FOF (Functionally Observable Fault): A unified mode for testing and debugging - ATPG and application to debugging -

Masahiro Fujita

VLSI Design and Education Center

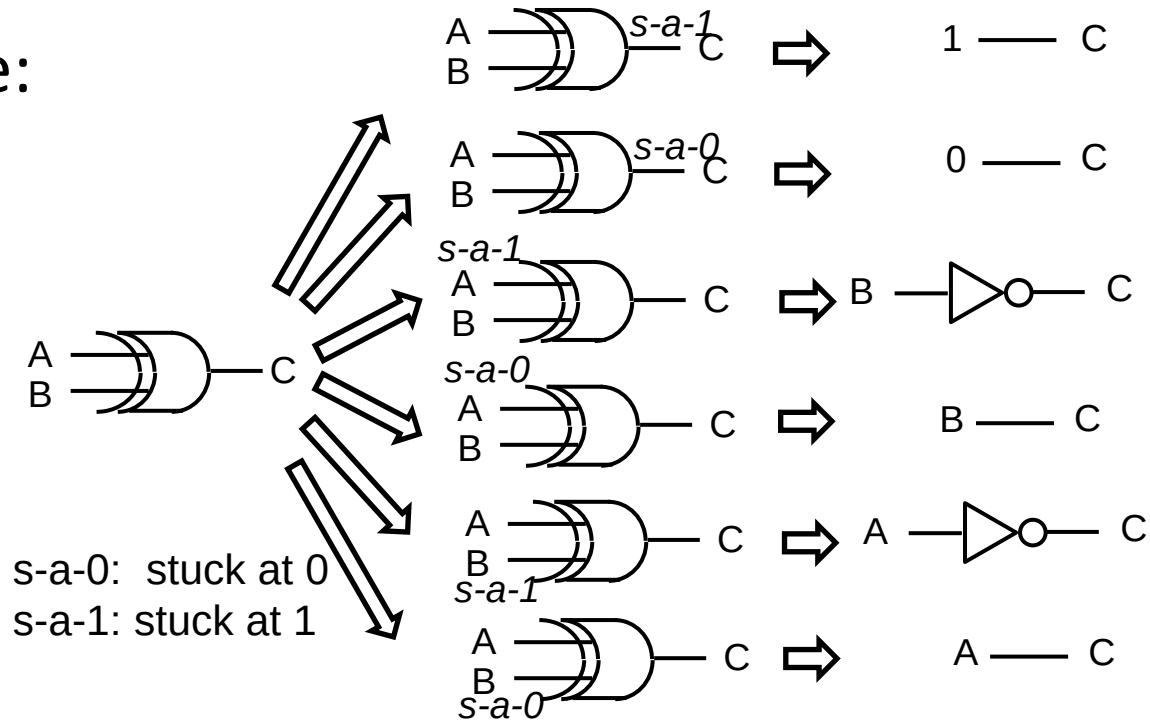
(VDEC)

University of Tokyo

Stuck-at fault and functional fault

For an 2-input EOR gate:

- Only 6 different functions with stuck-at faults
- $(2^{2^2}-1)=15$ different functions with functional faults



Non-faulty

x	y	f
0	0	0
0	1	0
1	0	0
1	1	0

x	y	f
0	0	1
0	1	0
1	0	0
1	1	0

x	y	f
0	0	0
0	1	1
1	0	0
1	1	0

x	y	f
0	0	1
0	1	1
1	0	0
1	1	0

x	y	f
0	0	0
0	1	0
1	0	1
1	1	0

x	y	f
0	0	1
0	1	0
1	0	1
1	1	0

x	y	f
0	0	0
0	1	1
1	0	1
1	1	0

x	y	f
0	0	1
0	1	1
1	0	1
1	1	0

x	y	f
0	0	0
0	1	0
1	0	0
1	1	1

x	y	f
0	0	1
0	1	0
1	0	0
1	1	1

x	y	f
0	0	0
0	1	1
1	0	0
1	1	1

x	y	f
0	0	1
0	1	1
1	0	0
1	1	1

x	y	f
0	0	0
0	1	0
1	0	1
1	1	1

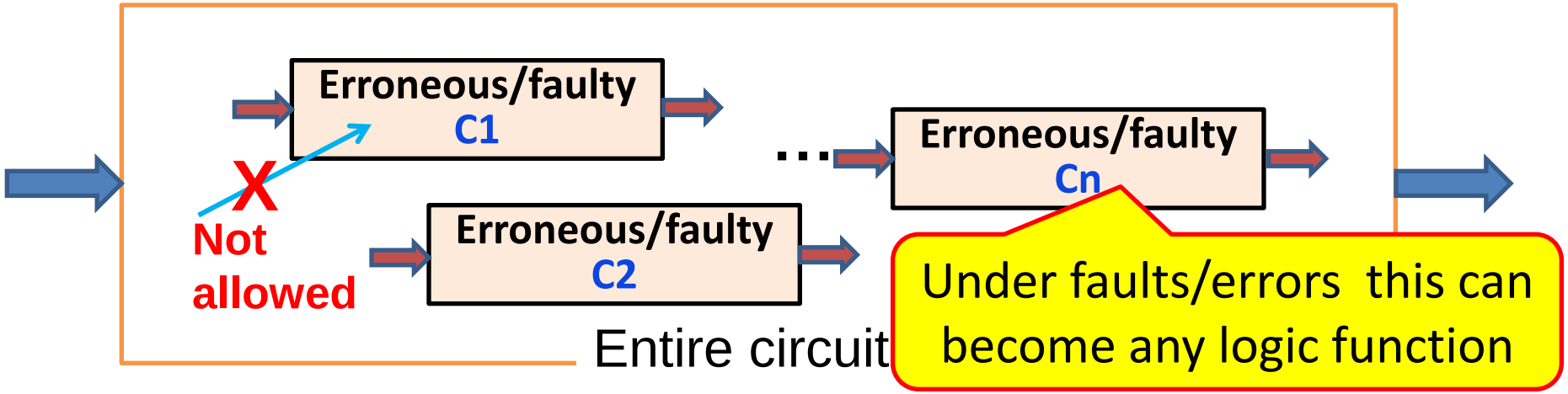
x	y	f
0	0	1
0	1	0
1	0	1
1	1	1

x	y	f
0	0	0
0	1	1
1	0	1
1	1	1

x	y	f
0	0	1
0	1	1
1	0	1
1	1	1

FOF: Functionally Observable Fault

- Any faults/errors which exist only inside sub-circuits
- They cannot refer to signals outside of the sub-circuits
- Many simultaneous multiple faults/errors are targeted



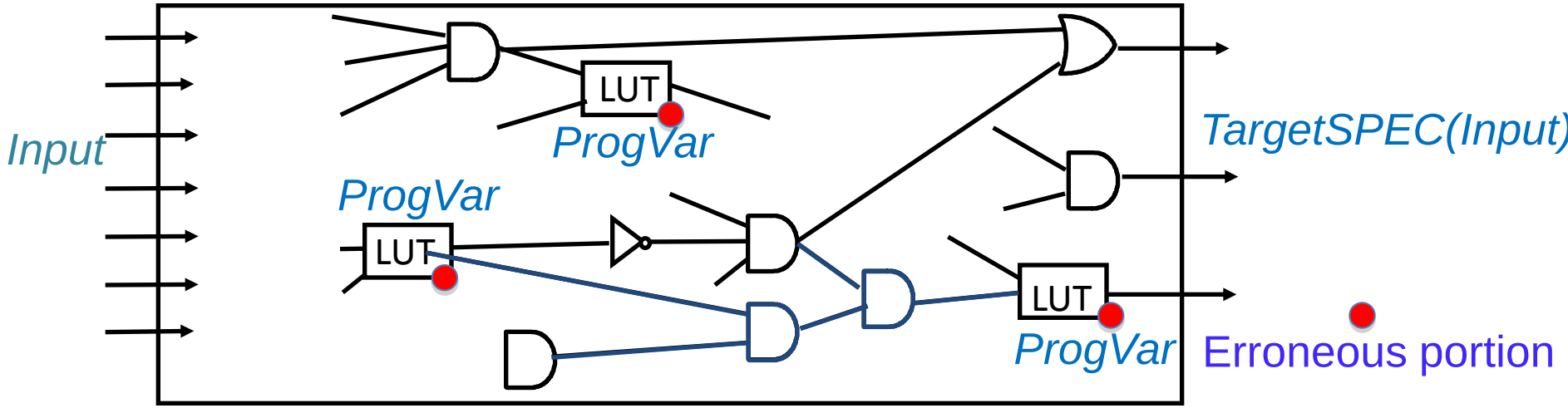
- Faulty/erroneous sub-circuits are replacing suspicious portions of the buggy entire circuit
  - FOF can be a good model for logical debugging as well as test
- Is there efficient procedures, say based on SAT available ?
  YES! Need Incremental SAT instead of SAT

