Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

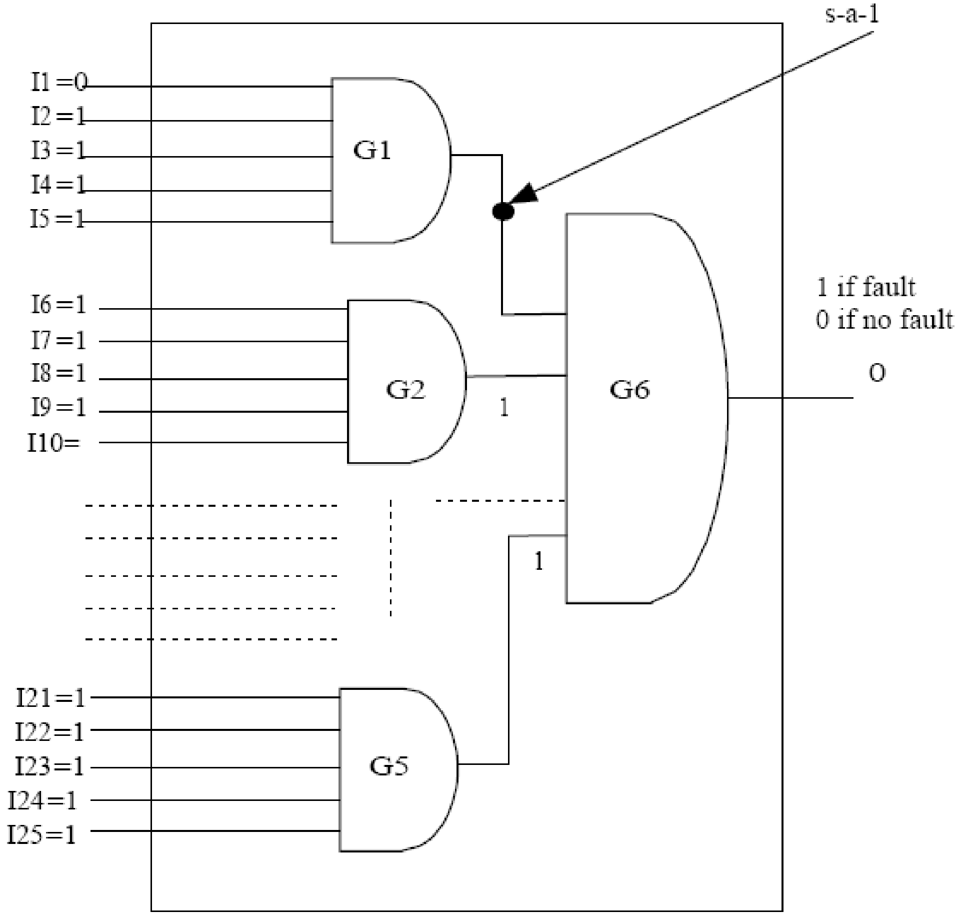
Module-VIII

Lecture-I

Fault Simulation

Introduction to Test Pattern Generation

- The procedure to generate a test pattern for a given a fault is called Test Pattern Generation (TPG). Generally TPG procedure is fully automated and called Automatic TPG (ATPG).



Introduction to Test Pattern Generation

- Fault Sensitization: Output net of G1 is stuck-at-1, we need to drive it to 0 to verify the presence/absence of the fault.
- Fault Propagation: Affect of the fault is to be propagated to a primary output (Output of G6, in this example).
- Justification: Determination of values at primary inputs so that Fault sensitization and Fault propagation are successful.

Introduction to Test Pattern Generation

Test Pattern	Test Pattern	Output
No.	I1 I2 I3 I4 I5 I6........................I25	
1	0 0 0 0 0 11111111111111111111	1 if fault 0 if no Fault
2	0 0 0 0 1 11111111111111111111	1 if fault 0 if no Fault
…………	…………………………………………….	……..
2^{25}	1 1 1 1 0 11111111111111111111	1 if fault 0 if no Fault

TPG procedure would generate any one of the patterns given in Table 1

Introduction to Test Pattern Generation

Do we require these three steps for all faults?
TPG would take significant amount of time.
However, one test pattern can test multiple faults.

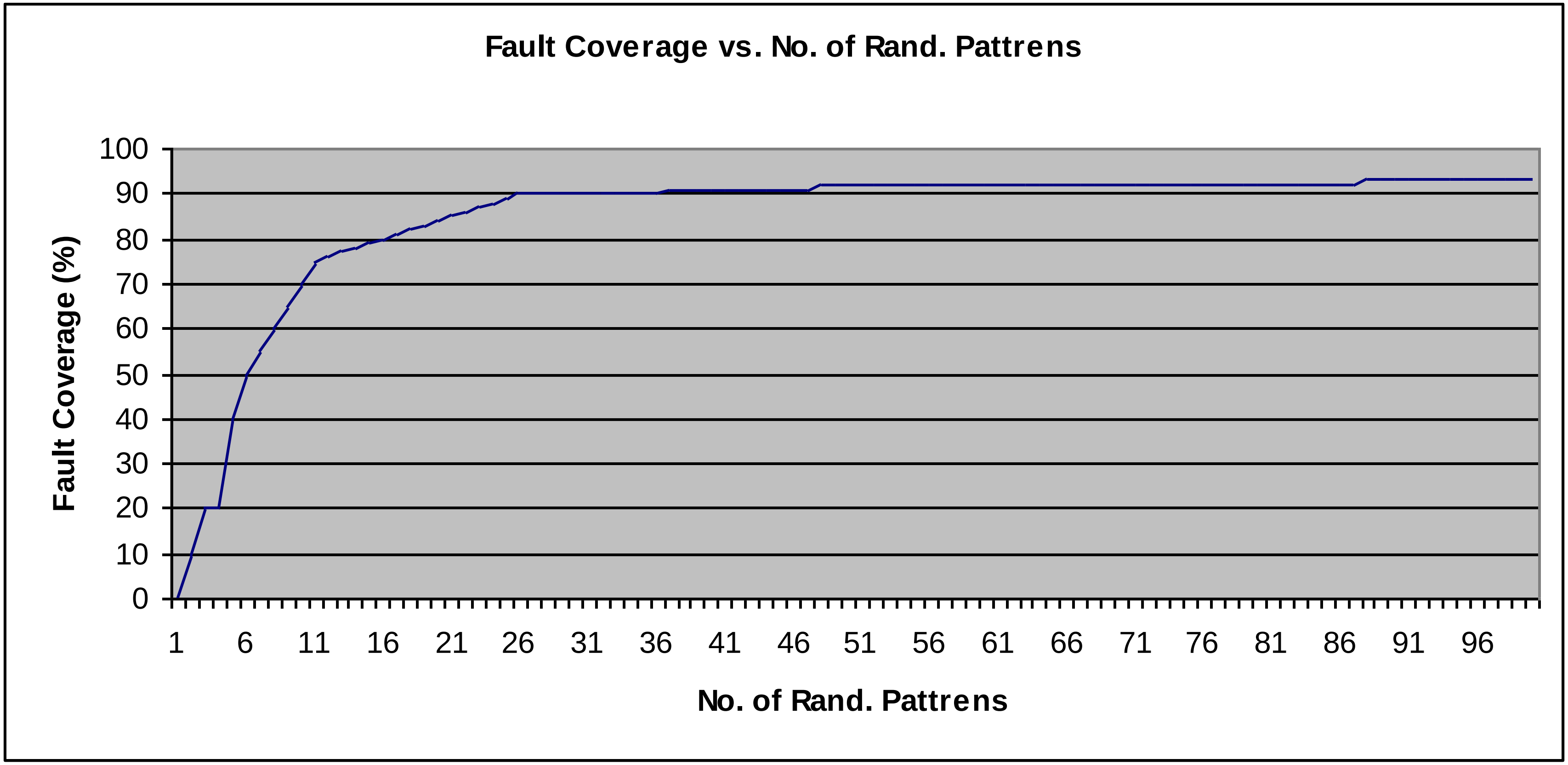
Pattern	Random Pattern		Faults Detected
No.	I1 I2 I3 I4 I5	I6……………………..I25	
1	1 0 0 0 1	11111111111111111111	s-a-1 at net “output of G1” s-a-1 at net “output of G6”
2	1 1 1 1 1	11111111111111111111	s-a-0 faults in all the nets of the circuit

On the other hand if we would have gone by the “sensitize-propagate-justify” approach these three steps would have been repeated 33 times.

Random test pattern generation

1. Generate a random pattern
2. Determine the output of the circuit for that random pattern as input
3. Take fault from the fault list and modify the Boolean functionally of the gate whose input has the fault.
   - The s-a-1 fault at the output of gate G1 modifies the Boolean functionality of gate G6 as 1 AND I2 AND I3 AND I4 AND I5 (which is equivalent to I2 AND I3 AND I4 AND I5) .
4. Determine output of the circuit with fault for that random pattern as input.
5. If the output of normal circuit varies from the one with fault, then the random pattern detects the fault under consideration.
6. If the fault is detected, it is removed from the fault list.
7. Steps 3 to 6 are repeated for another fault in the list. This continues till all faults are considered.
8. Steps 1 to 7 are repeated for another random pattern. This continues till all faults are detected.

Random test pattern generation



Typically beyond 90% fault coverage, it is difficult to find a random pattern that can test a new fault. For the remaining 10% of faults it is better to use the "sensitize-propagate-justify" approach--*difficult to test faults.*

Test pattern generation

TPG can be done in two phases

- Use random patters, till a newly added pattern detects a reasonable number of new faults

- For the remaining faults, apply “sensitize-propagate-justify” approach

Techniques to determine faults covered by random patterns, called *fault simulation*.

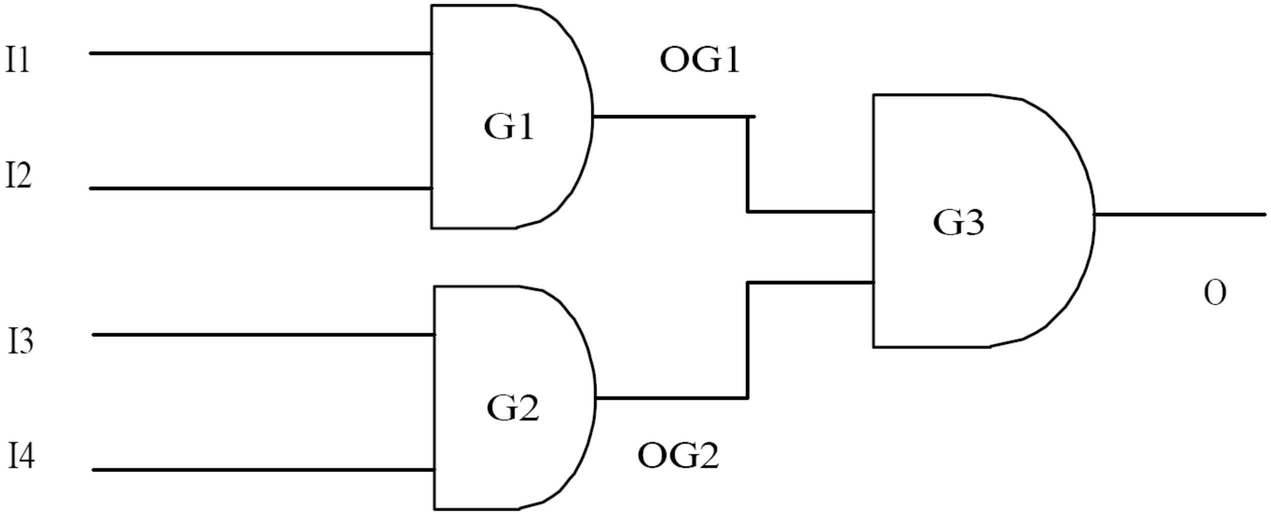
Circuit Simulation

- The imitative representation of the functioning of circuits by means of another alternative, a computer program say, is called simulation.

Compiled Code Simulation

1. Compiled Code method of simulation involves describing the circuit in a language that can be compiled and executed on a computer.
   - The circuit description can be in a Hardware Description Language such as VHDL or Verilog or simply described in C.
2. Inputs, outputs and intermediary nets are treated as variables in the code which can be Boolean, integer, etc. Gates such as AND, OR, etc., are directly converted into program statements (using bit wise operator).
3. For every input pattern, the code is repeatedly executed.

