representation and then give that CNF representation to the SAT solver to produce the desired result.

Now we encounter various forms of SAT problems in VLSI and we call them as a k-SAT problem if each clause in the CNF representation of a Boolean function is of maximum k literals. Now if we know that in a CNF representation you will have many clauses and each clause will have many literals. Now how many maximum number of literals can appear in a clause that defines what kind of k-SAT problem is. For example, let us suppose the function is like this. Now this function is in CNF representation now this clause is of 2 literals, this is also of 2 literals and this is also of 2 literals.

Since the maximum number of literals in any clause is 2 we can say that this is a 2-SAT problem meaning that 2 here represent the maximum number of literals appearing in the clauses. Now let us take the next one, we have 3 literals here and 2 literals and then 2 literals. So the maximum number number of literals in the clauses is 3 and therefore it is a 3-SAT problem. And in next let us see, in this function there are 3 literals, 2 literals and 4 literals so this is a 4-SAT problem. Now the complexity of how complex a SAT problem is dependent on that how many literals are there in the clauses that we are given and of course also on the number of clauses that appear in the CNF representation.



Satisfiability : k-SAT problem

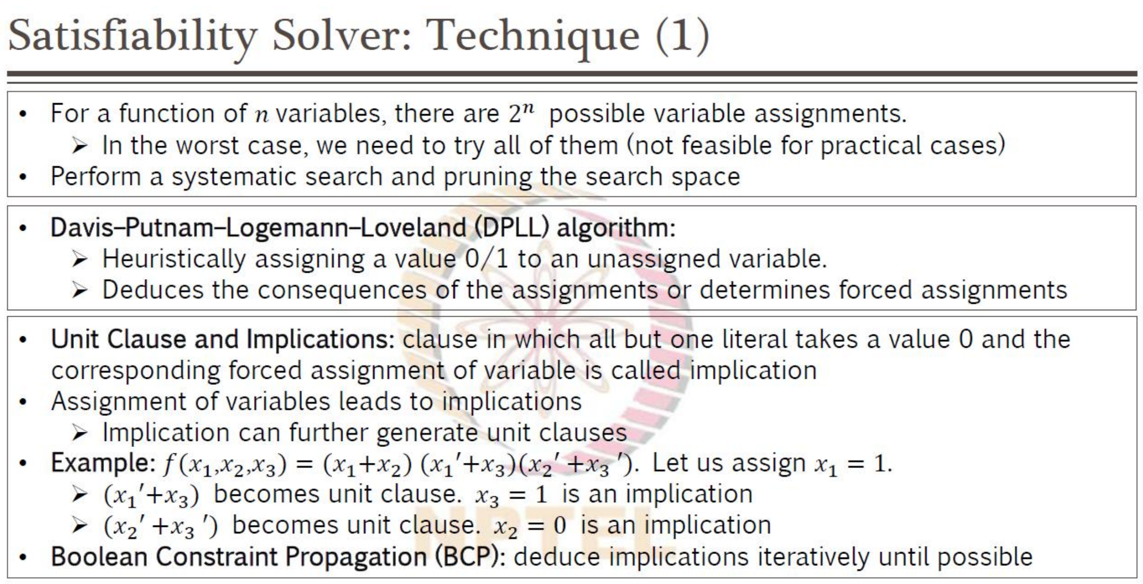
- We encounter various forms of SAT problems
- **k-SAT problem**: each clause in the CNF representation of a Boolean function is of maximum k literals

Examples
- 2-SAT Problem: $f(x_1,x_2,x_3) = (x_1+x_2)\ (x_1'+x_2)(x_1 + x_3')$
- 3-SAT Problem: $f(x_1,x_2,x_3) = (x_1+x_2 + x_3)\ (x_1'+x_2)(x_1 + x_3')$
- 4-SAT Problem: $f(x_1,x_2,x_3,x_4) = (x_1+x_2 + x_3)\ (x_1'+x_2)(x_1 + x_2 + x_3' + x_4')$

Complexity:
- 2-SAT Problem: can be solved in polynomial time
- 3-SAT Problem: NP-complete problem (No known algorithm exist that can solve in polynomial time for the worst case)
- k-SAT where k >3: NP-complete

Now if it is a 2-SAT problem meaning that if we restrict all the clauses to have maximum of 2 literals then the problem that we get is 2-SAT problem and it turns out that 2-SAT problem can be solved in polynomial time meaning that it is easier to solve in a polynomial time. But if we increase the maximum number of literals allowed in a clause to 3 and we consider 3-SAT problem then 3-SAT problem is an NP complete problem. And what is an NP complete problem? NP complete problem is a difficult problem and no known algorithm exists that can solve this problem in polynomial time for the worst case. So, NP complete problems are specific class of problems which are known to be difficult and for this class of problems in the worst case the complexity will not be polynomial. So in intuitive sense, an NP complete problems are difficult problems and we need to use some heuristics to solve those problems.