Problem formulation

- Look up Table (LUT) can represent all logic functions with the set of inputs
  - Introducing programmable variables (see next slide)
- Then the problem becomes QBF (Quantified Boolean Formula)

$\exists ProgVar.\ \forall Input.\ f\ (ProgVar, Input) = TargetNewSPEC(Input)$

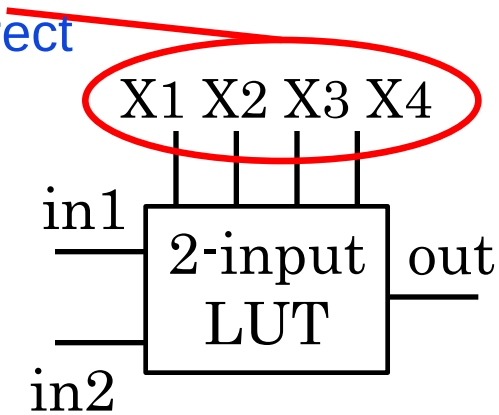
- Given appropriate programming of LUT, the circuit is equivalent to the spec



Representation of LUT with ProgVar

- Each row of truth table is assigned a program variable
- 2-inut LUT needs 4 variables, 4-input LUT needs 16 variables, and so on
- Can represent all possible logic functions
  - Determining values of ProgVar is to decide the logic function

Assign appropriate values to X1-X4 so that the outputs of the circuit becomes correct

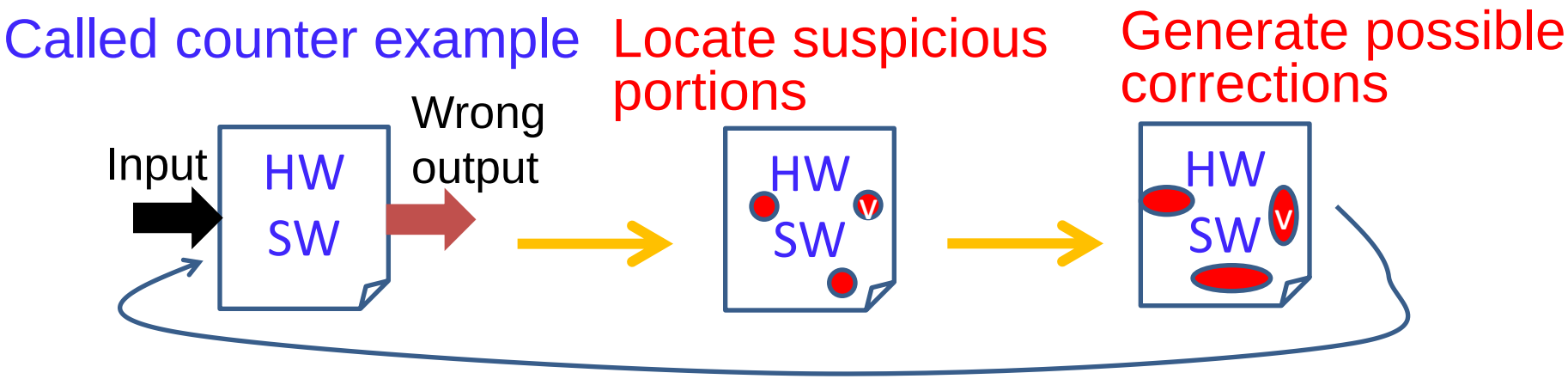


Truth table for LUT

in1	in2	out
0	0	X1
0	1	X2
1	0	X3
1	1	X4

Debugging problem

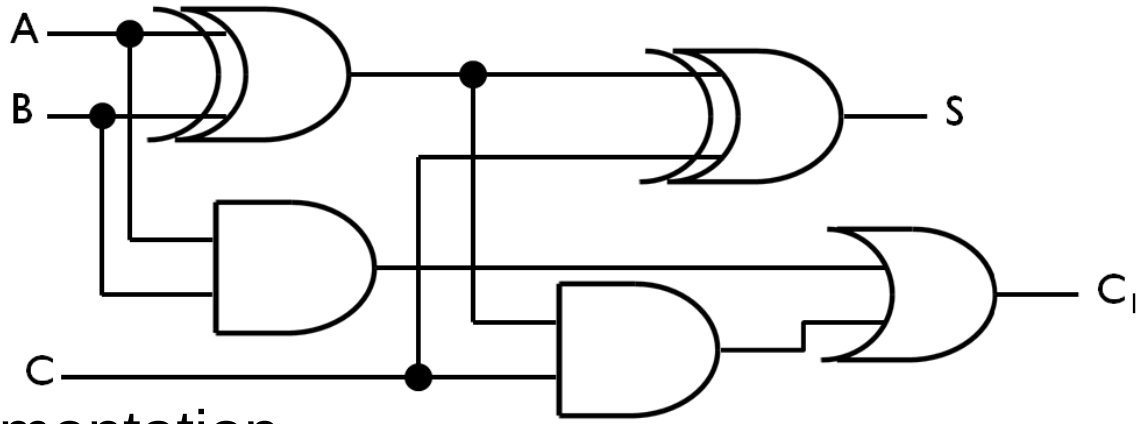
- Given counter examples (test cases that do not produce correct outputs)
  - Locate suspicious portions of HW/SW
  - Come up with possible correction on them
  - Make sure the corrected HW/SW is OK (start verification process again). If not, repeat the debugging process



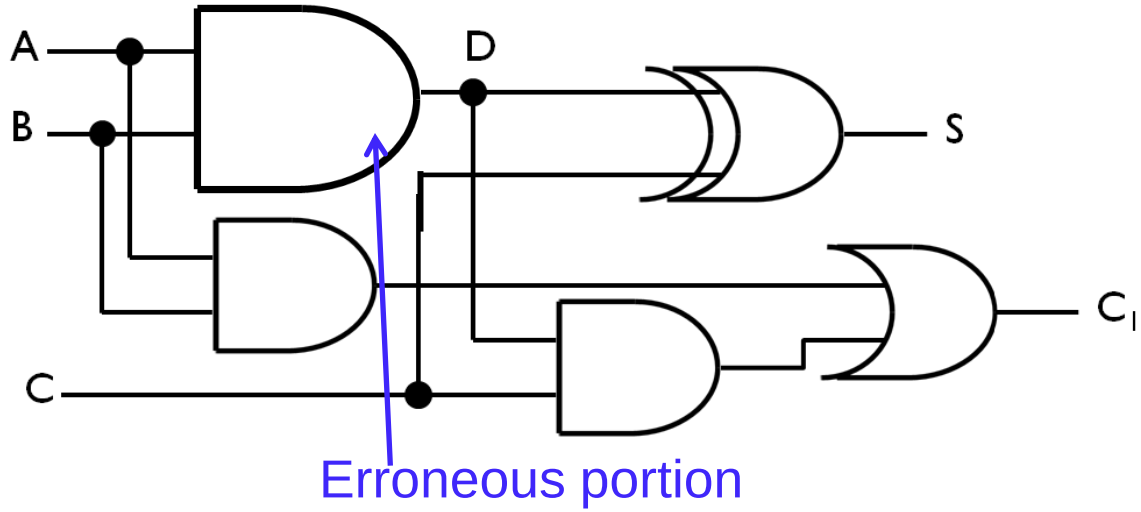
Verification efforts may have to be repeated many times !