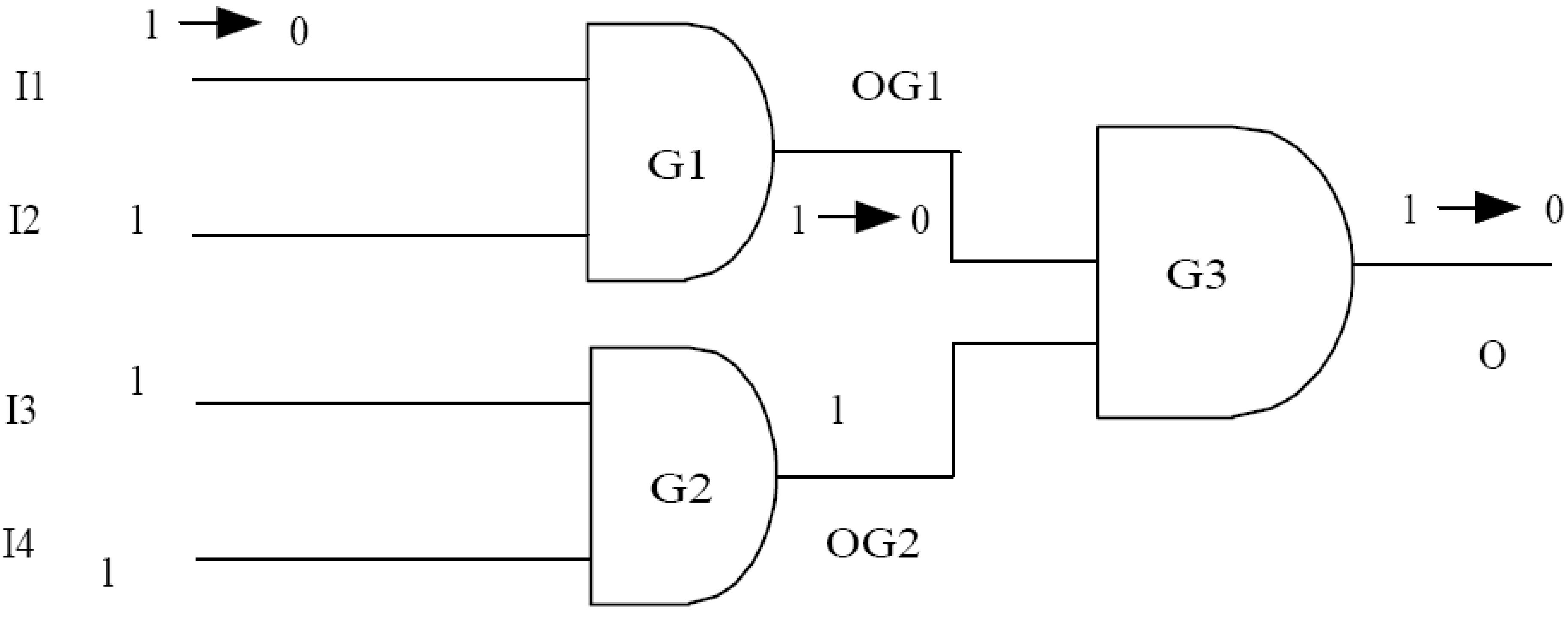
Compiled Code Simulation: Example



Compiled Code Simulation: Example

```
# include<stdio.h>
main()
{
 int I1,I2,I3,I4,OG1,OG2,O;
 printf("Input the Values of I1, I2, I3 and I4”);
scanf("%d", &I1);
scanf("%d", &I2);
scanf("%d", &I3);
scanf("%d", &I4);
  OG1 = I1 & I2;
 OG2 = I3 & I4;
O = OG1 & OG2;
 printf("\n Output of Circuit is %d",O);
}
```

Compiled Code Simulation: Signal Changes

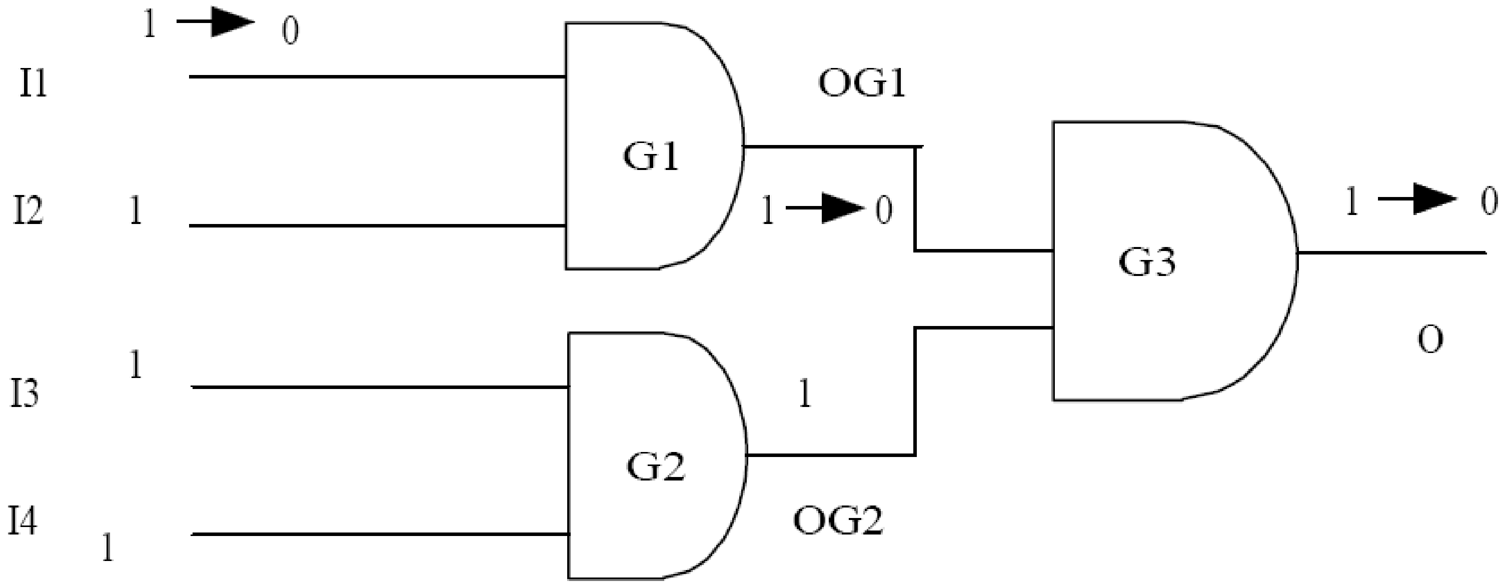


Changes in signal values in a circuit for change in input is low

Event Driven Simulation

- Event-driven simulation is a very effective scheme for circuit simulation as it is based on detection of any signal change (event) to trigger other signal(s).
- An event triggers new events, which in turn may trigger more events; the first trigger is a change in primary input.

Event Driven Simulation: Example

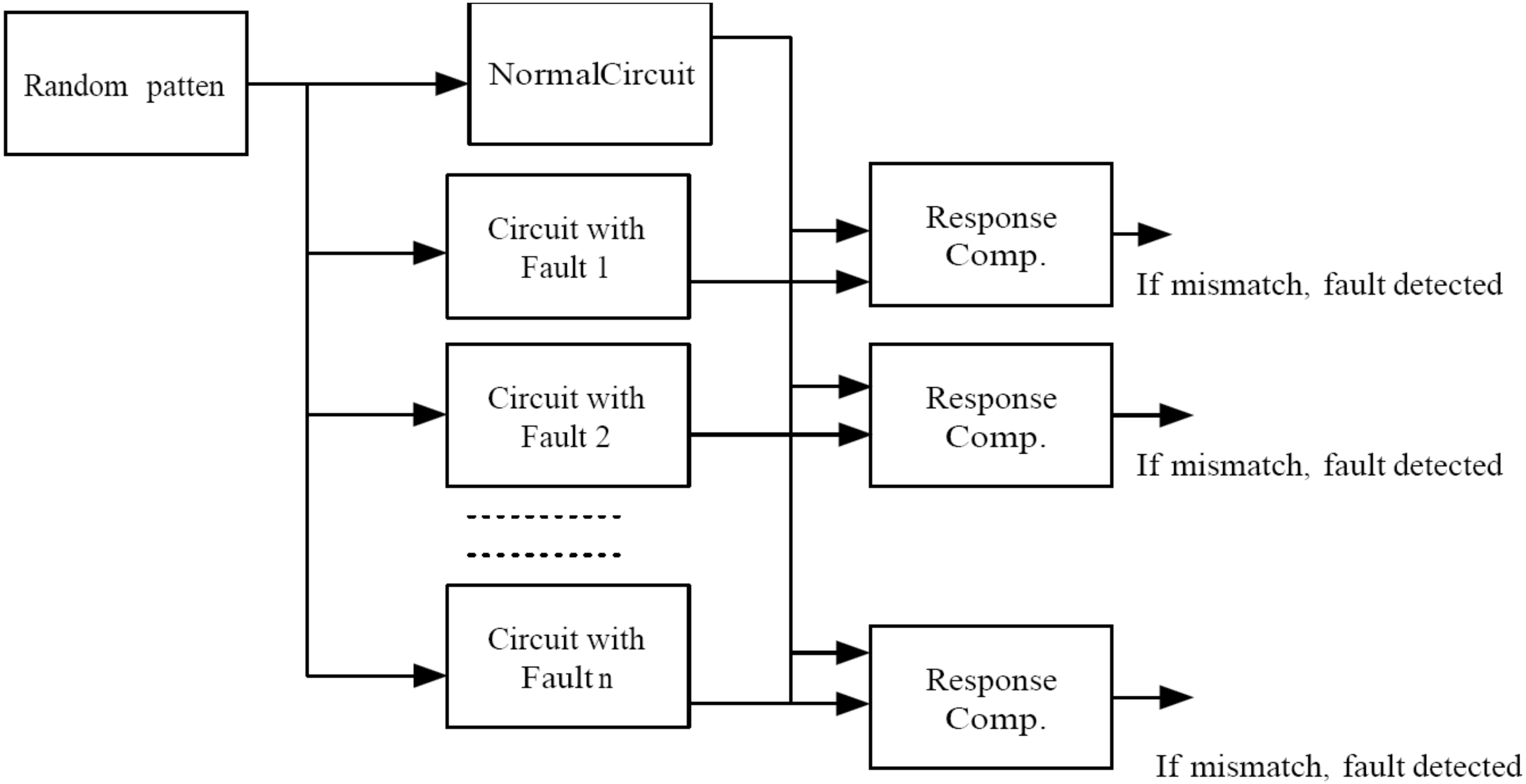


Time	Scheduled Events	Activity List
t=0	I1 =0	OG1
t=1	OG1=0	O
t=2	O=0	

Fault Simulation

- A fault simulator is like an ordinary simulator, but needs to simulate two versions of a circuit
  - without any fault for a given input pattern and
  - with a fault inserted in the circuit and for the same input pattern.
- If the outputs under normal and faulty situation differ, the pattern detects the fault.
- Step (ii) is repeated for all faults. Once a fault is detected it is dropped

Fault Simulation



The procedure is simple, but is too complex in terms of time required. Time required is

$$\sum_{i=i}^{no\ of\ random\ patterns} faults\ for\ i^{th}\ random\ pattren \times simulation\ time\ .$$

Improving Fault Simulation Algorithms

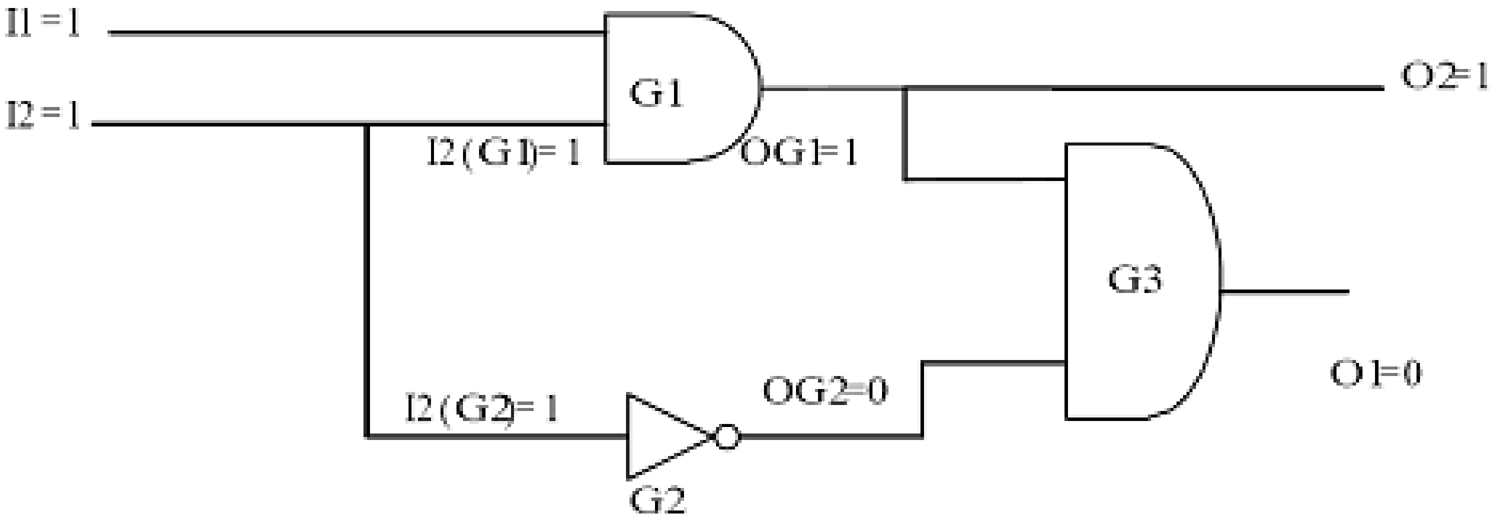
- Determine more than one fault that is detected by a random pattern during one simulation run

- Minimal computations when the input pattern changes; the motivation is similar to event driven simulation over complied code simulation.