Now k-SAT problem where k>3 is an NP complete problem meaning that if k is 2 then it is polynomial, if k is 3 or greater then it is a NP complete problem. So, though SAT problem is an NP complete problem meaning that if k>3 it is an NP complete problem nevertheless there exists very efficient algorithms which can solve such SAT problem for many practical cases or many practical problems that we encounter in VLSI. Now for a function of n variables there are 2^n possible variable assignments. So, in the worst case we need to try all of them. So, if there are say n=100 or 50 in this case, the number of combinations that we need to try will be too many 2^{100}, it will be too many combinations to try one by one and simply it is not feasible to solve a given SAT problem, if the number of variables is large using a brute force.

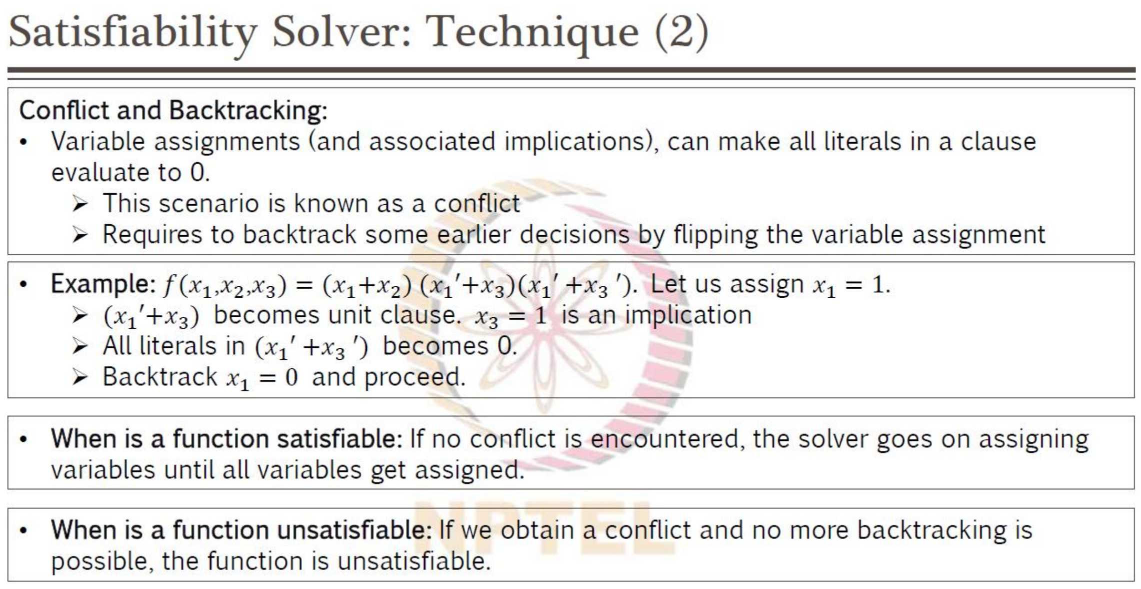


Satisfiability Solver: Technique (1)

- For a function of n variables, there are 2^n possible variable assignments.
  - ➢ In the worst case, we need to try all of them (not feasible for practical cases)
- Perform a systematic search and pruning the search space

- **Davis–Putnam–Logemann–Loveland (DPLL) algorithm:**
  - ➢ Heuristically assigning a value 0/1 to an unassigned variable.
  - ➢ Deduces the consequences of the assignments or determines forced assignments

- **Unit Clause and Implications:** clause in which all but one literal takes a value 0 and the corresponding forced assignment of variable is called implication
- Assignment of variables leads to implications
  - ➢ Implication can further generate unit clauses
- **Example:** $f(x_1,x_2,x_3) = (x_1+x_2)\,(x_1'+x_3)(x_2'+x_3')$. Let us assign $x_1 = 1$.
  - ➢ $(x_1'+x_3)$ becomes unit clause. $x_3 = 1$ is an implication
  - ➢ $(x_2'+x_3')$ becomes unit clause. $x_2 = 0$ is an implication
- **Boolean Constraint Propagation (BCP):** deduce implications iteratively until possible