A buggy design for a 1-bit full adder

- Specification



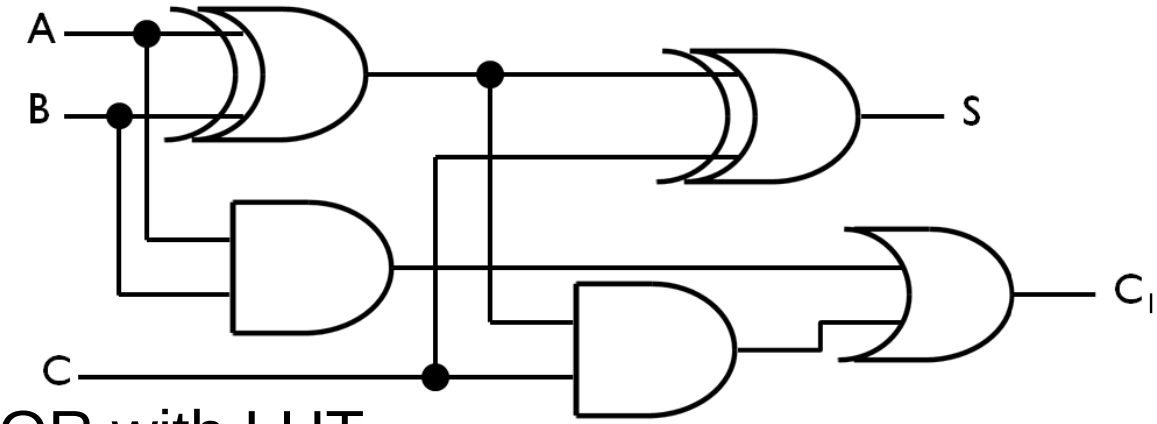
- A buggy implementation



A trace of correction for a 1-bit full adder

$$\exists X \forall A, B, C. f(X, A, B, C)$$

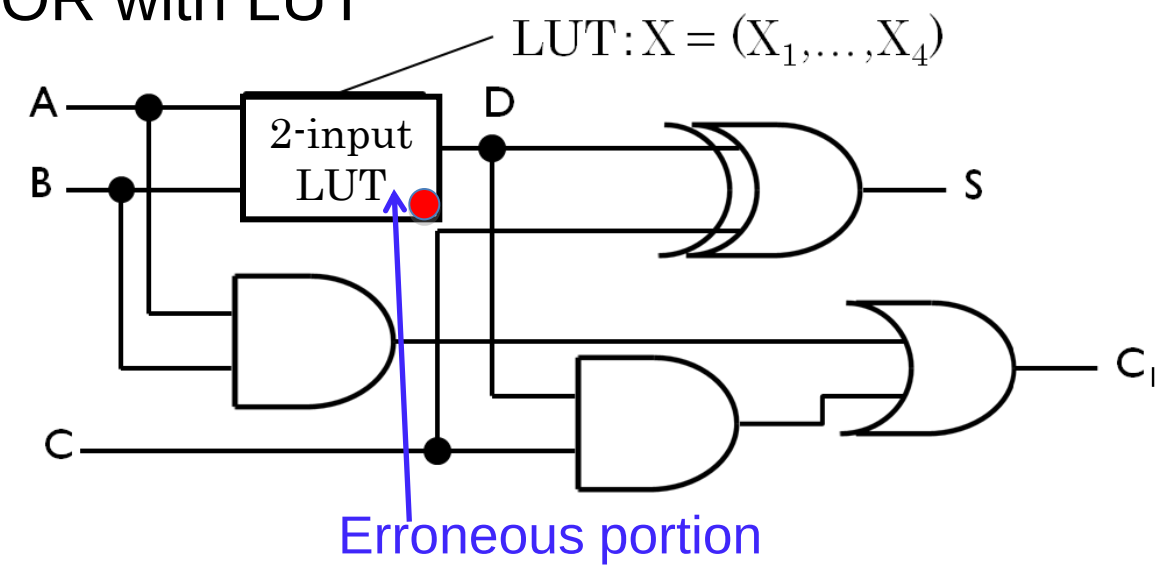
◆Specification



◆Replace an EOR with LUT

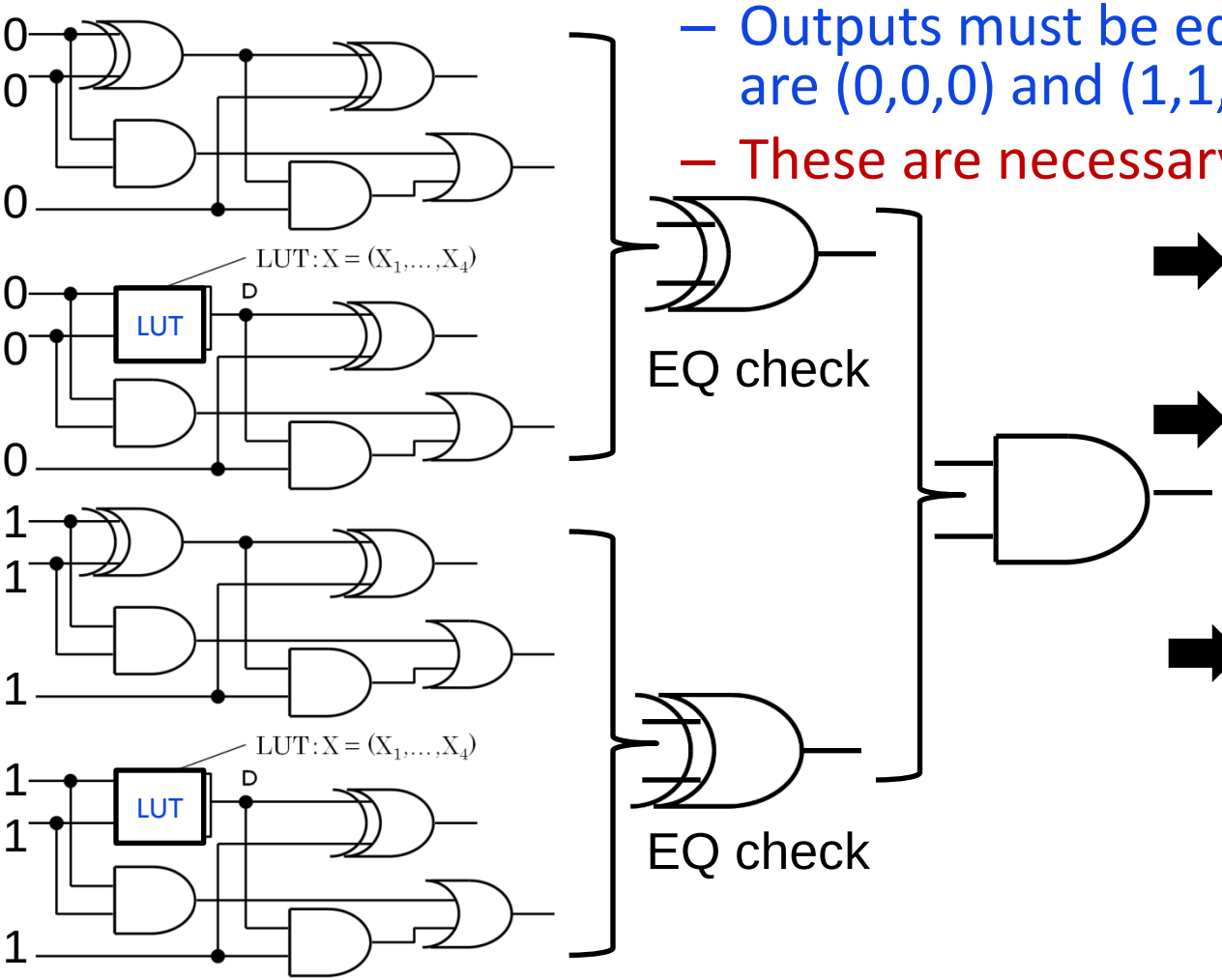
Truth table for LUT

A	B	D
0	0	X1
0	1	X2
1	0	X3
1	1	X4



Simple example: Step1

$$\exists X . f(X,0,0,0) \wedge f(X,1,1,1)$$



- Outputs must be equal when inputs are (0,0,0) and (1,1,1)
- These are necessary conditions

➡ Can be solved as simple SAT problem

➡ If unsat, there is no way to rectify

➡ Truth table for LUT

A	B	D
0	0	0
0	1	0
1	0	0
1	1	0

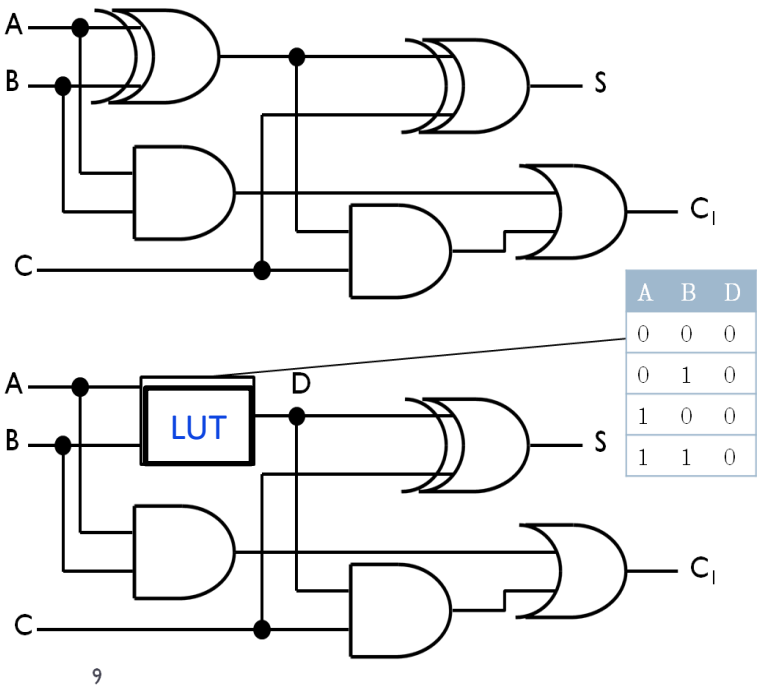
Current solution

Synthesis from sample values (necessary conditions)