Serial Fault Simulation

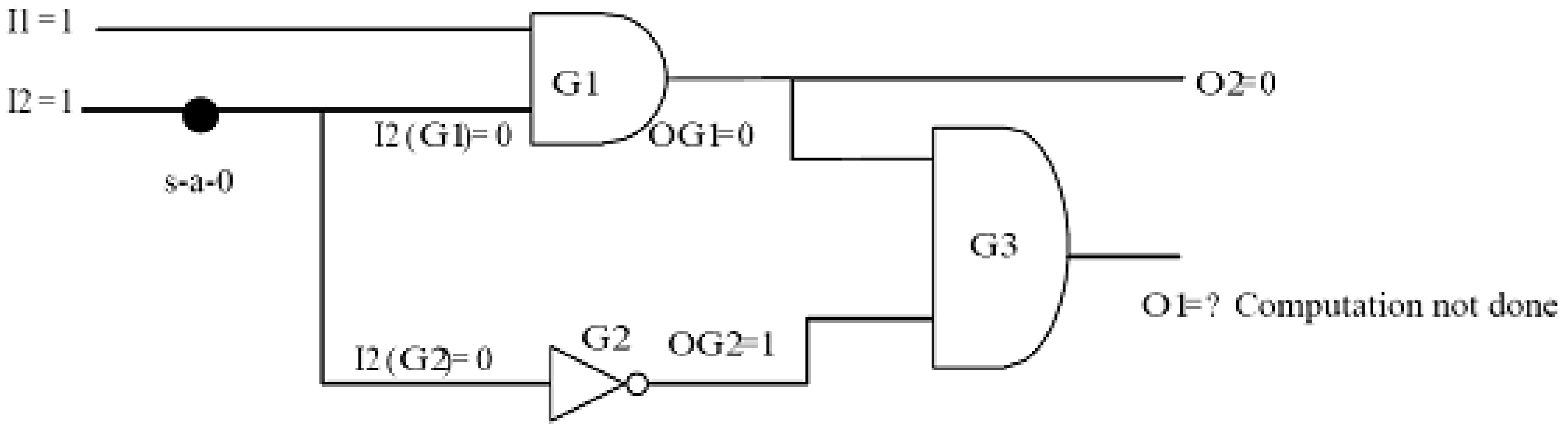
1. The circuit is first simulated (using event driven simulator) without any fault for a random pattern and primary output values are saved in a file.
2. Next, faults are introduced one by one in the circuit and are simulated for the same input pattern. This is done by modifying the circuit description for a target fault and then using event driven simulator.
3. The output values (at different primary outputs) of the faulty circuit are compared with the saved true responses. The simulation of a faulty circuit halts when output value at any primary output differs for the corresponding normal circuit response. All faults detected are dropped and the procedure repeats for a new random pattern.

Serial Fault Simulation: Example



Normal Condition

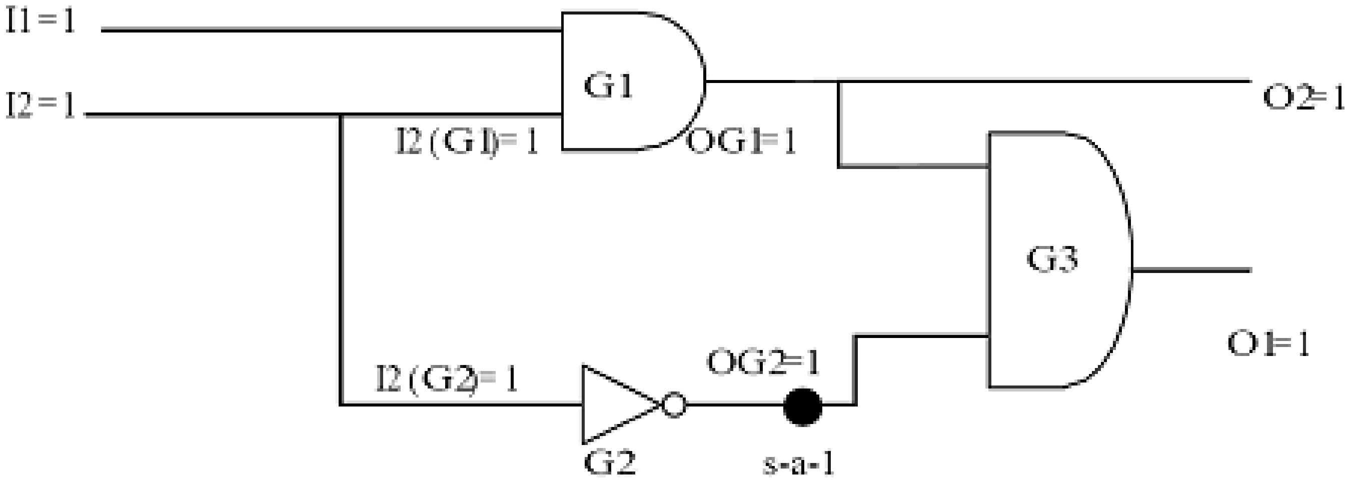
Serial Fault Simulation: Example



s-a-0 fault in I2 Condition

Time	Scheduled Event	Activity List
t=0	I1=1, I2=1	I2(G1), OG1,I2(G2),
t=1	I2(G1)=0,I2(G2)=0	OG1, OG2
t=2	OG1=0, OG2=1	O1,I1(G3),O2
t=3	O2=0 in faulty situation while O2=1 in normal condition	

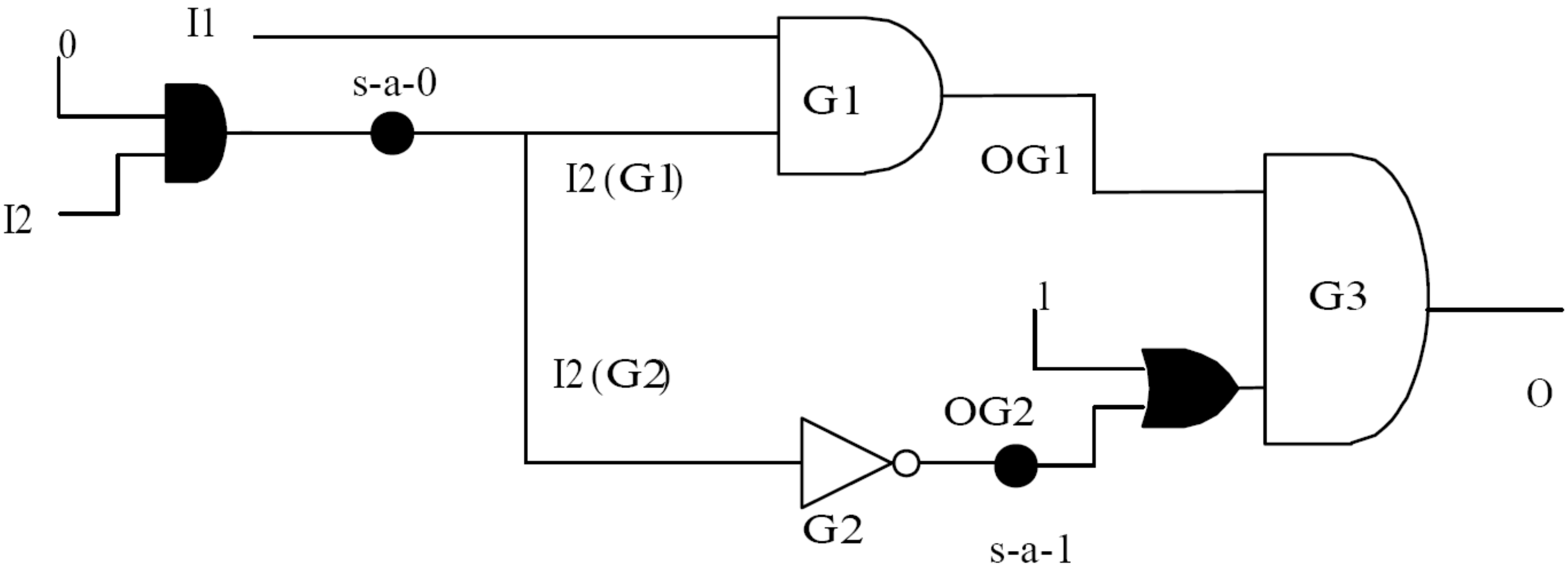
Serial Fault Simulation: Example



s-a-1 fault in OG2 Condition

Time	Scheduled Event	Activity List
t=0	I1=1, I2=1	I2(G1), OG1,I2(G2)
t=1	I2(G1)=1,I2(G2)=1	OG1, OG2
t=2	OG1=1,OG2=1	O1, I1(G3),O2
t=3	O2=1, I1(G3)=1	O1
t=4	O1=1 in faulty situation while O1=0 in normal condition	