Satisfiability Solver: Technique (2)

Conflict and Backtracking:
- Variable assignments (and associated implications), can make all literals in a clause evaluate to 0.
  - ➢ This scenario is known as a conflict
  - ➢ Requires to backtrack some earlier decisions by flipping the variable assignment

- **Example:** $f(x_1,x_2,x_3) = (x_1+x_2)\,(x_1'+x_3)(x_1'+x_3')$. Let us assign $x_1 = 1$.
  - ➢ $(x_1'+x_3)$ becomes unit clause. $x_3 = 1$ is an implication
  - ➢ All literals in $(x_1'+x_3')$ becomes 0.
  - ➢ Backtrack $x_1 = 0$ and proceed.

- **When is a function satisfiable:** If no conflict is encountered, the solver goes on assigning variables until all variables get assigned.

- **When is a function unsatisfiable:** If we obtain a conflict and no more backtracking is possible, the function is unsatisfiable.

So, what do SAT solvers do? They perform a systematic search and they prune the search space such that they need not explore solution in some regions of input combinations where solution cannot exist. So, they do not visit all possible cases, but they intelligently prune the search space such that a reduced space needs to be searched and based on that it can either find a satisfiable input combination or it can deduce that it is impossible to find a satisfiable input combination and therefore, a given input Boolean function is an unsat Boolean function. So, typically what SAT solvers use is a DPLL algorithm, Davis-Putnam-Logemann-Loveland algorithm or DPLL algorithm and their variations are used mostly by SAT solvers and what does this algorithm do? This algorithm basically heuristically assigns values 0 or 1 to unassigned variables and based on that assignments it deduces the consequences of the assignments and determines forced assignments. So, it says selects a variable, it says that ok let me assign it 0 or 1 and after assigning 0 or 1 it deduces that ok now if we assign this, what are the forced assignments and what do we mean by forced assignment let us understand that. Now there can happen that when we assign a value to a variable there are other clauses where all the variables got assigned except 1 if that happens then we say that clause has now become a unit clause.

So, a clause in which all, but one literal takes a value 0 and the corresponding force assignments of the variable is known as implication meaning that once we assign a variable 0 or 1 and maybe a group of variables we have assigned 0 or 1, then there can appear a clause in which all the literals are taking a value 0 except 1. Now if that is the case then we know that that particular variable which were not assigned will take either 0 or 1 based on its polarity. So, the corresponding force of assignment is known as the implication. So, assignment of a variable leads to implications and implication can further generate unit clauses and implications. So, let us take an example and understand it.

Suppose we are given a function (x1+x2)(x1'+x3)(x2'+x3). Now this is a given Boolean function in CNF representation. Let us assign x1=1. Now if we assign x1=1 then what happens to this clause, this clause gets satisfied we need not worry about it. Now what happens to this clause x1'+x3.

Now x1 we have assigned as 1 so this becomes 0. Now in this clause all but one variable that is x3 is assigned a value 0. So, now what do we need to assign to x3, we are forced to assign 1 and therefore x3=1 is an implication. Now if we assign x3=1 now this got satisfied meaning that the first two clauses became 1, what about the third one, this one. Now in this case we have assigned x1, x3 or we were forced to assign x3=1 therefore this term then this literal takes a value of 0 because it is x3' because we have made x3=1, x3' becomes 0.

And in this particular clause we have only x2' unassigned all others are 0 and therefore we are forced to have x2=0 such that this also becomes 1. So, the x2=0 is an implication in this. So, when we assign one of the variables in this case x1=1, there were forced assignments, so there was an unit clause and therefore we had an implication and this implication led to another unit clause and another implication. So, given a variable assignment, there can be multiple forced assignments or multiple implications which we are forced to make and therefore there is a task which is done inside the SAT solver and that task is known as Boolean constraint propagation. And in Boolean constraint propagation what it does is that it deduces implications iteratively until possible.