Simple example: Step2

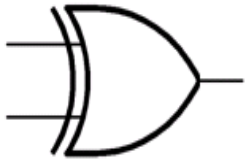
$$\exists ABC.\neg f(0000, A, B, C)$$

- Based on current solution, check if it is actually a real solution
- Outputs can never become non-equal



A	B	D
0	0	0
0	1	0
1	0	0
1	1	0

Non-EQ check



➡ Typical SAT formulation

➡ If unsat, current solution candidate is real solution

➡ (A,B,C) = (0,1,0) is a counter example

Simple example: Step3

- Add the counter example to the constraints
  - Outputs must be equal when inputs are (0,0,0), (1,1,1) and (0,1,0) $\exists X . f(X,0,0,0) \wedge f(X,1,1,1) \wedge f(X,0,1,0)$
  - Generate solution candidates for LUT
    - ⇨Any counter example with the current solution ?
    - ⇨Add counter example to constraints

Inputs
(0,0,0)
(1,1,1)
(0,1,0)

Truth table for LUT

A	B	D
0	0	0
0	1	1
1	0	0
1	1	0

Inputs
(0,0,0)
(1,1,1)
(0,1,0)
(1,0,1)

Truth table for LUT

A	B	D
0	0	0
0	1	1
1	0	1
1	1	0

There is no counter example and so finished

This two-level QBF can be efficiently solved by "incremental" SAT solvers

- Basically we are solving the following "incremental SAT problems" until becoming UNSAT

$\exists Y.X.f(Y,X) \neq SPEC(X)$ => SAT, solution is (y_1,x_1)

$\exists Y.X.f(Y,X) \neq Spec(X) \wedge f(Y,x1) = Spec(x1)$

=> SAT, solution is (y_2,x_2)

$\exists Y.X.f(Y,X) \neq Spec(X) \wedge f(Y,x1) = Spec(x1) \wedge f(Y,x2) = Spec(x2)$

=> SAT, solution is (y_3,x_3)

...

$\exists Y.X.f(Y,X) \neq Spec(X) \wedge f(Y,x_1) = Spec(x_1) \wedge f(Y,x_2) = Spec(x_2)$

$\ldots \wedge f(Y,x_{n-1}) = Spec(x_{n-1})$ => SAT, solution is (y_n,x_n)

$\exists Y.X.f(Y,X) \neq Spec(X) \wedge f(Y,x_1) = Spec(x_1) \wedge f(Y,x_2) = Spec(x_2)$

$\ldots \wedge f(Y,x_{n-1}) = Spec(x_{n-1}) \wedge f(Y,x_n) = Spec(x_n)$ => UNSAT

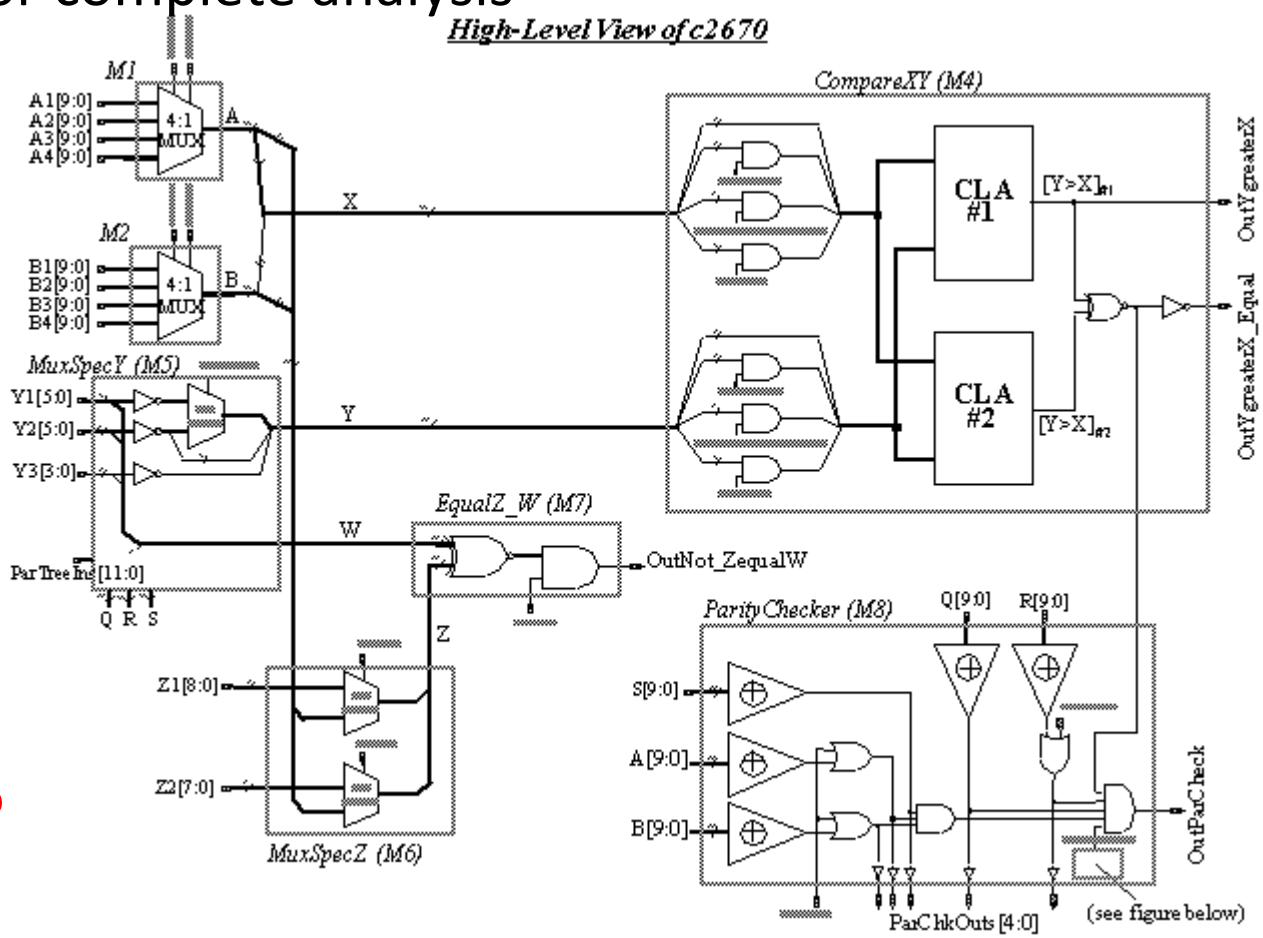
- Then $(x_1,x_2,\ldots,x_n)$ are complete test vectors !
- Word-level designs can be similarly ATPGed with SMT

Observation

- With slight modification, we can generate sets of test vectors by which 100% (and so formal analysis) correctness of "portions" of circuits can be checked
  - Checking if there is any other solution candidate which behaves differently from the current solution
  - This is a SAT problem
- Test vectors realize complete verification assuming that all the other portions of circuits are correct (fixed)
  - And we can model simultaneous multiple faults/errors easily
- Numbers of iterations are the numbers of test vectors needed
  - Normally tens or hundreds for ISCAS85 and OpenRISC