Insertion of faults in the circuit fault simulation in event driven simulator



Thank you

Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

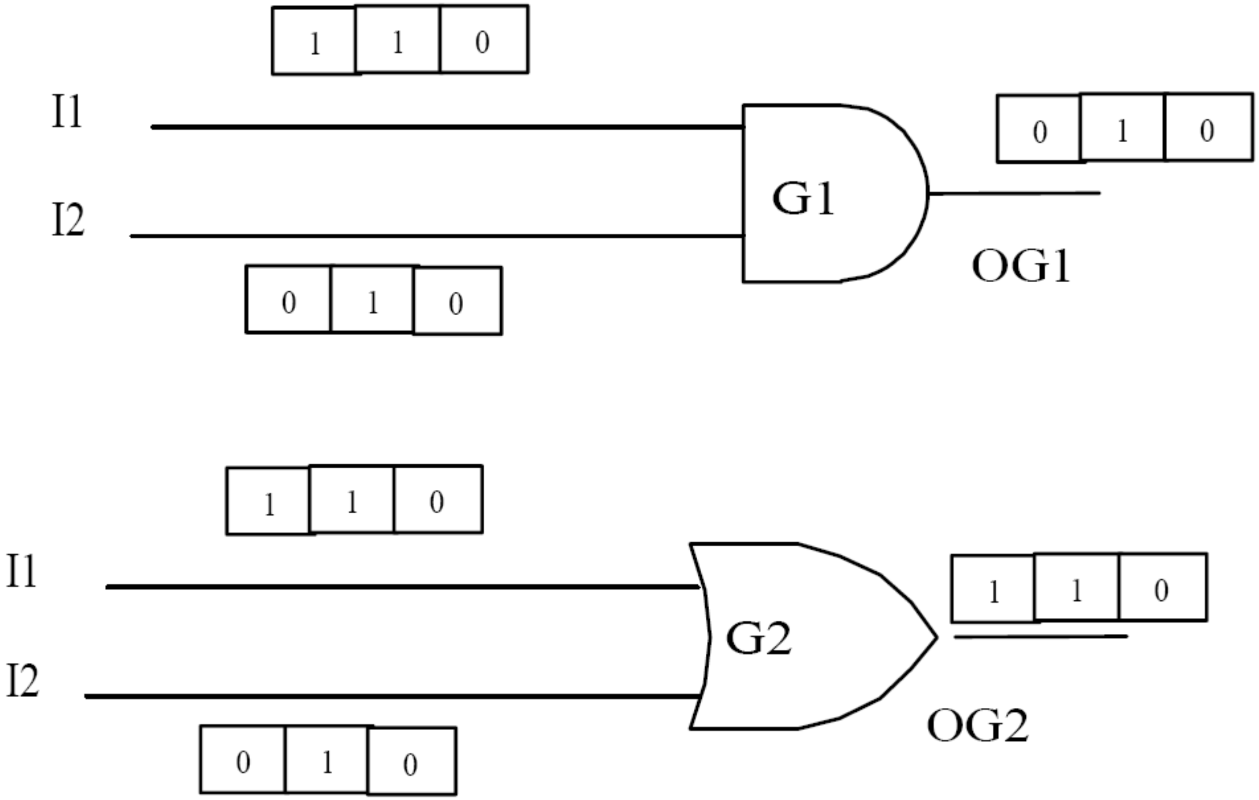
Module-VIII

Lecture-II

Fault Simulation

Introduction to Parallel Fault Simulation

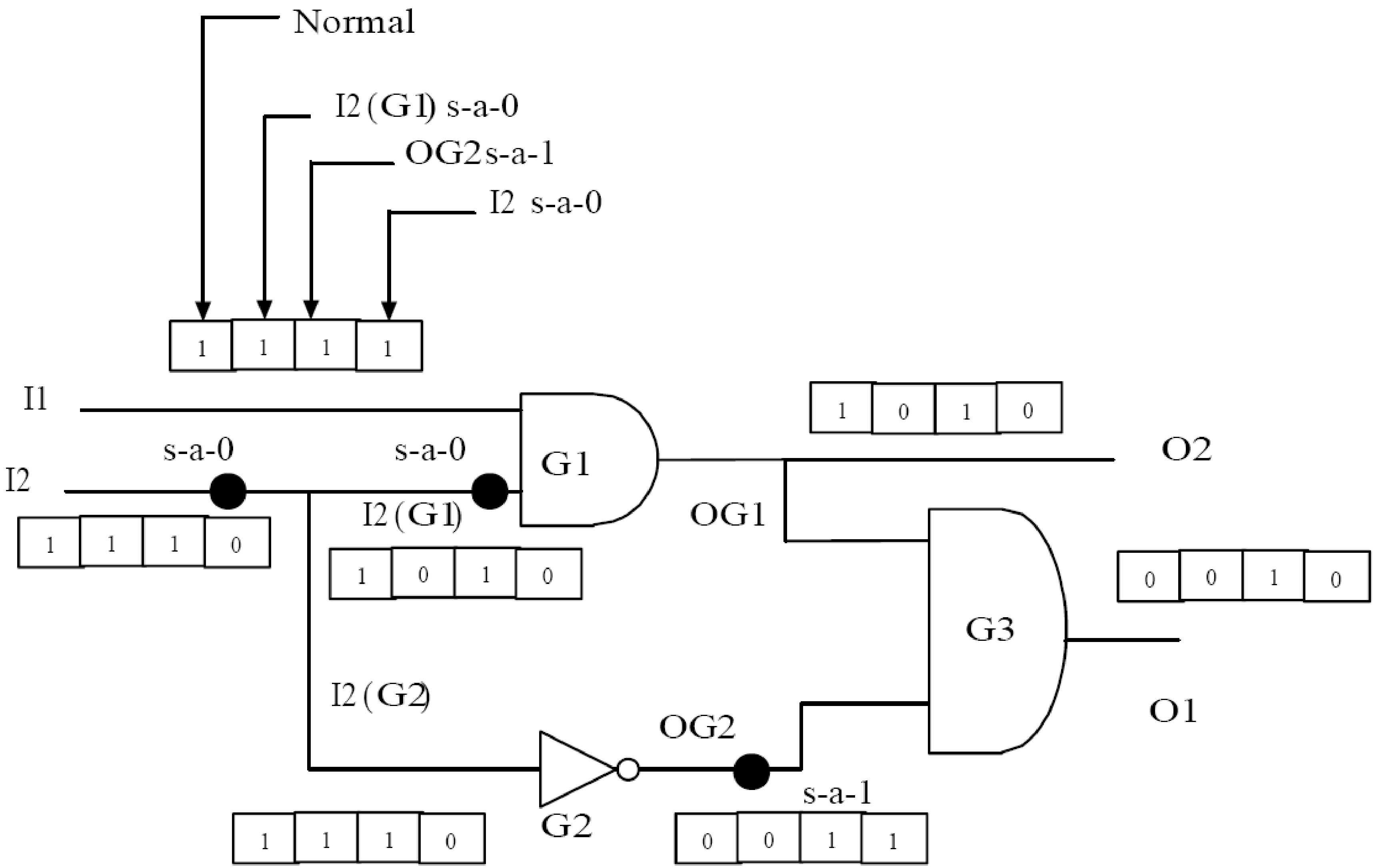
- Parallel fault simulation can processes more than one fault in one pass of the circuit simulation.
- Uses bit-parallelism of a computer.



Introduction to Parallel Fault Simulation

- – Input lines for any gate comprise binary words of length *w* (instead of single bits) and output is also a binary word of length *w.*
- – The output word is determined by simple logical operation (corresponding to the gate) on the individual bits of the input words.

Parallel Fault Simulation: Example



Parallel Fault Simulation: Example

The array at O2 is 1010. It implies that on the input I1=1 and I2=1

- O2 is 1 under normal condition
- O2 is 0 under s-a-0 fault at I2(G1),
- O2 is 1 under s-a-1 fault at OG2,
- O2 is 0 under s-a-0 fault at I2.

Pattern I1=1,I2=1 at output O2 can detect s-a-0 fault at I2(G1) and s-a-0 fault at I2 but cannot detect s-a-1 fault at OG2.

Parallel Fault Simulation: Example

The array at O1 is 0010. It implies that on the input I1=1 and I2=1

- O1 is 0 under normal condition
- O1 is 0 under s-a-0 fault at I2(G1),
- O1 is 1 under s-a-1 fault at OG2,
- O1 is 0 under s-a-0 at I2.

I1=1,I2=1 at output O1 detects s-a-1 fault at OG2. So all the three faults are detected by I1=1,I2=1.

Parallel Fault Simulation

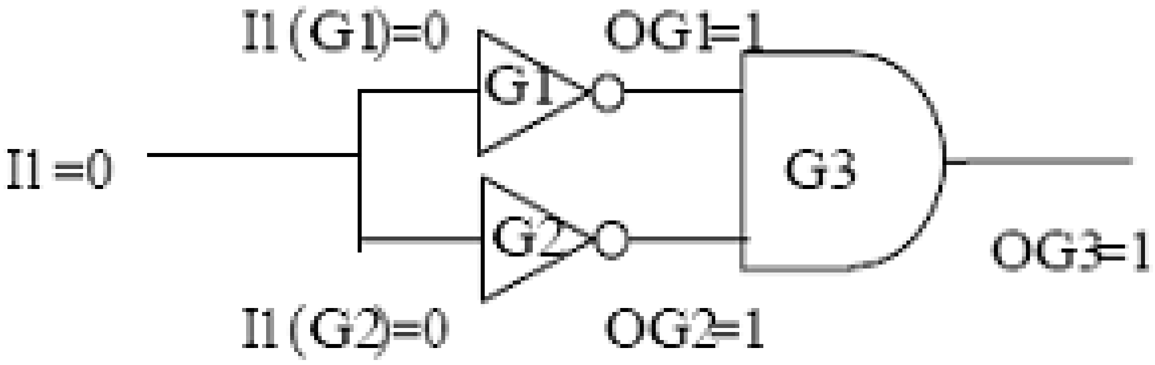
- Parallel fault simulation speeds up the serial fault simulation scheme by *w-1* times.
- After an iteration of parallel fault simulation, next set of *w-1* faults are considered and the procedure repeated. After all the faults are considered (i.e., *total number of faults/(w-1)* iterations) the ones detected by the random pattern are dropped.
- Next another random pattern is taken and a new set of iterations are started.

Parallel fault simulation speeds up serial fault simulation *w-1* times, but for a random pattern more than one iterations may be required.

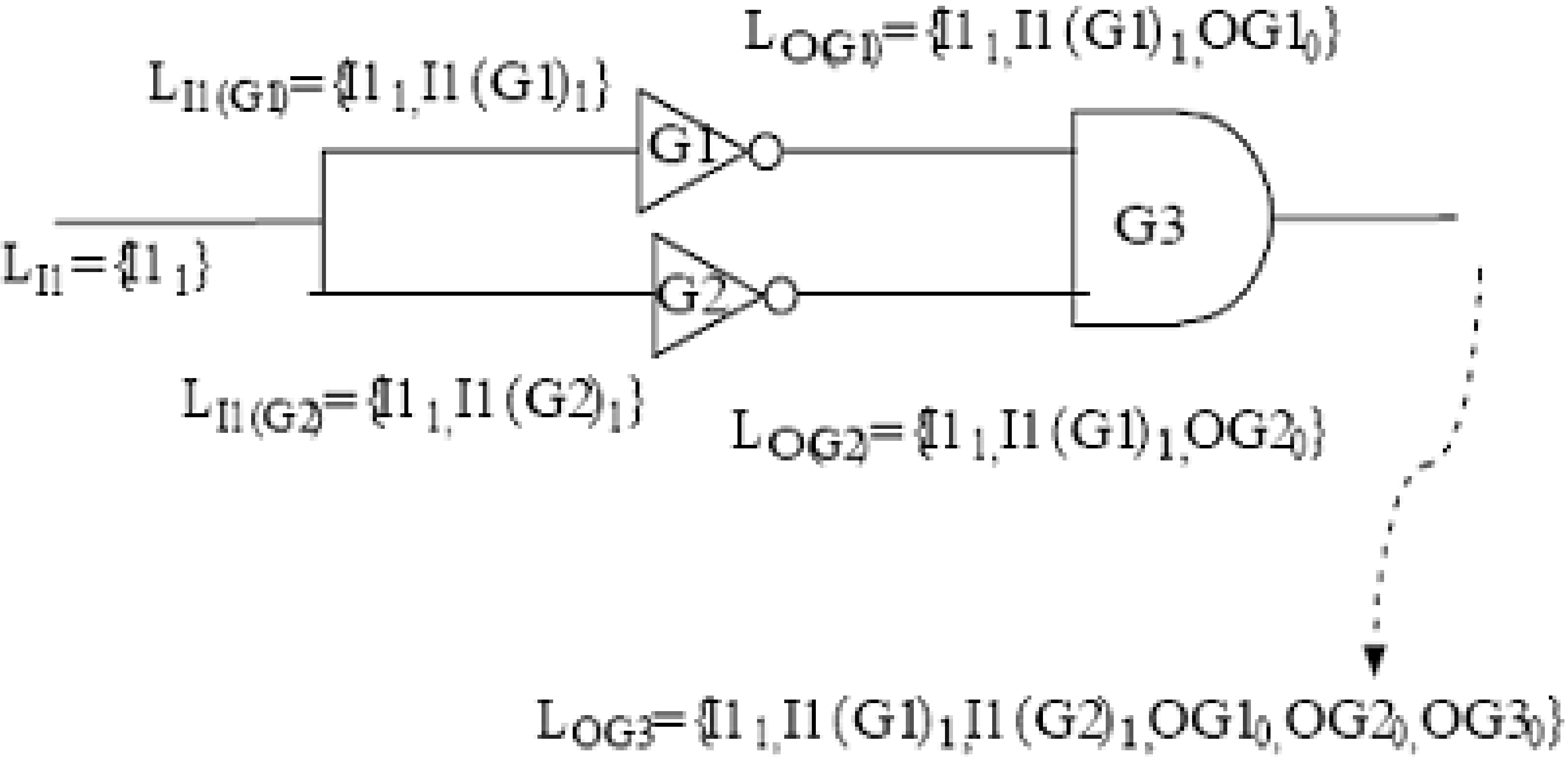
Introduction to Deductive Fault Simulation

- A procedure which can determine in a single iteration, detectability/undetectability about all faults by a given random pattern.
- First the fault-free circuit is simulated by a random pattern and all the nets are assigned the corresponding signal values.
- "Deductive", as the name suggests, all faults detectable at the nets are determined using the structure of the circuit and the signal values of the nets.
  - Circuit structure remains the same for all faulty circuits, all deductions are carried out simultaneously.
- Once detectability of all the faults for a random pattern is done, the same procedure is repeated for the next random pattern after eliminating the covered faults.

