# Logic Synthesis & Verification, Fall 2023

**National Taiwan University**

## Programming Assignment 2

Due on 11/30 23:59 on GitHub.

*Submission Guidelines.* Please develop your code under `src/ext-lsv`. Please do not modify any code outside `src/ext-lsv`, and we will only copy your files under `src/ext-lsv` for evaluation. You are asked to submit your assignments by creating pull requests to your own branch. To avoid plagiarism, please push files and create pull requests at the last moment before the deadline. Please see the GitHub page (https://github.com/NTU-ALComLab/LSV-PA) for more details.

## 1 [Symmetry Checking with BDD] (50%)

Given a circuit $C$ in BDD form, an output pin $y_k$, and two input variables $x_i, x_j$, write a procedure in ABC to check whether the output pin $y_k$ is symmetric in $x_i$ and $x_j$. If not, show a counterexample to prove it. Integrate this procedure into ABC (under `src/ext-lsv/`), so that after reading in a circuit (by command "`read`") and transforming it into BDD (by command "`collapse`"), running the command "`lsv_sym_bdd`" would invoke your code. The command should have the following format.

`lsv_sym_bdd <`$k$`> <`$i$`> <`$j$`>`

where $k$ is the output pin index starting from 0, and $i$ and $j$ are input variable indexes starting from 0. If the $k^{\text{th}}$ output is symmetric in the $i^{\text{th}}$ and the $j^{\text{th}}$ variable, just print "`symmetric`", as shown below.

```
symmetric
```

Otherwise, print "`asymmetric`" and show a counterexample in the following format.

```
asymmetric
<pattern 1>
<pattern 2>
```

The values in the input pattern follow the input variable order. Note that your counterexample should be able to prove the asymmetry. In other words, patterns 1 and 2 should (1) lead to different output values for the $k^{\text{th}}$ output pin and (2) only differ in exchanging the values of the $i^{\text{th}}$ and the $j^{\text{th}}$ variable.

For example, suppose the BDD has only one output pin representing the function $y_0 = x_0x_1 + x_2$. The command and the output should look like:

```
abc 01> lsv_sym_bdd 0 0 1
symmetric
abc 02> lsv_sym_bdd 0 0 2
asymmetric
100
001
```

where the counterexample shows that $(x_0, x_1, x_2) = (1, 0, 0)$ and $(x_0, x_1, x_2) = (0, 0, 1)$ will lead to different $y_0$ values.

Notice. When operating BDDs, remember to use `Cudd_Ref` when you create a BDD node and use `Cudd_RecursiveDeref` when you dereference a BDD node. This helps to avoid `cuddGarbageCollect` errors. Here is an example showing how to use these commands.

```
Ddnode* cube = Cudd_ReadOne(manager);
for (int i = 0; i < n; ++i) {
    Ddnode* var = Cudd_bddIthVar(manager, i);
    Cudd_Ref(var);
    Ddnode* new_cube = Cudd_bddAnd(manager, cube, var);
    Cudd_Ref(new_cube);
    Cudd_RecursiveDeref(manager, cube);
    Cudd_RecursiveDeref(manager, var);
    cube = new_cube;
}
```

## 2 [Symmetry Checking with SAT] (50%)

Repeat Exercise 1 with all conditions being the same except that the circuit $C$ is in the form of AIG (by commands "`read`" and "`strash`"). Use SAT solver to check whether the output pin $y_k$ is symmetric in $x_i$ and $x_j$. Your procedure should implement the command in the following format.

`lsv_sym_sat <`$k$`> <`$i$`> <`$j$`>`

where $k$ is the output pin index starting from 0, and $i$ and $j$ are input variable indexes starting from 0.

The output format is the same as in Exercise 1. For example, suppose the logic network has only one output pin, which represents the function $y_0 = x_0x_1 + x_2$. The command and the output should look like:

```
abc 01> lsv_sym_sat 0 0 1
symmetric
abc 02> lsv_sym_sat 0 0 2
asymmetric
100
001
```