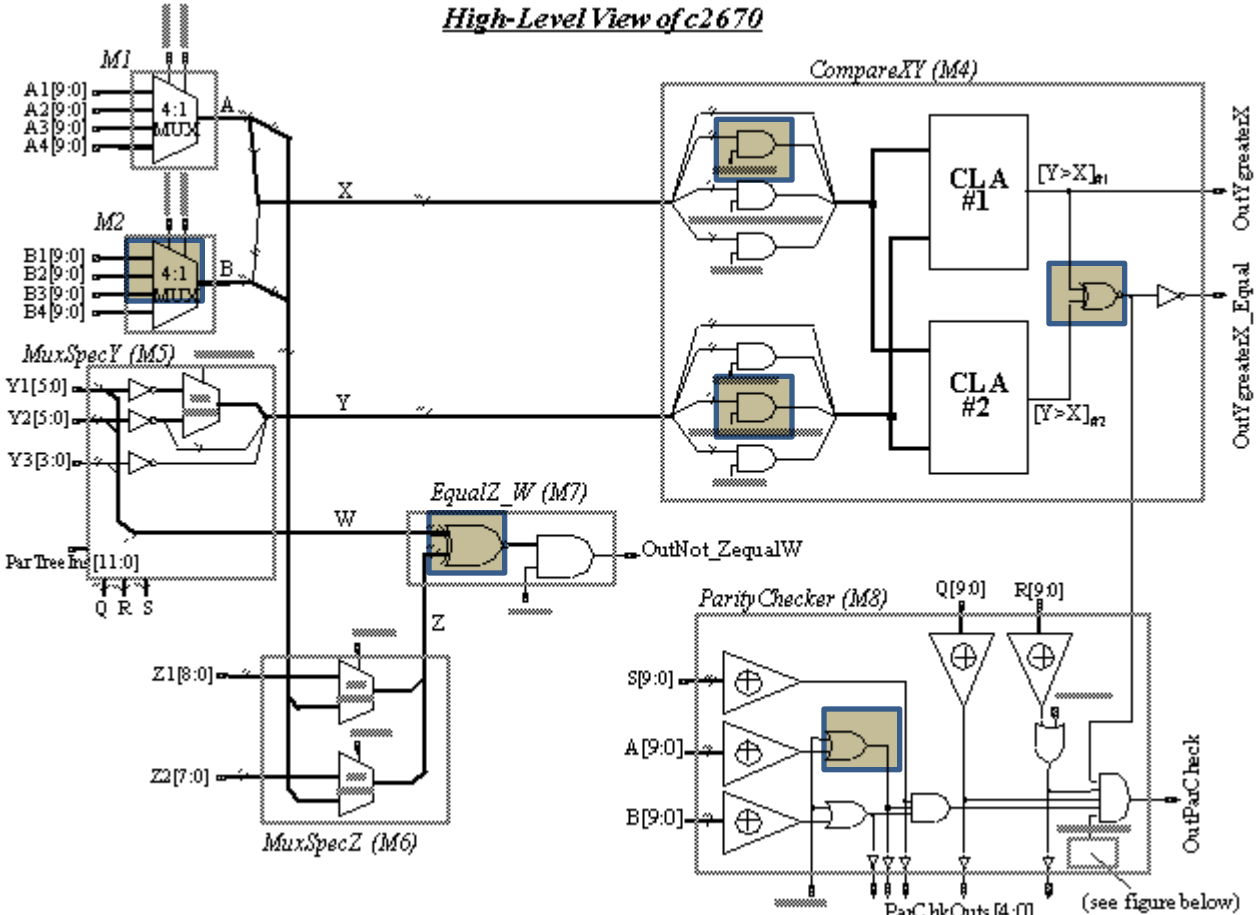
Larger example (1)

- ISCAS85 circuit, c2670: 12-bit ALU and controller
  - 233 inputs; 140 outputs; 1193 gates
  - If we do not know internal structure (below), we need to simulate 2^{233} for complete analysis
- In other words, if we like to analyze entire circuits, we have to do so implicit/explicit
- But if we only like to analyze their portions, any difference ?



Larger example (2)

- If we like only to check the grey areas are correct or not, do we need 2^{223} vectors for complete analysis ?
  - No, we need only 100 primary input vectors for 100% analysis
  - There is an assumption that non grey areas are correct (fixed)



ATPG results for ISCAS85 circuits

- All error/faulty sub-circuits are two-input gates
- Number of test vectors grows with (#faulty sub-circuits)$^{0.7}$

Circuit	#gates	Number of sub-circuits for FOF	Average time (in sec)	Average number of test patterns	Maximum number of test patterns for 20 trials	(Average numbers of vectors) / (Numbers of faulty sub-circuits)$^{0.7}$
c499	202	10	0.9	7.3	16	1.46
		20	1.9	12.6	23	1.55
		50	4.3	23.9	36	1.54
		100	9.7	37.0	48	1.47
c880	383	10	4.1	23.91	33	4.77
		20	10.7	42.8	53	5.25
		50	35.0	78.5	93	5.08
		100	97.0	121.3	148	4.83
c1350	546	10	5.3	20.3	26	4.05
		20	9.7	29.3	33	3.59
		50	31.2	57.3	65	3.70
		100	69.2	82.4	97	3.28
c1908	880	10	4.5	15.7	22	3.12
		20	10.6	27.5	39	3.37
		50	29.80	50.4	71	3.26
		100	73.8	78.4	95	3.12
c2670	1193	10	9.1	20.0	22	3.98
		20	25.3	36.9	44	4.52
		50	104.8	79.7	111	5.15
		100	276.2	128.5	140	5.11
c3540	1669	10	10.2	18.3	25	3.59
		20	21.6	30.0	37	3.31
		50	85.5	64.1	71	4.07
		100	240.4	107.5	127	3.90
c5315	2406	10	16.3	20.4	28	4.07
		20	37.8	34.0	40	4.18
		50	124.0	66.3	84	4.28
		100	328.7	109.4	127	4.35