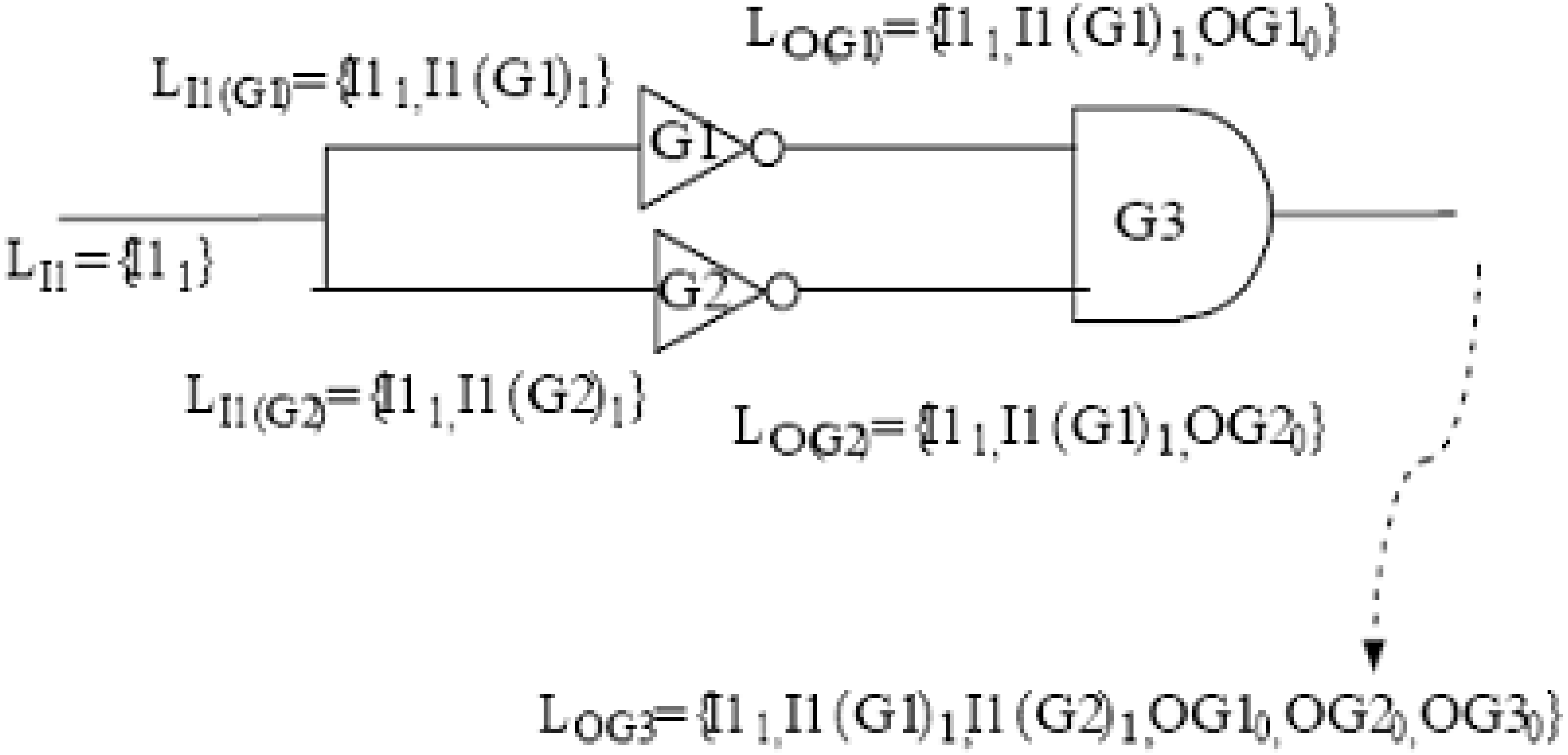
Deductive Fault Simulation: Example



Deductive Fault Simulation: Example


$L_{OG1}=\{I1_1, I1(G1)_1, OG1_0\}$
$L_{I1(G1)}=\{I1_1, I1(G1)_1\}$
G1
G3
$L_{I1}=\{I1_1\}$
G2
$L_{I1(G2)}=\{I1_1, I1(G2)_1\}$
$L_{OG2}=\{I1_1, I1(G1)_1, OG2_0\}$
$L_{OG3}=\{I1_1, I1(G1)_1, I1(G2)_1, OG1_0, OG2_0, OG3_0\}$

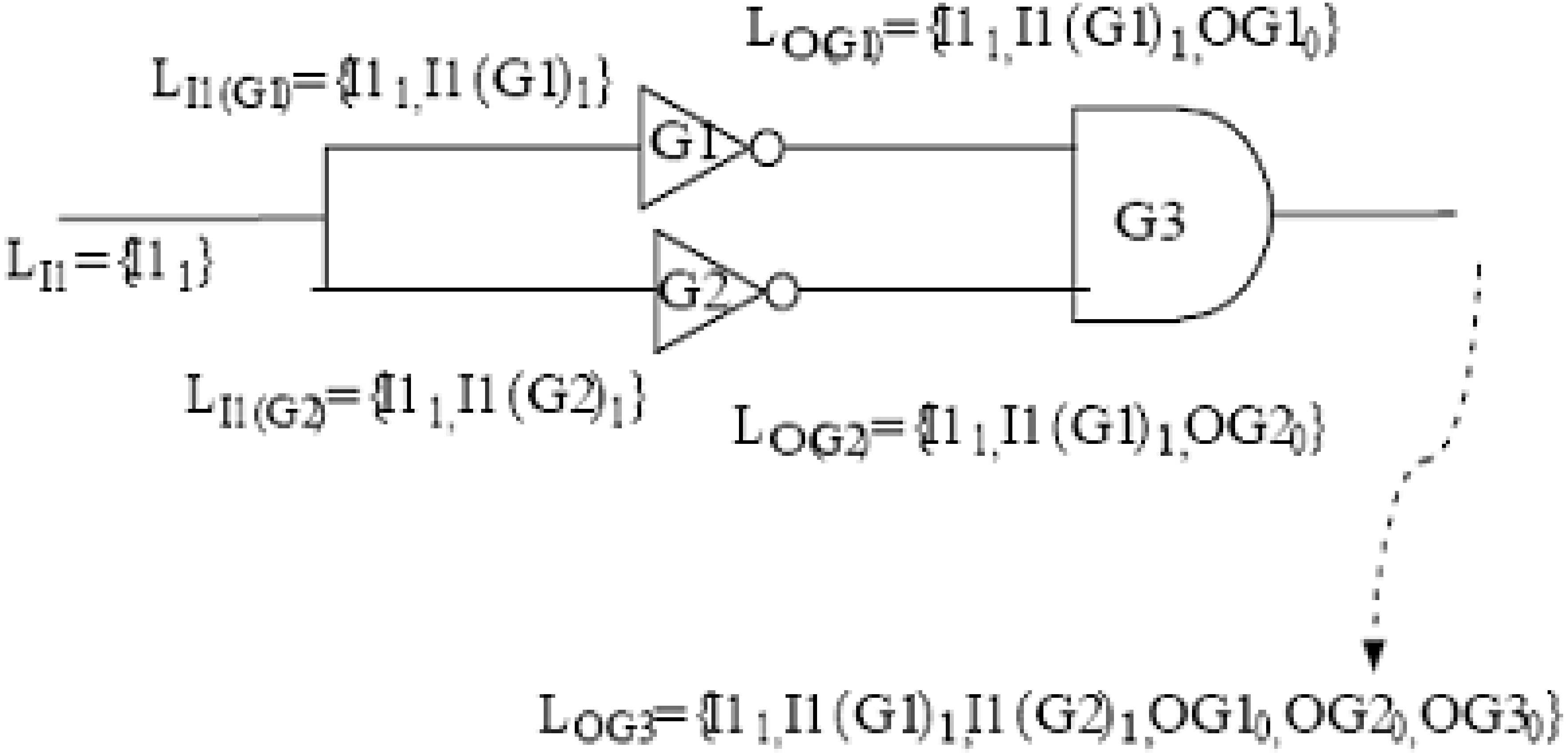
Step-1: $L_{I1}=\{I1_1\}$. s-a-1 at I1 can be detected at I1 as input pattern is I1=0.


$L_{OG1}=\{I1_1, I1(G1)_1, OG1_0\}$
$L_{I1(G1)}=\{I1_1, I1(G1)_1\}$
G1
G3
$L_{I1}=\{I1_1\}$
G2
$L_{I1(G2)}=\{I1_1, I1(G2)_1\}$
$L_{OG2}=\{I1_1, I1(G1)_1, OG2_0\}$
$L_{OG3}=\{I1_1, I1(G1)_1, I1(G2)_1, OG1_0, OG2_0, OG3_0\}$

Step-2: $L_{I1(G1)}=\{ I1_{1,}\ I1(G1)_1\}$

fault deduction at fanout branches

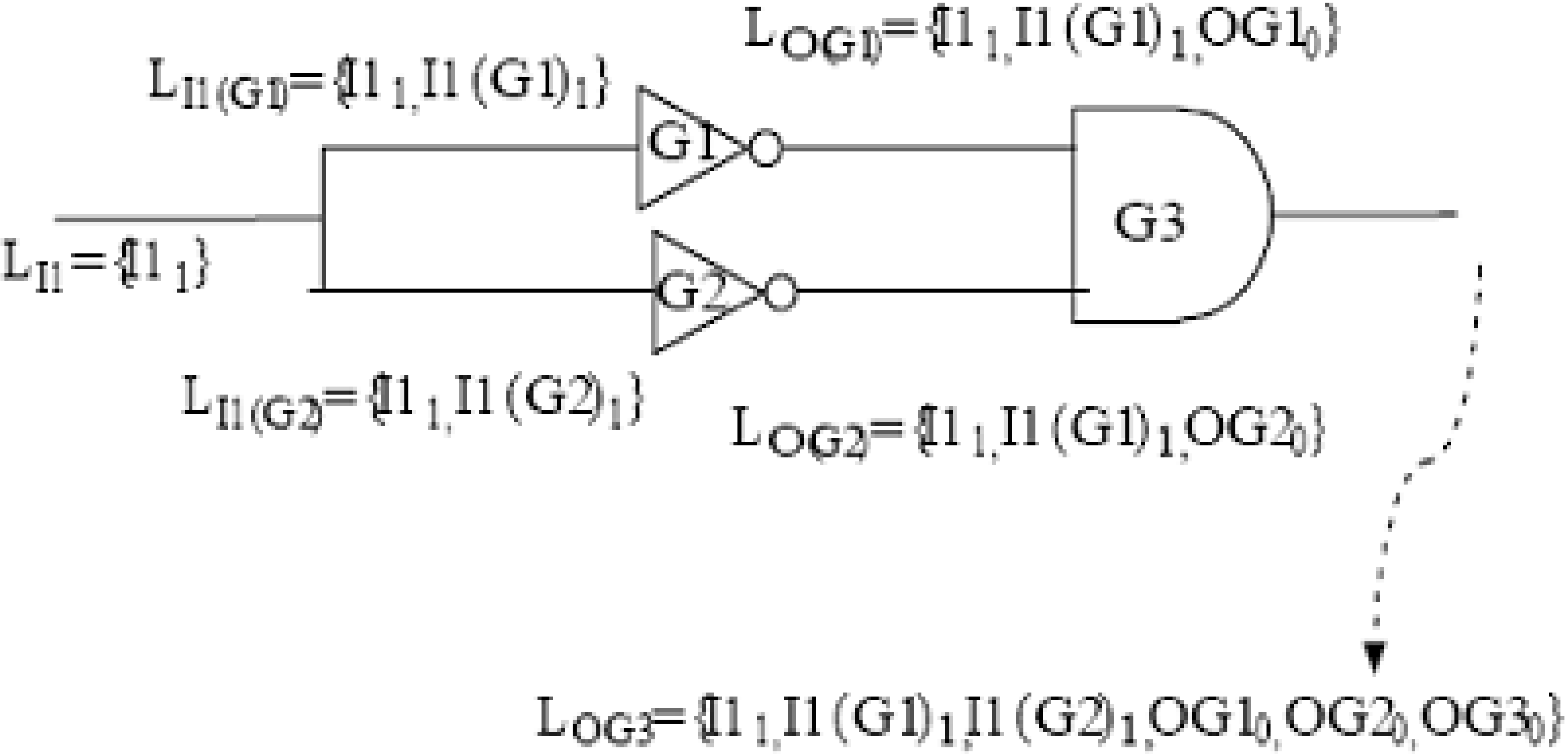
Fault list detected at fanout branch comprise (i) all faults at the fanout stem and (ii) s-a-1 if the signal value at the branch is 0, else s-a-0.


$L_{OG1}=\{I1_1, I1(G1)_1, OG1_0\}$
$L_{I1(G1)}=\{I1_1, I1(G1)_1\}$
G1
G3
$L_{I1}=\{I1_1\}$
G2
$L_{I1(G2)}=\{I1_1, I1(G2)_1\}$
$L_{OG2}=\{I1_1, I1(G1)_1, OG2_0\}$
$L_{OG3}=\{I1_1, I1(G1)_1, I1(G2)_1, OG1_0, OG2_0, OG3_0\}$

Step-3A: L_{OG1}={ $I1_{1,}$ $I1(G1)_{1,}OG1_0$}.