Hint 1. You can follow the following steps.
(1) Use `Abc_NtkCreateCone` to extract the cone of $y_k$.
(2) Use `Abc_NtkToDar` to derive a corresponding AIG circuit.
(3) Use `sat_solver_new` to initialize an SAT solver.
(4) Use `Cnf_Derive` to obtain the corresponding CNF formula $C_A$, which depends on variables $v_1, \ldots, v_n$.
(5) Use `Cnf_DataWriteIntoSolverInt` to add the CNF into the SAT solver.
(6) Use `Cnf_DataLift` to create another CNF formula $C_B$ that depends on different input variables $v_{n+1}, \ldots, v_{2n}$. Again, add the CNF into the SAT solver.
(7) For each input $x_t$ of the circuit, find its corresponding CNF variables $v_A(t)$ in $C_A$ and $v_B(t)$ in $C_B$. Set $v_A(t) = v_B(t) \forall t \notin \{i, j\}$, and set $v_A(i) = v_B(j)$, $v_A(j) = v_B(i)$. This step can be done by adding corresponding clauses to the SAT solver.
(8) Use `sat_solver_solve` to solve the SAT problem. Note that $y_k$ is symmetric in $x_i$ and $x_j$ if and only if $v_A(y_k) \oplus v_B(y_k)$ is unsatisfiable, where $v_A(y_k)$ and $v_B(y_k)$ are the CNF variables in $C_A$ and $C_B$ that corresponds to $y_k$.
(9) If $y_k$ is asymmetric in $x_i$ and $x_j$, use `sat_solver_var_value` to obtain the satisfying assignment, which can be used to derive the counterexample.

Hint 2. To use `Abc_NtkToDar` and `Cnf_Derive` functions, you should include the following code.

```
#include "sat/cnf/cnf.h"
extern "C"{
    Aig_Man_t* Abc_NtkToDar( Abc_Ntk_t * pNtk, int fExors, int fRegisters );
}
```

Hint 3. The variable orders in CNF differ from the ones in AIG. For a pointer `pObj` in `Aig_Obj_t*` type, you can use `pCnf->pVarNums[pObj->ID]` to find its variable index in `pCnf`. If the pointer `pObj` is from the original network in `Abc_Obj_t*` type, to make it have the same ID as its counterpart in AIG, make sure you set the last parameter of `Abc_NtkCreateCone` to 1 in step (1), so that all input variables are always included.

Hint 4. You can refer to our GitHub page (https://github.com/NTU-ALComLab/LSV-PA/wiki/Reasoning-with-SAT-solvers) for more details about using SAT solvers in ABC.

## 3 [Symmetry Checking with Incremental SAT] (Bonus 20%)

Repeat Exercise 2, but this time, only $k$ is given, and you are asked to find out all $(i, j)$ pairs $(i < j)$ to make the output pin $y_k$ symmetric in $x_i$ and $x_j$. Your procedure should implement the command in the following format.

`lsv_sym_all <`$k$`>`

For the output format, list each $(i, j)$ pair in a line. For example, suppose the logic network has only one output pin, which represents the function $y_0 = x_0x_1x_2 + x_3x_4$. The command and the output should look like:

```
abc 01> lsv_sym_all 0
0 1
0 2
1 2
3 4
```

You must use incremental SAT to solve this problem. We will prepare larger benchmarks to test your program, so just calling the function in Exercise 2 iteratively will not be fast enough. We have released a benchmark `demo.blif` at the GitHub page. As an example, your program should be able to finish $k = 50$ for this benchmark in 90 seconds.

Hint 1. Note that symmetry satisfies transitivity. In other words, given that $y_k$ is symmetric in $(x_i, x_j)$ and $(x_j, x_k)$, then $y_k$ is also symmetric in $(x_i, x_k)$.

Hint 2. For incremental SAT solver, you can follow steps (1) to (6) in Exercise 2 to build up an SAT solver. Next, add the following constraints into the solver, where $m$ is the number of input variables of the circuit.

(a) $v_A(y_k) \oplus v_B(y_k)$

(b) $(v_A(t) = v_B(t)) \vee v_H(t)$ , for all $0 \le t < m$

(c) $(v_A(t_1) = v_B(t_2)) \vee \neg v_H(t_1) \vee \neg v_H(t_2)$ , for all $0 \le t_1 < t_2 < m$

(d) $(v_A(t_2) = v_B(t_1)) \vee \neg v_H(t_1) \vee \neg v_H(t_2)$ , for all $0 \le t_1 < t_2 < m$

There are $m$ control variables $v_H(0)$ to $v_H(m-1)$, which are used to enable or disable clauses in (b) to (d). When checking symmetry between $x_i$ and $x_j$, we set $v_H(i) = v_H(j) = 1$ and others to 0. These unit assumptions can be taken by `sat_solver_solve` when solving the problem. Then some constraints will be automatically satisfied (disabled). You can find that the enabled constraints are as follows.

(i) Constraints in (a) are always enabled.

(ii) Constraints in (b) that are not related to $i$ and $j$ are enabled, indicating that variables other than $x_i$ and $x_j$ should be the same between $C_A$ and $C_B$.

(iii) Constraints in (c) and (d) that are related to $i$ or $j$ are enabled, indicating that variables $x_i$ and $x_j$ should swap their values between $C_A$ and $C_B$.

Therefore, the enabled clauses are exactly the same as in Exercise 2.