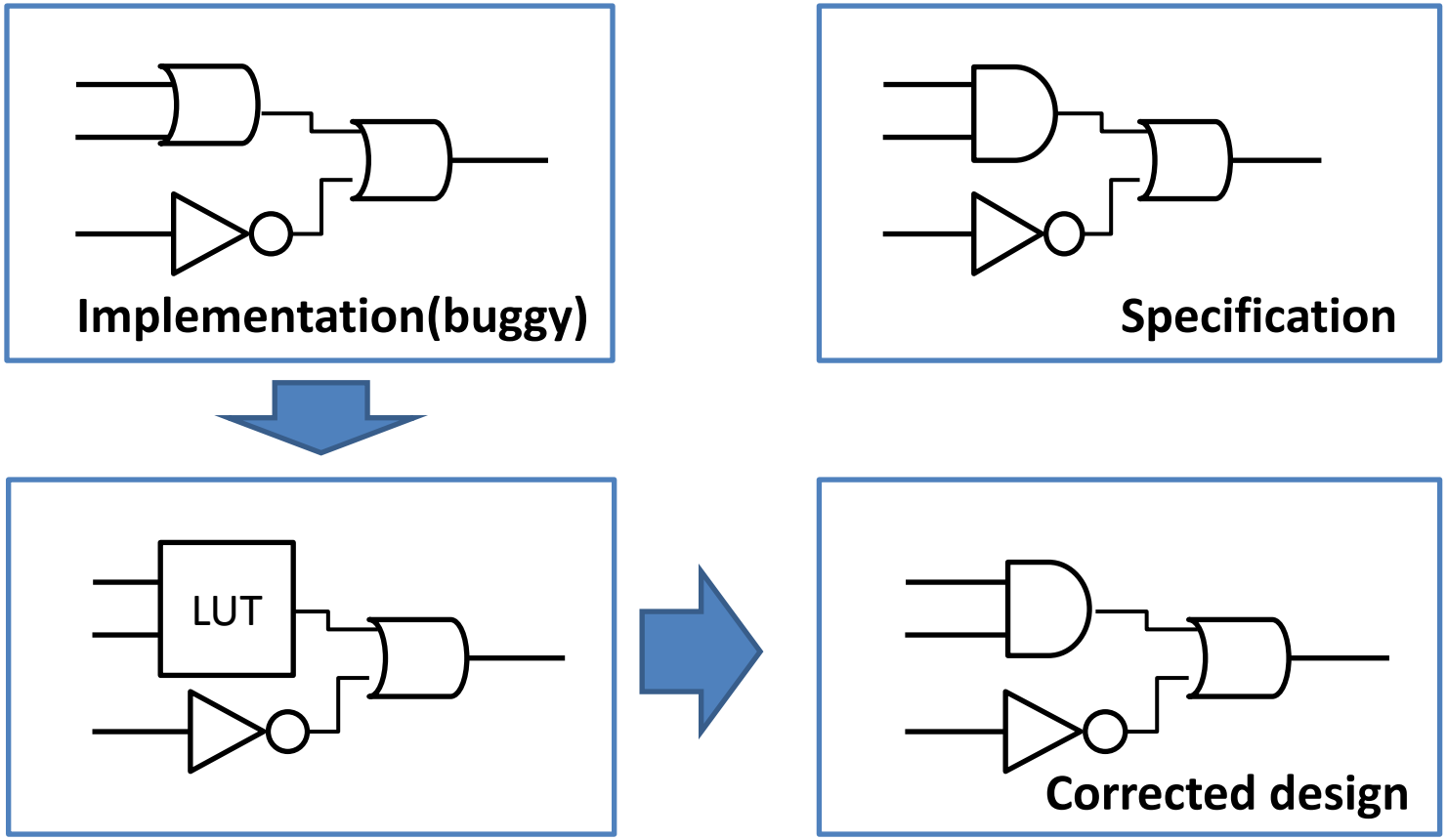
Results by incremental SAT solvers

- Incremental SAT solvers are 10-40X faster

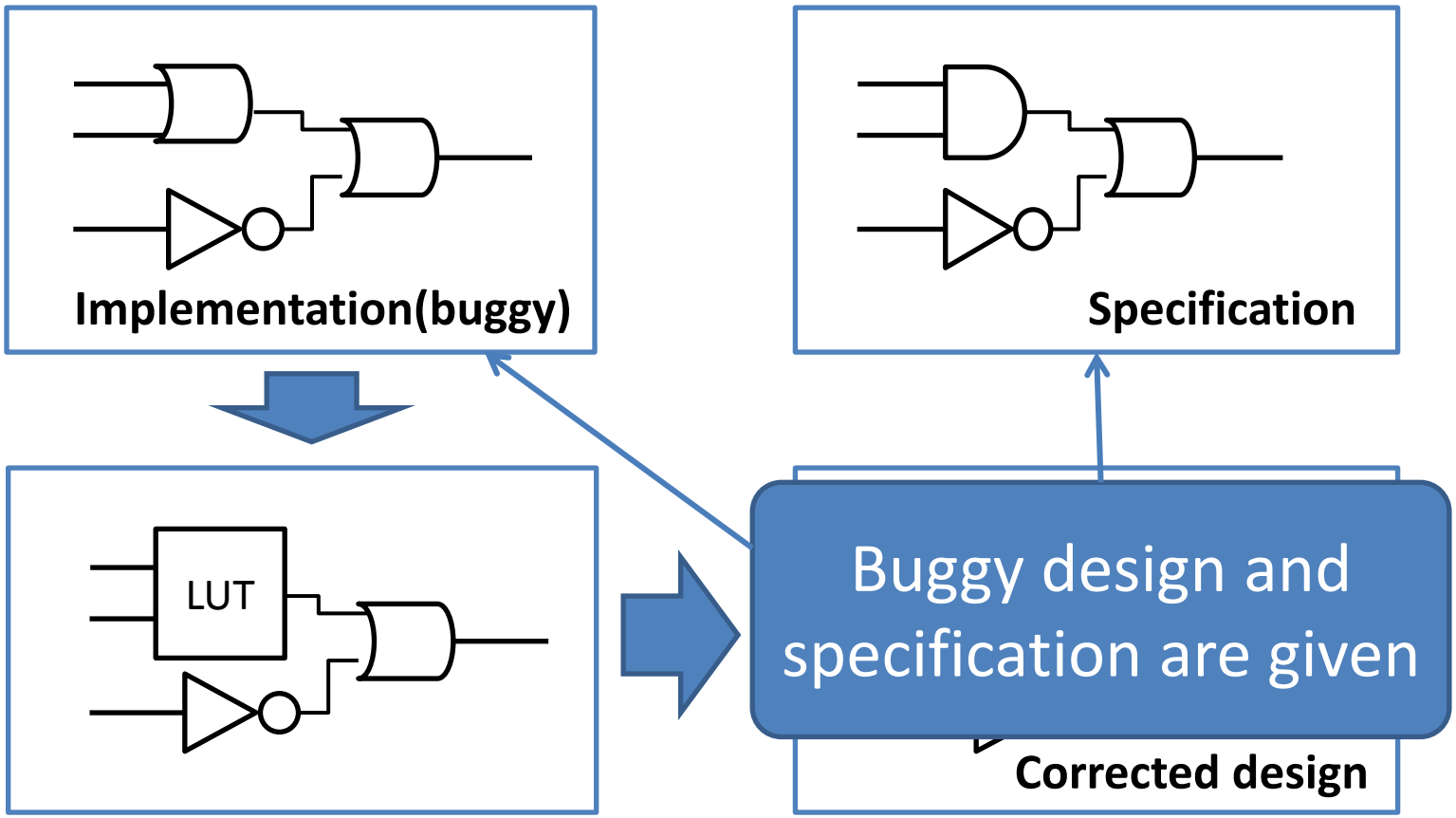
Circuit	#gates	Number of sub-circuits for FOF	Normal non-incremental SAT solvers		Incremental SAT solvers	
			Average time (in sec)	Average number of test patterns	Average time (in sec)	Average number of test patterns
c499	202	10	0.9	7.3	0.016	6.85
		20	1.9	12.6	0.049	13.6
		50	4.3	23.9	0.13	25.5
		100	9.7	37.0	0.47	48.0
c880	383	10	4.1	23.9	0.067	21.8
		20	10.7	42.8	0.29	44.3
		50	35.0	78.5	1.22	76.9
		100	97.0	121.3	6.29	107.4
c1350	546	10	5.3	20.3	0.10	21
		20	9.7	29.3	0.24	33.4
		50	31.2	57.3	0.99	58.3
		100	69.2	82.4	3.88	74.7
c1908	880	10	4.5	15.7	0.07	17.5
		20	10.6	27.5	0.23	35.6
		50	29.80	50.4	1.38	73.2
		100	73.8	78.4	7.47	103.8
c2670	1193	10	9.1	20.0	0.11	21.25
		20	25.3	36.9	0.37	42.35
		50	104.8	79.7	2.74	101.9
		100	276.2	128.5	26.8	243.1
c3540	1669	10	10.2	18.3	0.31	19.2
		20	21.6	30.0	0.63	33.75
		50	85.5	64.1	2.55	70.7
		100	240.4	107.5	9.13	127.1
c5315	2406	10	16.3	20.4	0.23	21.2
		20	37.8	34.0	0.77	35.8
		50	124.0	66.3	3.33	74.7
		100	328.7	109.4	11.5	122.7

Overview of application to debugging (1)

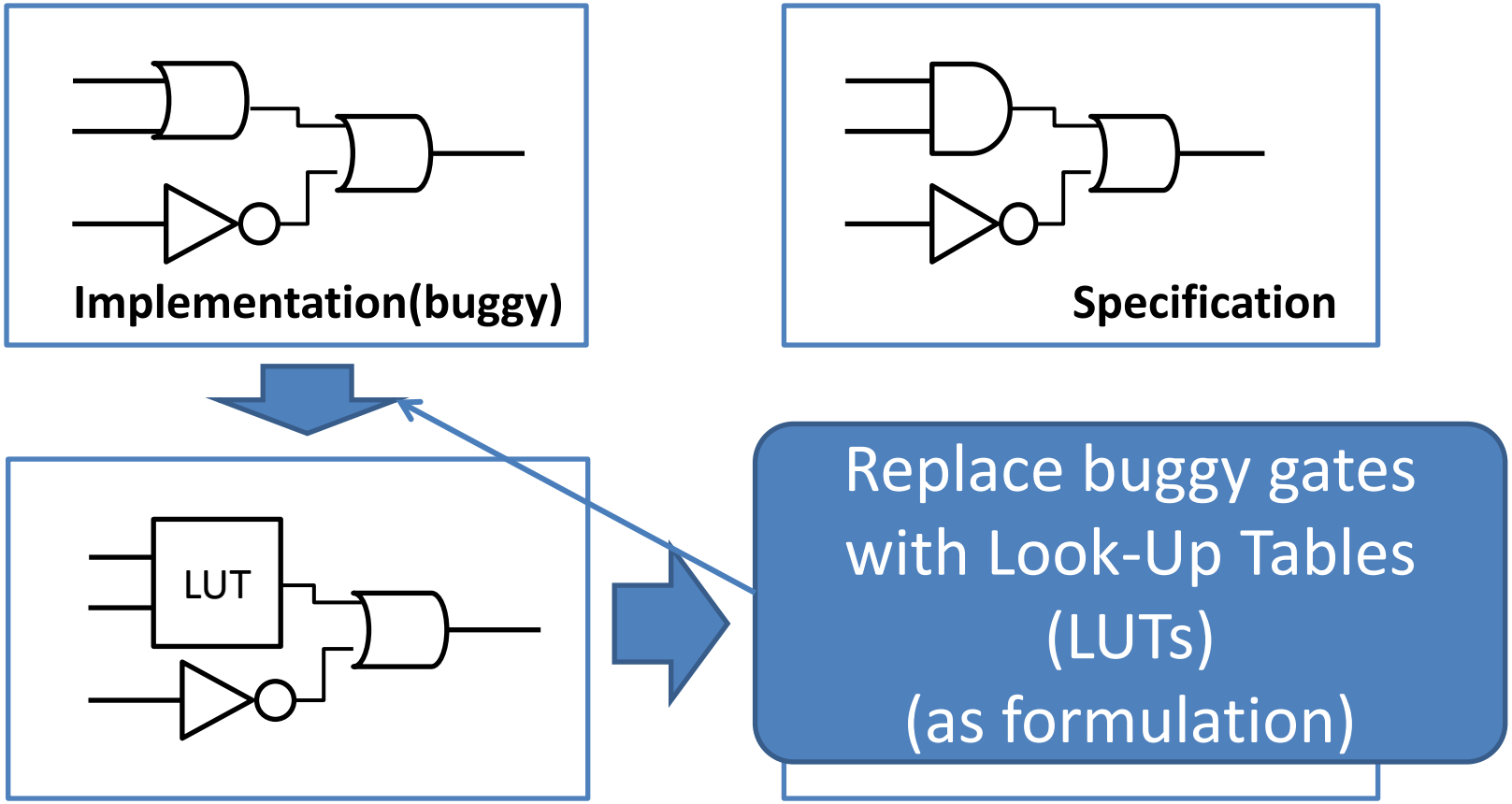
- Introducing programmability for formulation of debugging