fault deduction at inverter

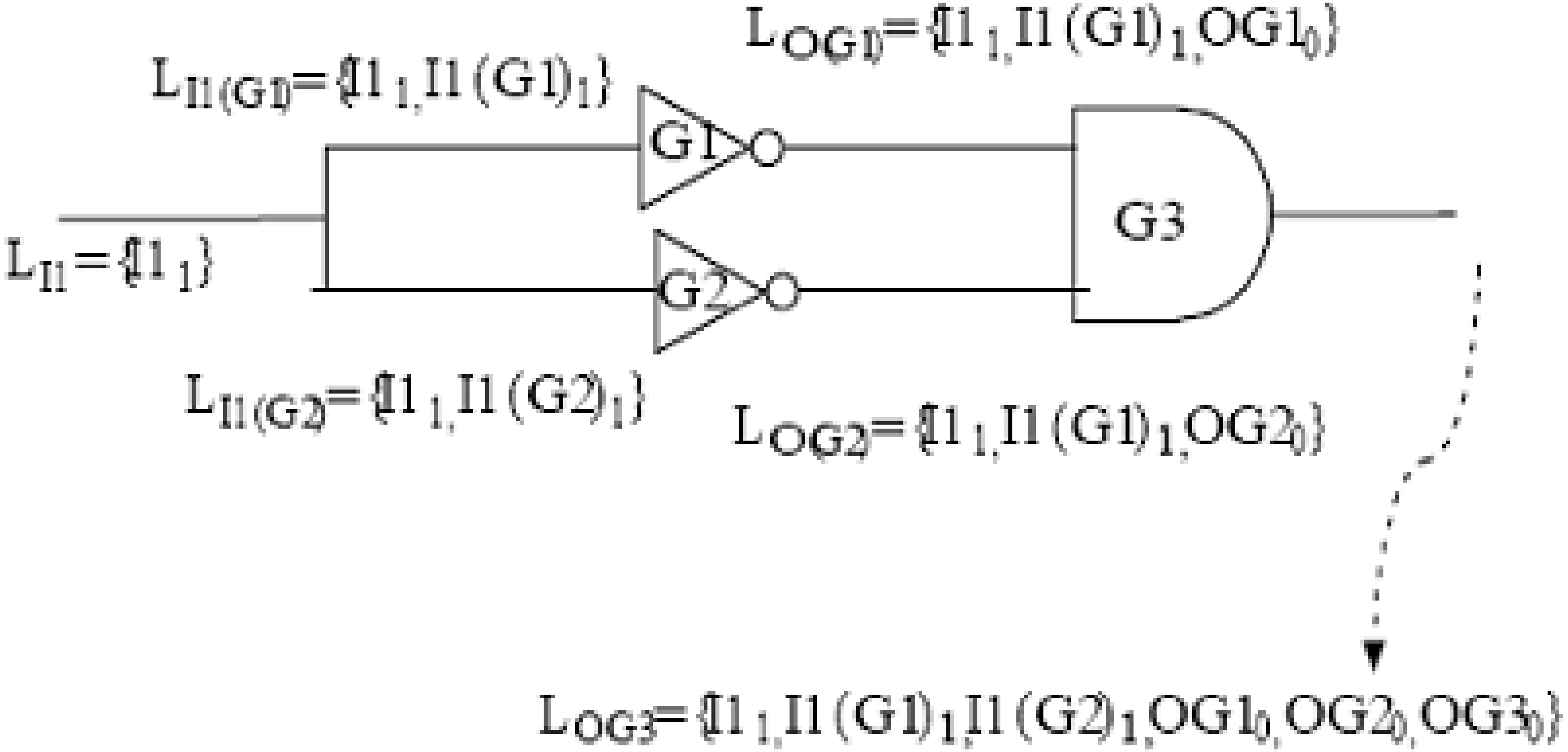
Fault list detected at output of an inverter comprise (i) all faults at the input and (ii) s-a-1 of the signal value at the output is 0, else s-a-0.


$L_{OG1}=\{I1_{1,}I1(G1)_{1,}OG1_0\}$
$L_{I1(G1)}=\{I1_{1,}I1(G1)_1\}$
G1
G3
$L_{I1}=\{I1_1\}$
G2
$L_{I1(G2)}=\{I1_{1,}I1(G2)_1\}$
$L_{OG2}=\{I1_{1,}I1(G1)_{1,}OG2_0\}$
$L_{OG3}=\{I1_{1,}I1(G1)_{1,}I1(G2)_{1,}OG1_{0,}OG2_{0,}OG3_0\}$

Step-3B: L_{OG2}={ $I1_{1,}$ $I1(G2)_{1,}OG2_0$}.

fault deduction at inverter

Fault list detected at output of an inverter comprise (i) all faults at the input and (ii) s-a-1 of the signal value at the output is 0, else s-a-0.

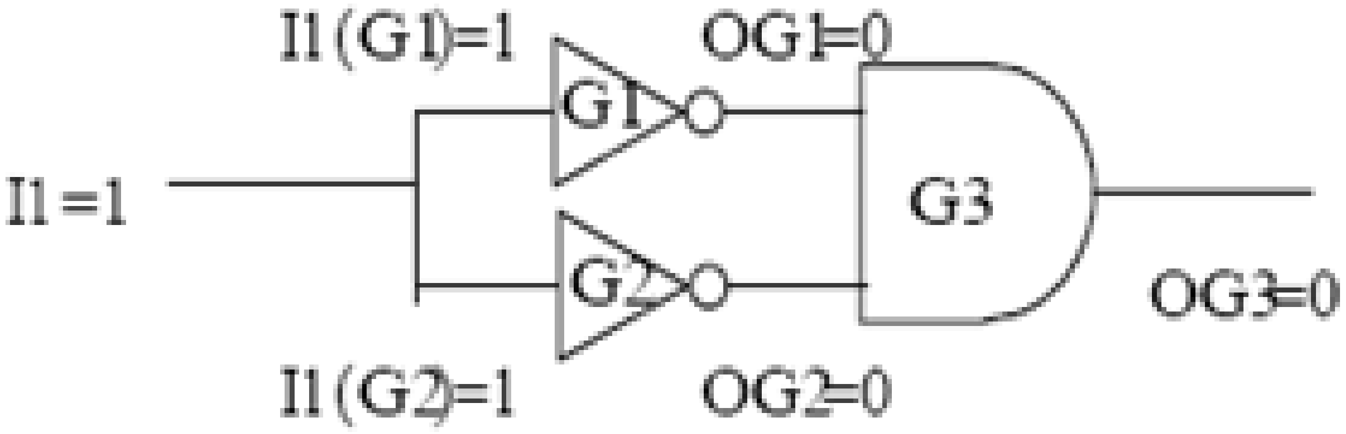

L_{OG1}={I1$_1$,I1(G1)$_1$,OG1$_0$}
$L_{I1(G1)}$={I1$_1$,I1(G1)$_1$}
G1
G3
L_{I1}={I1$_1$}
G2
$L_{I1(G2)}$={I1$_1$,I1(G2)$_1$}
L_{OG2}={I1$_1$,I1(G1)$_1$,OG2$_0$}
L_{OG3}={I1$_1$,I1(G1)$_1$,I1(G2)$_1$,OG1$_0$,OG2$_0$,OG3$_0$}

Step 4: L_{OG3}={ $I1_{1,}$ $I1(G1)_{1,}$ $I1(G2)_{1,}$ $OG1_0$,$OG2_0$,$OG3_0$}.

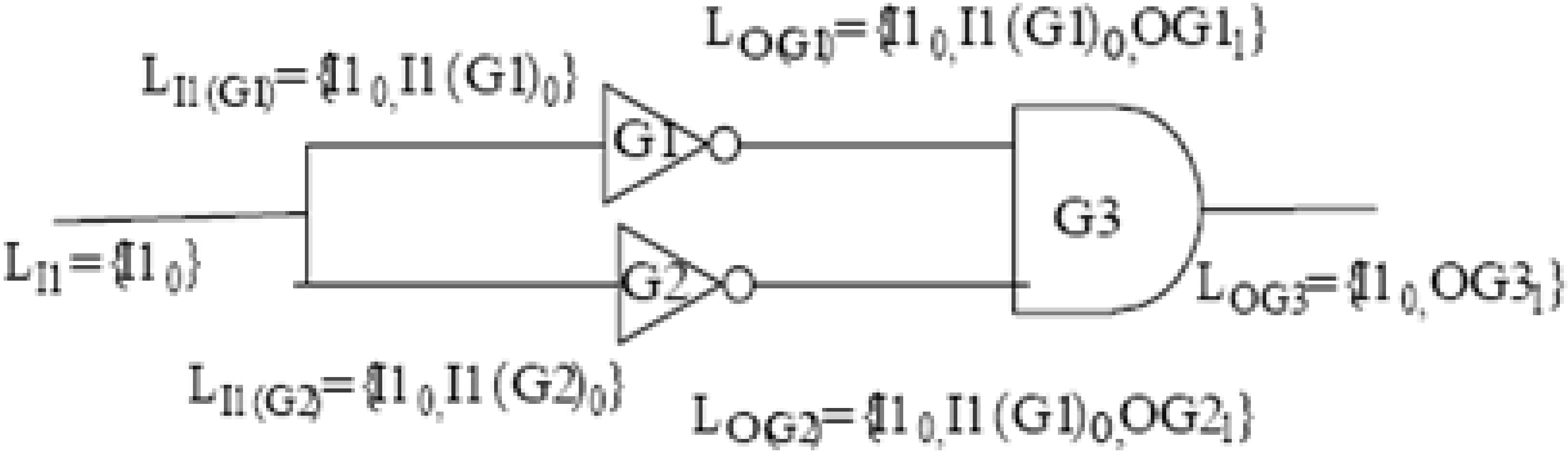
fault deduction at AND gate with both inputs as 1

Fault list detected at output of the AND gate comprise (i) all faults at both the inputs and (ii) s-a-0 at the output of the AND gate.

Deductive Fault Simulation: Example

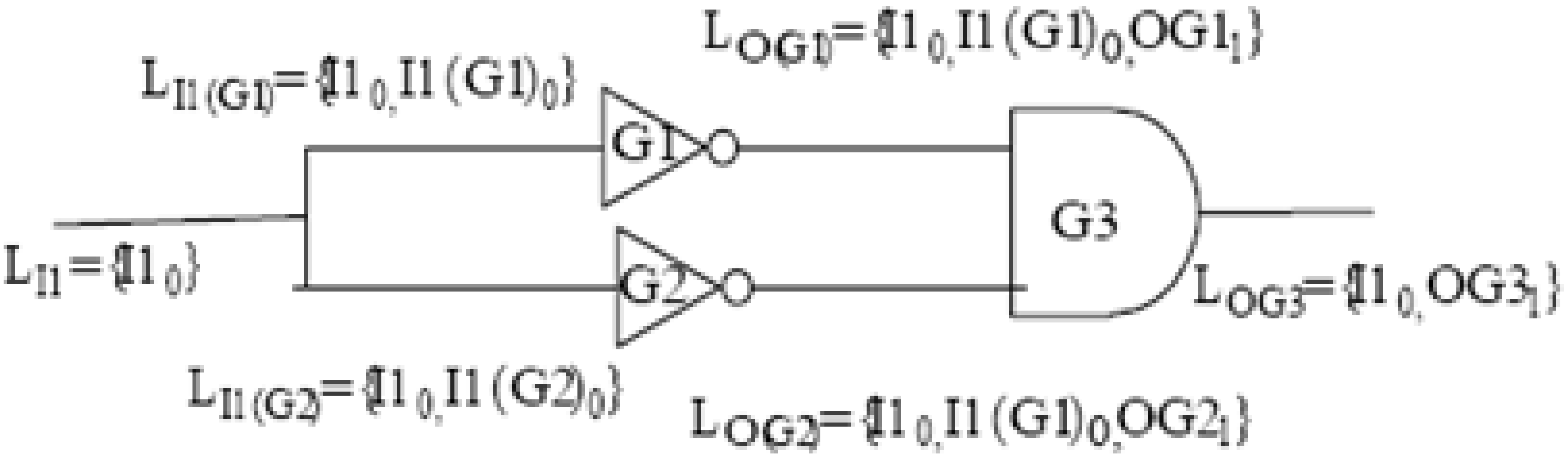


Deductive Fault Simulation: Example


$L_{OG1}=\{I1_0, I1(G1)_0, OG1_1\}$
$L_{I1(G1)}=\{I1_0, I1(G1)_0\}$
G1
G3
$L_{I1}=\{I1_0\}$
G2
$L_{OG3}=\{I1_0, OG3_1\}$
$L_{I1(G2)}=\{I1_0, I1(G2)_0\}$
$L_{OG2}=\{I1_0, I1(G1)_0, OG2_1\}$

Step-1: L_{I1}={ I1$_0$}. s-a-0 at I1 can be detected at I1 as input pattern is I1=1.

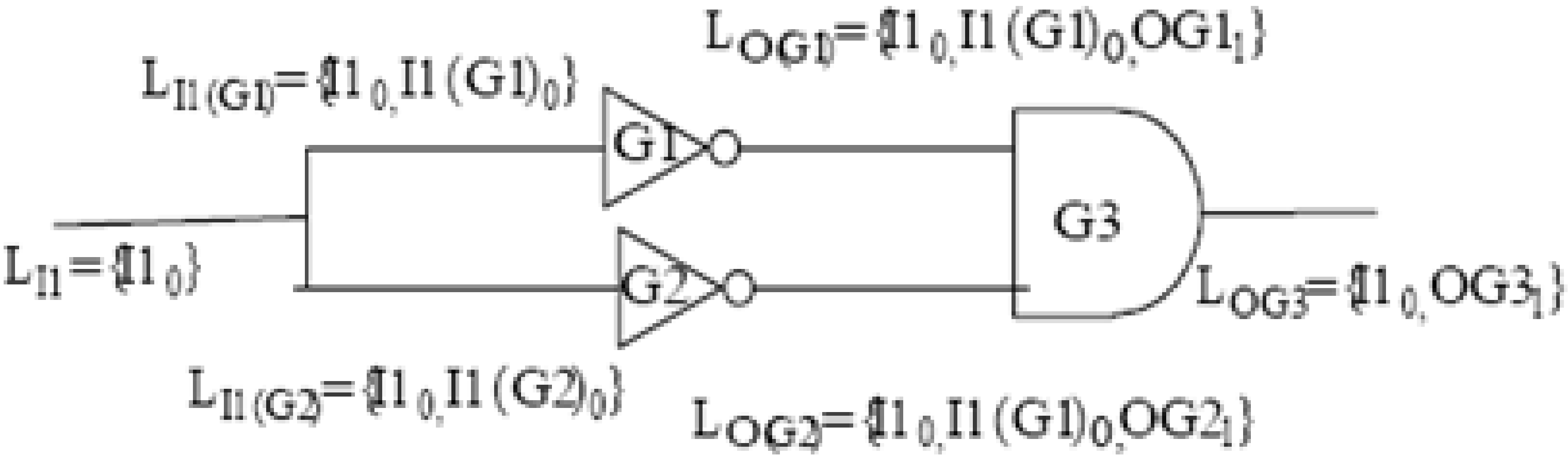
Deductive Fault Simulation: Example


$L_{OG1}=\{I1_0, I1(G1)_0, OG1_1\}$
$L_{I1(G1)}=\{I1_0, I1(G1)_0\}$
G1
G3
$L_{I1}=\{I1_0\}$
G2
$L_{OG3}=\{I1_0, OG3_1\}$
$L_{I1(G2)}=\{I1_0, I1(G2)_0\}$
$L_{OG2}=\{I1_0, I1(G1)_0, OG2_1\}$

Step-2: $L_{I1(G1)}$={ $I1_{0,}$ $I1(G1)_0$}.