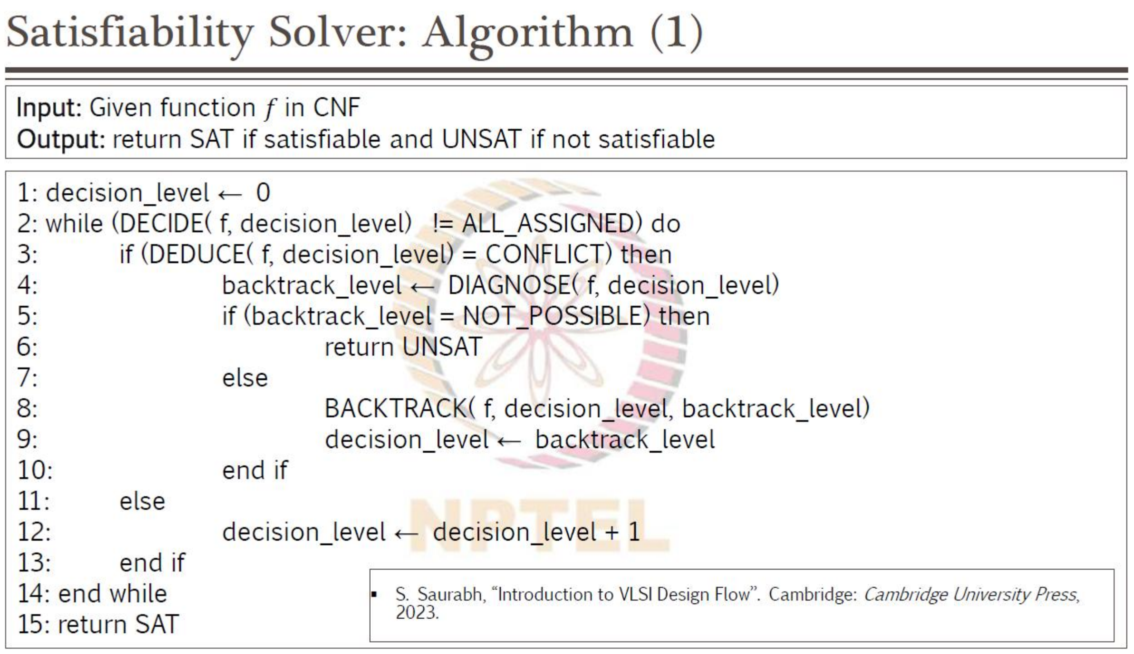
So, here we started with x1=1 and then x3=1 and x2=0 these were forced and so in Boolean constraint propagation these implications will be deduced and propagated until it is possible. Now sometimes the variable assignments can make a given clause, all of the literals of that as 0. So, if variable assignments and associated implications make all the literals in a clause evaluate to 0 then we have reached a kind of conflict we reached a situation which is known as conflict and then what we need to do is that we are required to do a backtracking meaning that some earlier decisions must be flipped, we need to toggle the earlier decisions, if we have chosen say x1=1 now we have to flip it and choose x1=0 then only we can proceed further. So, once we proceed we have chosen some variable assignment and that led to implications and we perform Boolean constraint propagation as a result of that we reach the point where for a given clause all the literals takes a value 0 then we have reached a conflict situation and then what we need to do is that we need to go back, need to backtrack and maybe toggle some of the variable assignment. So, that the function can become satisfied.

Now, for example, in this case in this Boolean function if we assign say x1=1, this gets satisfied no issue in this case x1' becomes 0. So, we are forced to assign x3=1. Now what happened in this case, for this clause now this 1, if we have assigned x1=1. So, this takes a value 0 and we have also assigned x3=1. So, this also takes value 0, so for this clause we have now got a conflict.

So, all literals in this clause now become 0 and that means that we have now got a conflict and we need to backtrack, backtracking means that now we have to toggle the value of x1=1 and consider x1=0. Now suppose we make x1=0 then we are forced to make x2=1 and if we have made this x1=0. So, this becomes satisfied and this we have also made this x1=0. So, this one will also get satisfied and we have a satisfiable function.

So, this is a satisfiable function. So, while assigning variable a conflict can be reached then we have to backtrack. Now when is a function satisfiable if no conflict is encountered the solver goes on assigning variables until all the variables gets assigned like in this case we toggled the earlier assignment and we proceeded then we reached the satisfiable solution and when we toggled it we did not get any conflict. So, the solver goes on assigning variable until all variables get assigned and we reach the satisfiable solution or we say that the function is satisfied and when is function not satisfiable or when it is unsat if we obtain a conflict and no more backtracking is possible then we say that the function is unsatisfiable. So, we go and try to toggle it for example, from x1=1 to x1=0, but we found that earlier we had already tried x1=0.