Overview of application to debugging (2)

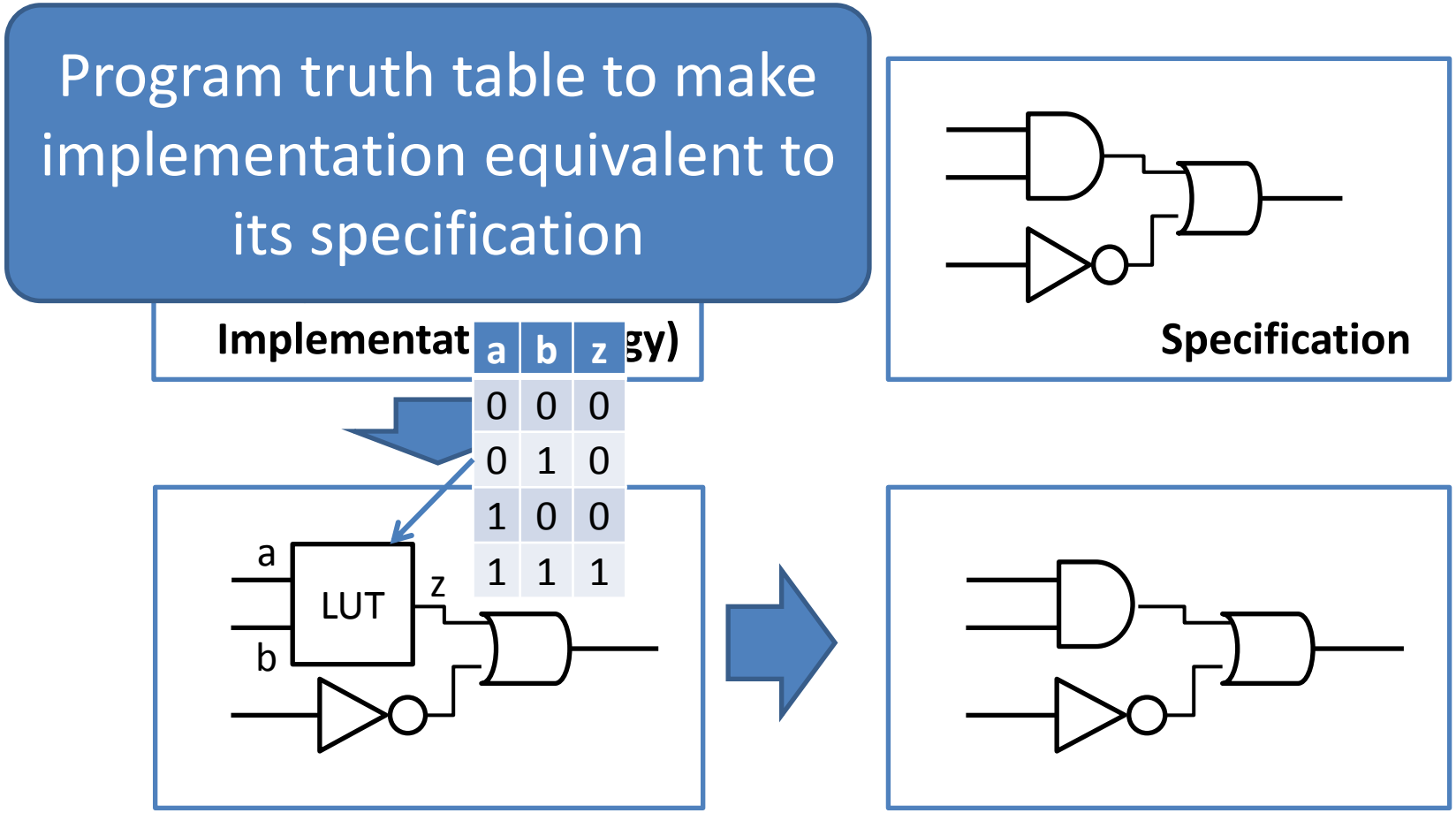


Overview of application to debugging (3)

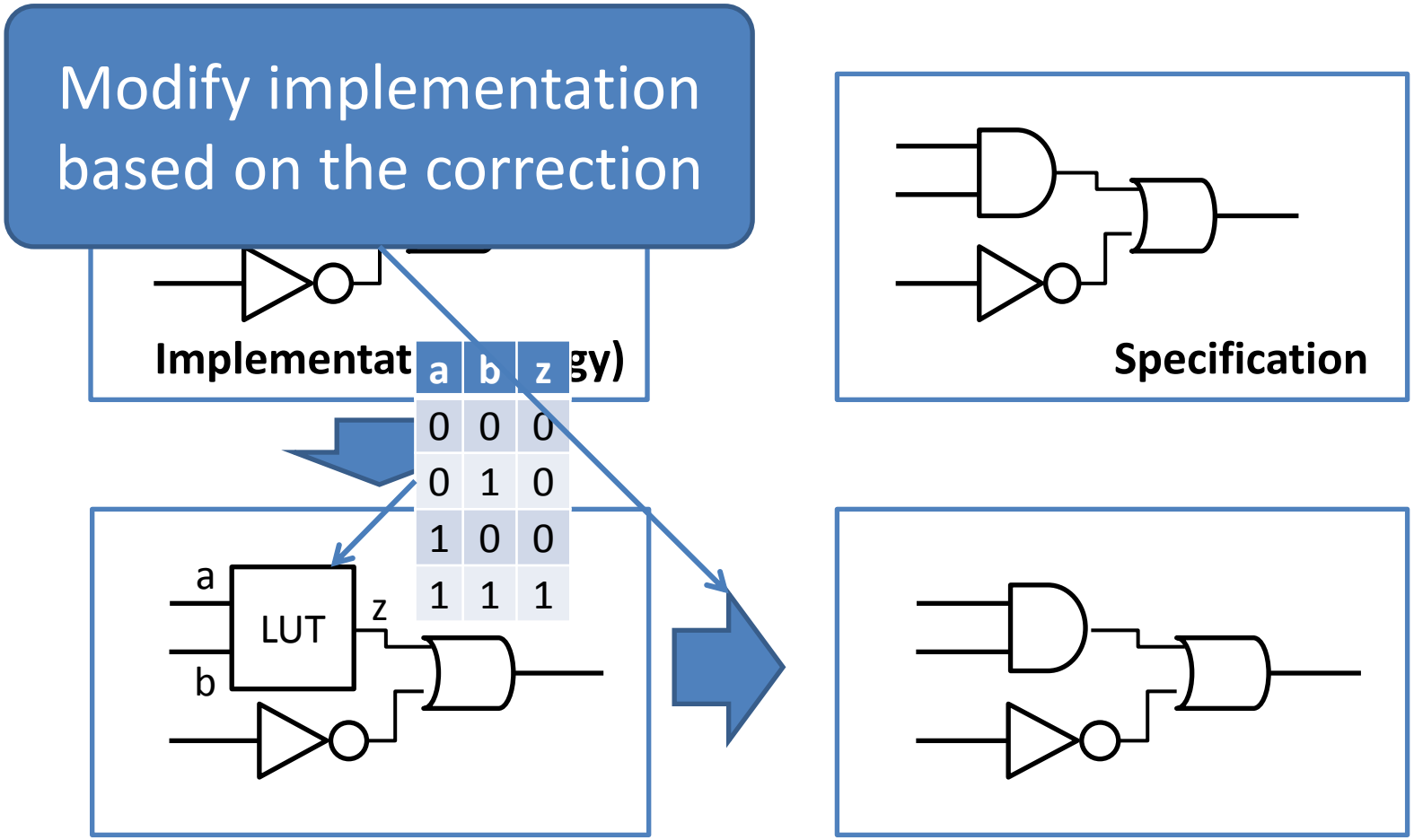


Overview of application to debugging (4)