$L_{I1(G2)}$={ $I1_{0,}$ $I1(G2)_0$}.

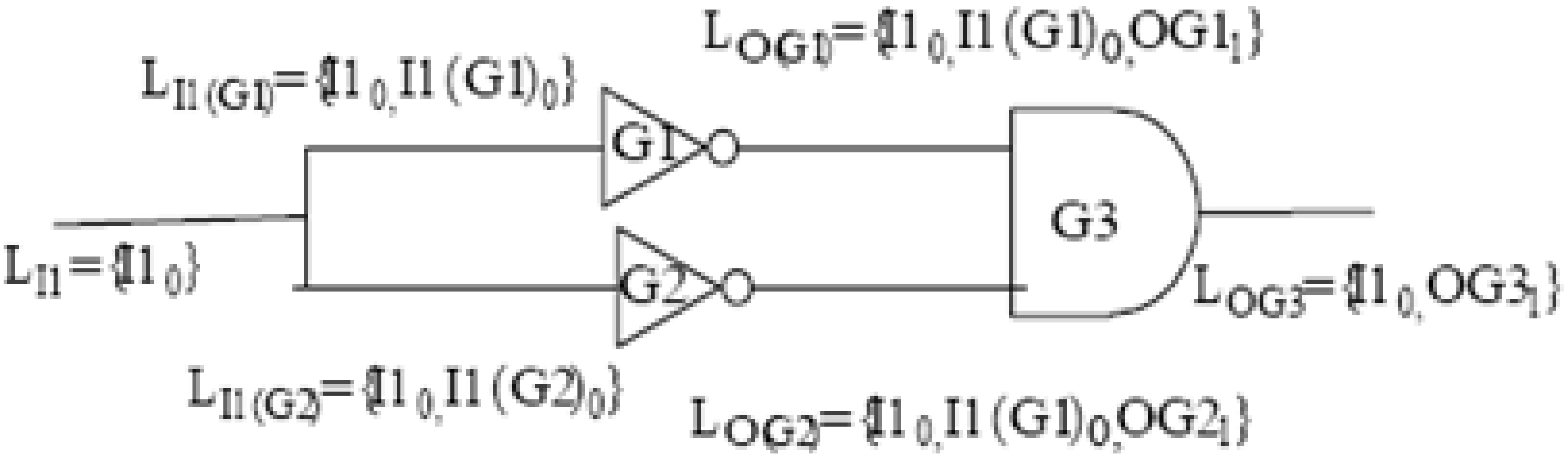
Deductive Fault Simulation: Example



Step 3A: $L_{OG1}=\{ I1_{0,}\ I1(G1)_{0,}OG1_{1}\}$

Step 3B: $L_{OG2}=\{ I1_{0,}\ I1(G2)_{0,}OG2_{1}\}$

Deductive Fault Simulation: Example



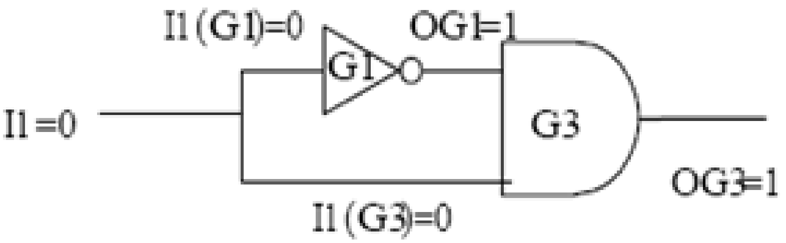
$L_{OG1}=\{I1_0, I1(G1)_0, OG1_1\}$
$L_{I1(G1)}=\{I1_0, I1(G1)_0\}$
G1
G3
$L_{I1}=\{I1_0\}$
G2
$L_{OG3}=\{I1_0, OG3_1\}$
$L_{I1(G2)}=\{I1_0, I1(G2)_0\}$
$L_{OG2}=\{I1_0, I1(G1)_0, OG2_1\}$

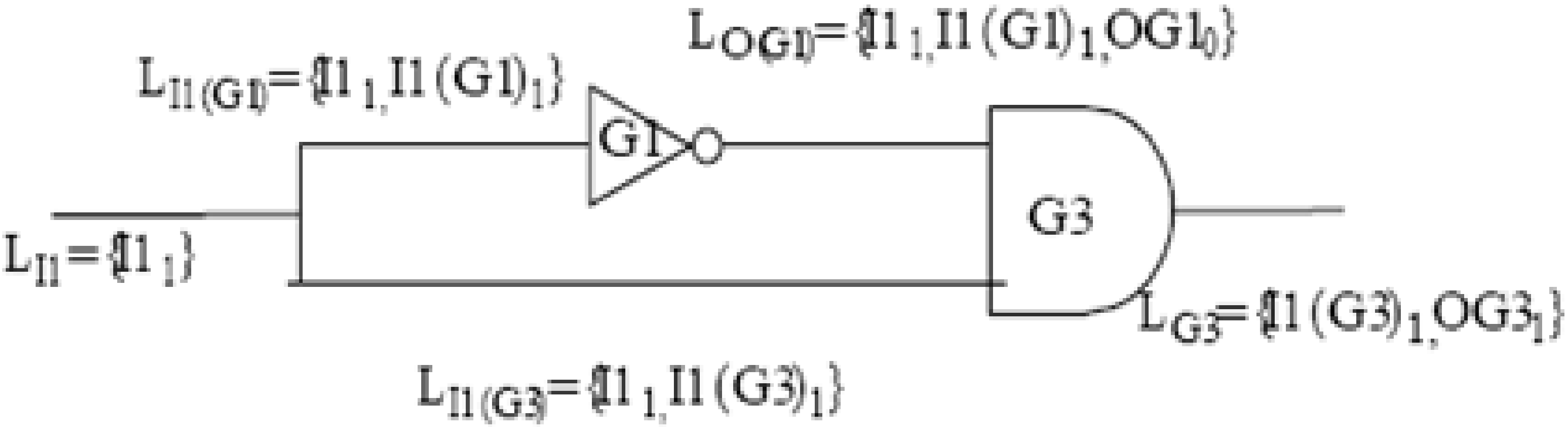
Step-4: $L_{OG3}=\{ I1_{0,} OG3_1\}$

fault deduction at a 2-input AND gate with both inputs as 0

Fault list detected at output of the AND gate comprise (i) all faults COMMON at both the inputs and (ii) s-a-1 at the output of the AND gate.

Deductive Fault Simulation: Example

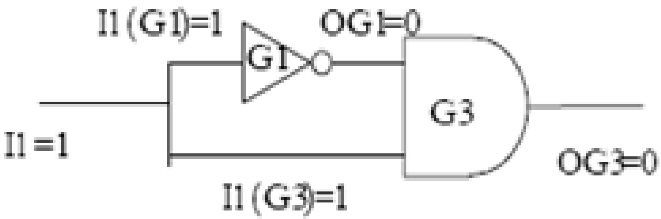


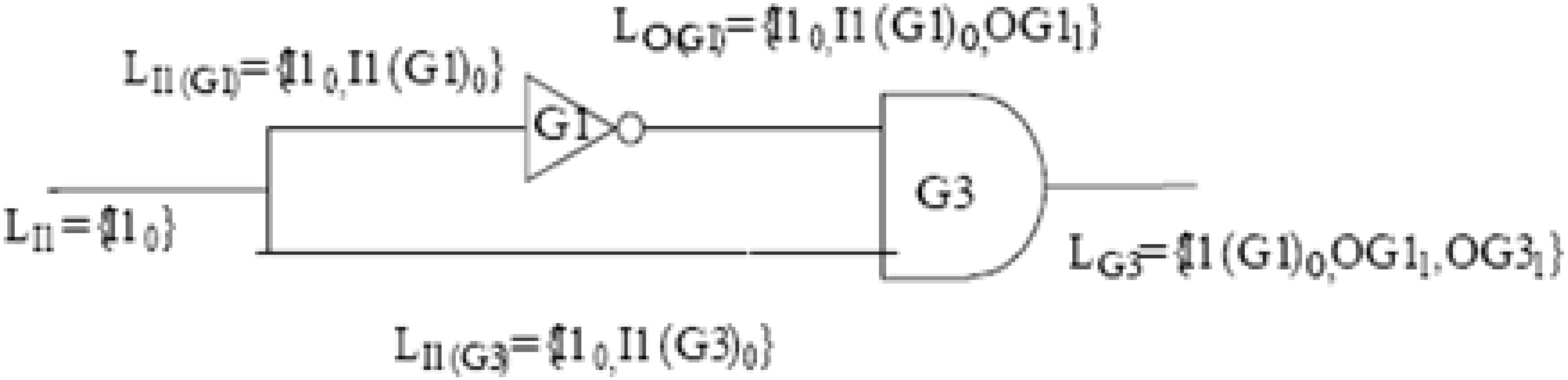

$L_{OG1}=\{I1_1, I1(G1)_1, OG1_0\}$
$L_{I1(G1)}=\{I1_1, I1(G1)_1\}$
G1
G3
$L_{I1}=\{I1_1\}$
$L_{G3}=\{I1(G3)_1, OG3_1\}$
$L_{I1(G3)}=\{I1_1, I1(G3)_1\}$

fault deduction at a 2-input AND gate with In1=1 and In2=0

Fault list detected at output of the AND gate comprise (i) all faults at In2 but not at In1 and (ii) s-a-1 at the output of the AND gate.

Deductive Fault Simulation: Example




L_OG1={I1_0, I1 (G1)_0, OG1_1}
L_I1 (G1)={I1_0, I1 (G1)_0}
G1
G3
L_I1={I1_0}
L_G3={I1 (G1)_0, OG1_1, OG3_1}
L_I1 (G3)={I1_0, I1 (G3)_0}

fault deduction at a 2-input AND gate with In1=0 and In2=1

Fault list detected at output of the AND gate comprise (i) all faults at In1 but not at In2 and (ii) s-a-1 at the output of the AND gate.