So, now there is nothing to toggle. Now if we have obtained a conflict and no more backtracking is possible then we declare that the function is unsatisfiable. Now let us look more into that how the code of this satisfiability solver may look like, a pseudo code for this. Now for a SAT solver what do we give as an input. So, we give a function f as in the CNF form and what is the output of a SAT solver, it returns SAT if the function f is satisfiable otherwise retains a label which is unsat where it is not satisfied and how will the code look like. So, we say that we start with a decision level of 0 we have not yet decided any variable then what we do is that we decide the value of a variable in the function f. Suppose f was having say 3 variables.



Satisfiability Solver: Algorithm (1)

Input: Given function f in CNF
Output: return SAT if satisfiable and UNSAT if not satisfiable

```
1: decision_level ← 0
2: while (DECIDE( f, decision_level)  != ALL_ASSIGNED) do
3:        if (DEDUCE( f, decision_level) = CONFLICT) then
4:                backtrack_level ← DIAGNOSE( f, decision_level)
5:                if (backtrack_level = NOT_POSSIBLE) then
6:                        return UNSAT
7:                else
8:                        BACKTRACK( f, decision_level, backtrack_level)
9:                        decision_level ← backtrack_level
10:               end if
11:       else
12:               decision_level ← decision_level + 1
13:       end if
14: end while
15: return SAT
```

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

So, first of all we now decide the value of a Boolean variable and if we are able to decide it and then we deduce that if that decision leads to conflict or not. So, suppose we assigned a value x1=1. Now there are 2 cases that can happen. After assignment, we did the Boolean constraint propagation and we got that there was no conflict at all. If there is no conflict at all we say that we just increase the decision level by 1. If there was a conflict, if we deduce and we find a conflict then what we do in that case we backtrack 1 level, diagnose that which variable we want to backtrack.

And then if backtracking is not possible we have exhausted all the possibilities then we say that it is an unsat problem, the function f is unsat otherwise what we say is that backtrack to the previous level or to some decision level and then the algorithm moves forward. Now if there is no conflict and we go on deciding variable and all get assigned in that case we return that the function is a SAT instant. Now this is just a pseudo code for solving the SAT problem, in actual implementation, the function DECIDE, the function DEDUCE, the function DIAGNOSE and the function BACKTRACK can vary from implementation to implementation. This is very much implementation specific and different SAT solver choose various kind of heuristics to decide which variable to assign or if given a conflict which variable to backtrack, some can choose the variable which was last assigned that is backtracked or some SAT solver can say that I will backtrack that decision variable which led to most number of conflicts and so on. So, all these things DECIDE, DEDUCE and BACKTRACK, these are very much implementation specific and different SAT solvers can use various heuristics in implementing these functions.

Now the algorithm that we saw in the previous slide was just a bare bone organizations of various functions or task that is carried out solving a SAT problem. Now over the last 20 years a lot of improvement has been made in the SAT solvers. For example, now SAT solvers intelligently do pre-processing of the given SAT problem rather than directly solving the given SAT problem, it first pre-processes and simplifies the problem and then moves ahead or it employs very efficient data structures for Boolean constraint propagation. So, that given an assignment it can propagate the values of the or the effect of that assignment in a very fast and efficient manner. And it also prunes the search space very intelligently so that it does not look into those areas or regions of the solutions where no solutions can be found.

So, it intelligently prunes the search space and also select some random restart strategy to quickly reach at the solution. And these days multi core processing has made the SAT solver really fast. So, now the SAT solvers have become very very powerful and those are used in solving many VLSI design problems as we will see in the subsequent lectures. Now, if you want to go deeper into the topics that I discussed today you can look into these references. So, now let me summarize what we have done in last two lectures.

In last two lectures we have looked into formal verification and we have seen that the formal verification techniques have become become popular in VLSI design flow due to two core techniques. The first one was BDD or the invention of BDD as a compact and canonical representation of Boolean function and the second one was the efficient SAT solvers. Now, in the next two lectures we will be looking at how these tools, BDD packages or Boolean representation using BDD and the SAT solvers can simplify or help us solve VLSI design problems. Thank you very much.