Program truth table to make implementation equivalent to its specification



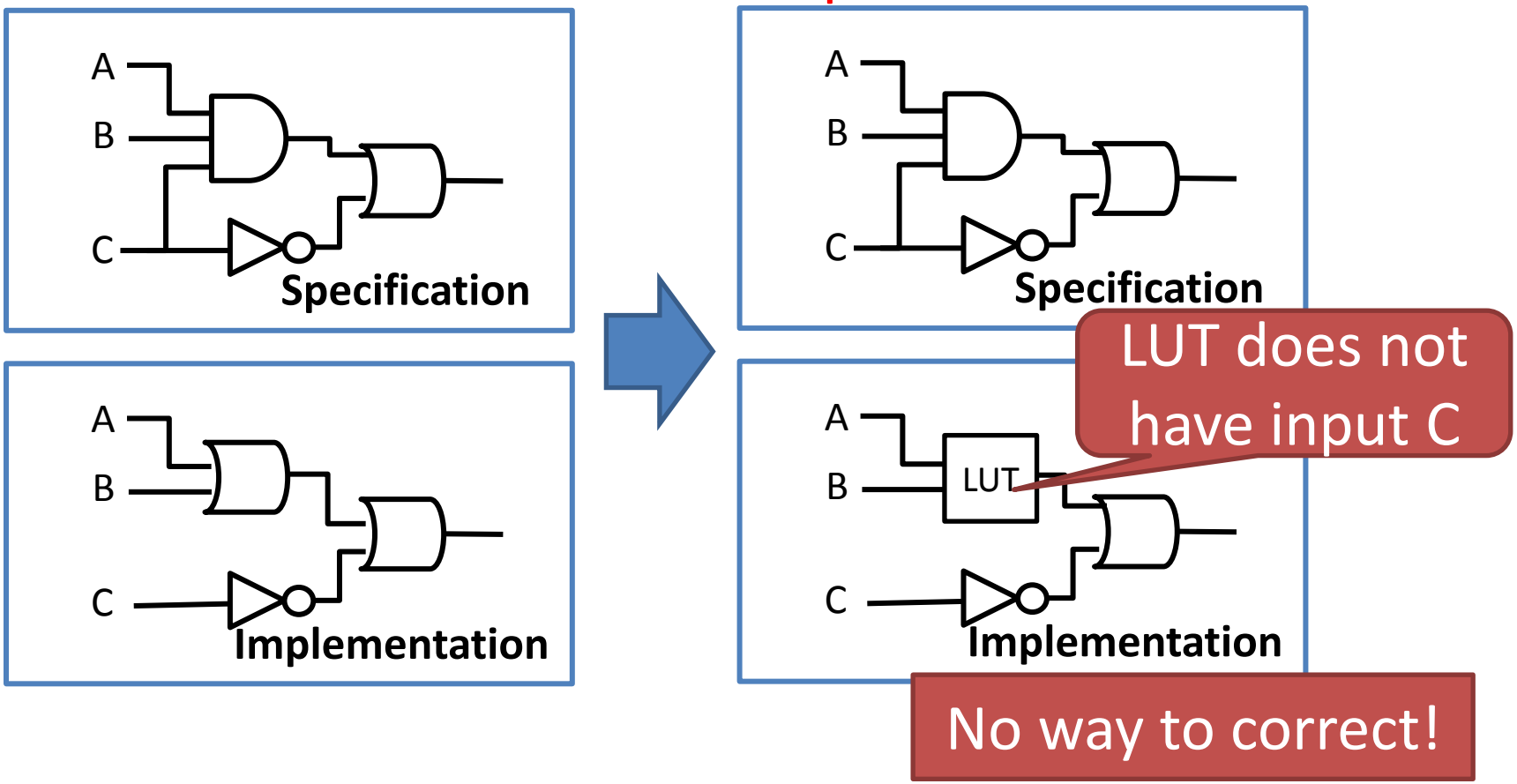
Overview of application to debugging (5)



Problem encountered when applied to industrial designs

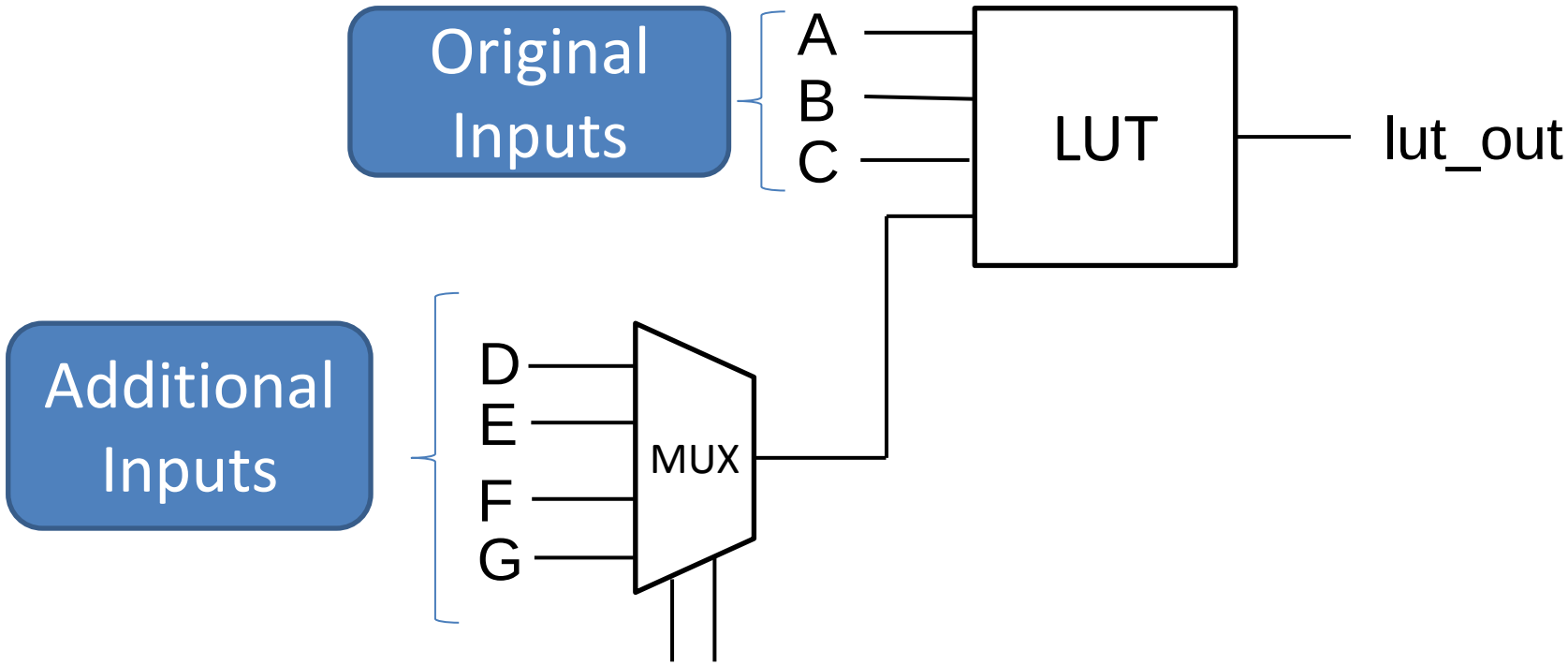
- Missing wire errors cannot be corrected by only replacing gates with LUTs

  →Need methods to add more inputs



Proposed method

- Add more variables to inputs of LUTs with MUXs
- Here, A, B, C and one of D, E, F, G are possible inputs to the LUT



Experimental Results

- Real designs in industry
- ARM processor from OpenCore
- Around 50% of bugs, additional inputs are required
- Such examples are shown below:

	Gate	Input	Input of MUX	Selected Variables	Time(Convert/SAT solving) [sec]
Industrial	8289	1201	1(no MUX)	-	Timeout >5[h]
			16	240	5281(5175/106)
			64	256	12794(12555/239)
			256	256	211(209/2)
ARM processor	4666	895	1(no MUX)	-	Timeout >5[h]
			16	128	11204(3388/7816)
			64	128	8857(1081/7771)
			256	256	5909(1324/4585)

Future perspectives

- The proposed techniques can be extended in varieties of ways:
  - For sequential circuits (other than time-frame expansion)
  - For high level designs such as word-level or C-based designs
  - For automatic assertion generations (or generation of complicated implications with small numbers of test vectors)
  - For general logic optimization of logic circuits
  - For Engineering Change Order (ECO)