Fault deduction rules for logic gates

Gate Type	Inputs (In1 and In2)		Output (O)	Deductive Fault list at O
AND	0	0	0	$[L_{In1} \cap L_{In2}] \cup O_1$
	0	1	0	$[L_{In1} \cap \overline{L_{In2}}] \cup O_1$
	1	0	0	$[\overline{L_{In1}} \cap L_{In2}] \cup O_1$
	1	1	1	$[L_{In1} \cup L_{In2}] \cup O_0$
OR *(Dual of AND gate)*	0	0	0	$[L_{In1} \cup L_{In2}] \cup O_1$
	0	1	1	$[\overline{L_{In1}} \cap L_{In2}] \cup O_0$
	1	0	1	$[L_{In1} \cap \overline{L_{In2}}] \cup O_0$
	1	1	1	$[L_{In1} \cap L_{In2}] \cup O_0$
NOT	0	----	1	$[L_{In1}] \cup O_0$
	1	-----	0	$[L_{In1}] \cup O_1$
Fanout	0	----	0	$[L_{In1}] \cup O_1$
	1	------	1	$[L_{In1}] \cup O_0$

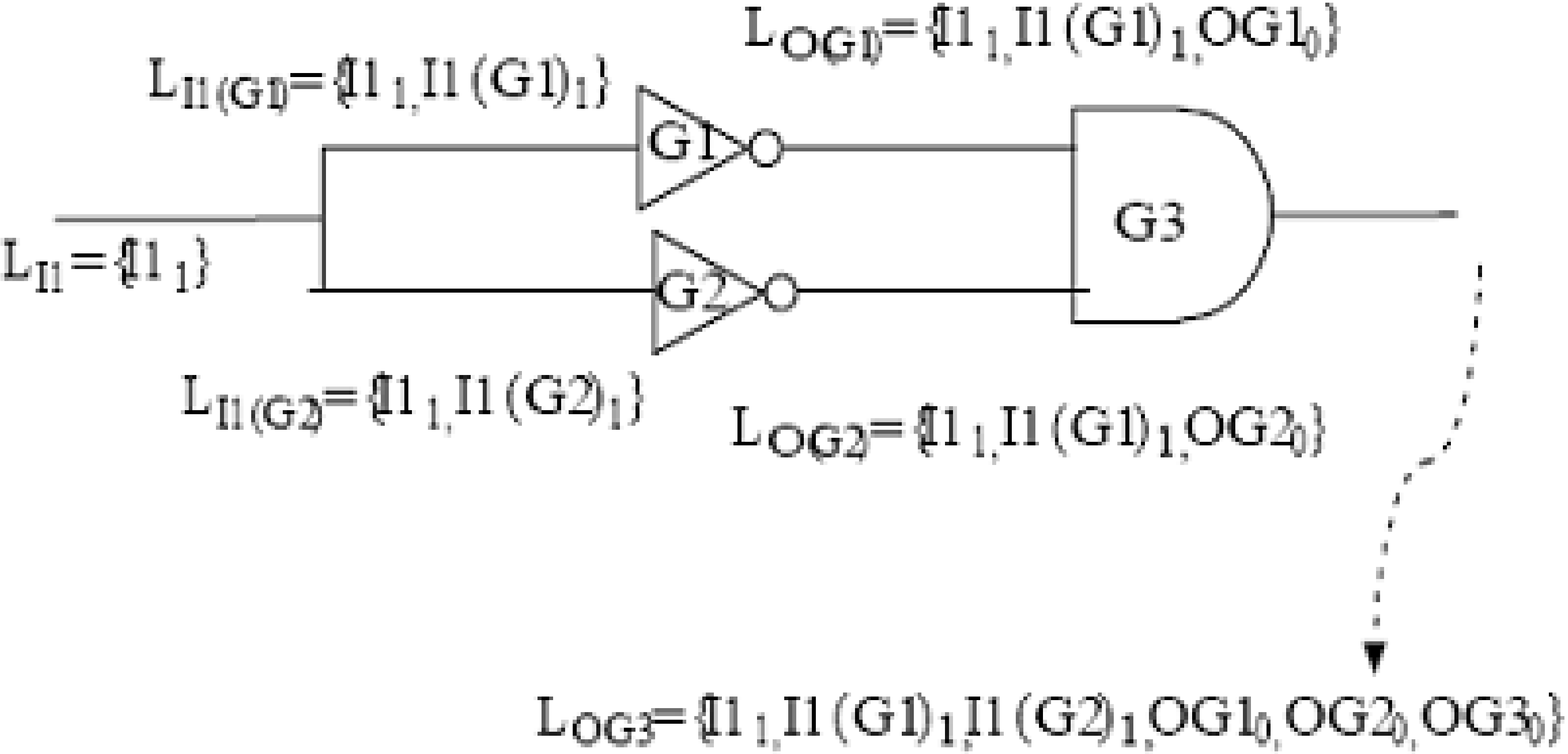
Thank You

Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

Module-VIII

Lecture-III

Fault Simulation

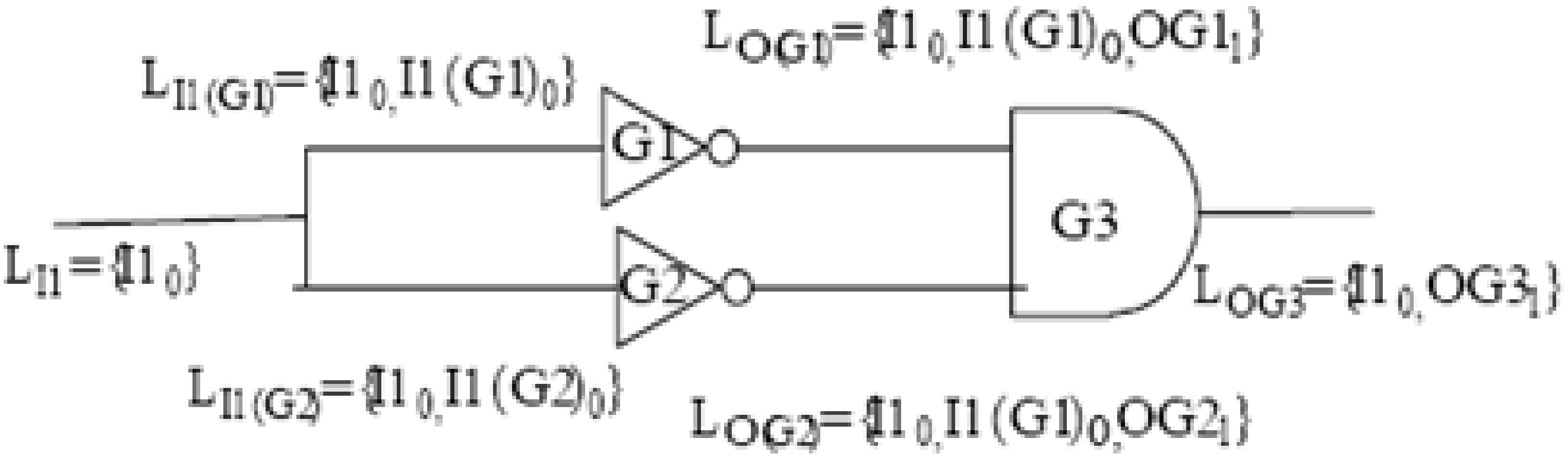

$L_{OG1}=\{I1_{1,}I1(G1)_{1,}OG1_0\}$
$L_{I1(G1)}=\{I1_{1,}I1(G1)_1\}$
G1
G3
$L_{I1}=\{I1_1\}$
G2
$L_{I1(G2)}=\{I1_{1,}I1(G2)_1\}$
$L_{OG2}=\{I1_{1,}I1(G1)_{1,}OG2_0\}$
$L_{OG3}=\{I1_{1,}I1(G1)_{1,}I1(G2)_{1,}OG1_{0,}OG2_{0,}OG3_0\}$

Step 4: L_{OG3}={ $I1_{1,}$ $I1(G1)_{1,}$ $I1(G2)_{1,}$ $OG1_0$,$OG2_0$,$OG3_0$}.

fault deduction at AND gate with both inputs as 1

Fault list detected at output of the AND gate comprise (i) all faults at both the inputs and (ii) s-a-0 at the output of the AND gate.

Deductive Fault Simulation: Example



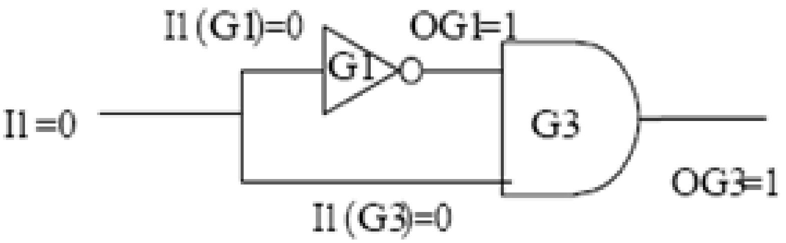
L_I1(G1)={I1_0, I1(G1)_0}
L_OG1={I1_0, I1(G1)_0, OG1_1}
G1
G2
G3
L_I1={I1_0}
L_OG3={I1_0, OG3_1}
L_I1(G2)={I1_0, I1(G2)_0}
L_OG2={I1_0, I1(G1)_0, OG2_1}

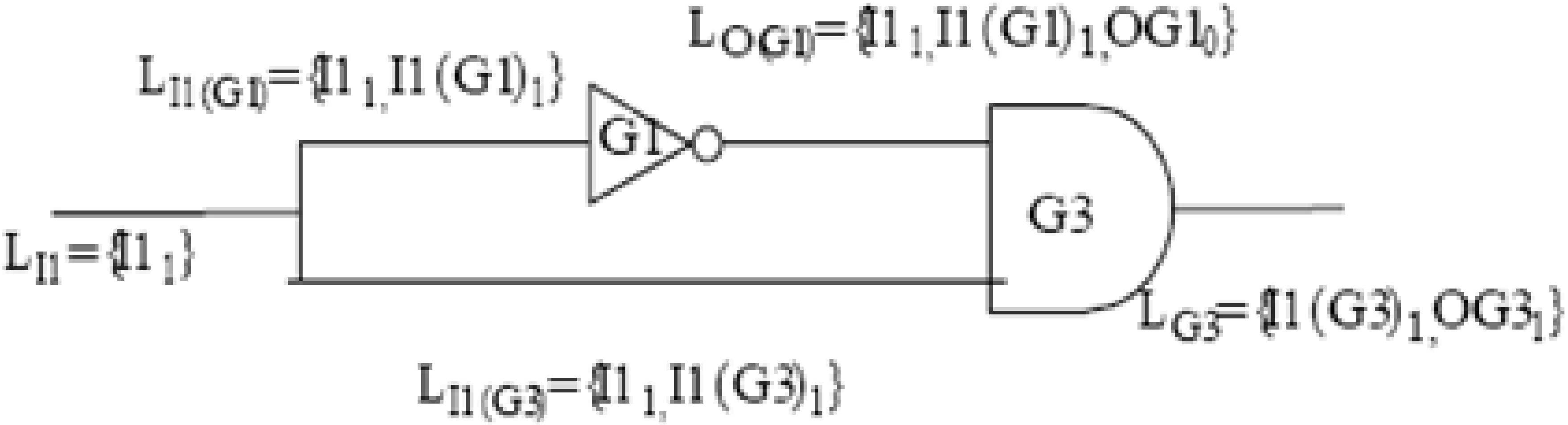
Step-4: L_{OG3}={ $I1_{0,}$ $OG3_1$}

fault deduction at a 2-input AND gate with both inputs as 0

Fault list detected at output of the AND gate comprise (i) all faults COMMON at both the inputs and (ii) s-a-1 at the output of the AND gate.

Deductive Fault Simulation: Example

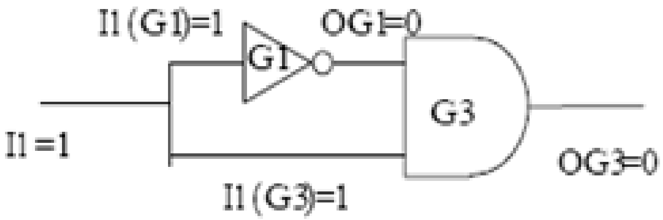


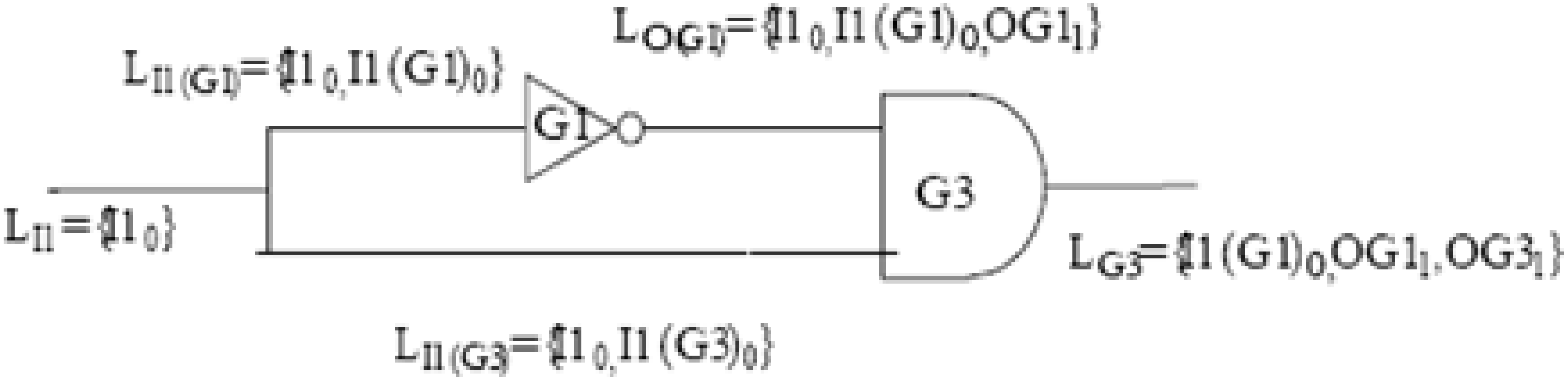

L_OG1={I1_1, I1 (G1)_1, OG1_0}
L_I1(G1)={I1_1, I1 (G1)_1}
G1
G3
L_I1={I1_1}
L_G3={I1 (G3)_1, OG3_1}
L_I1(G3)={I1_1, I1 (G3)_1}

fault deduction at a 2-input AND gate with In1=1 and In2=0

Fault list detected at output of the AND gate comprise (i) all faults at In2 but not at In1 and (ii) s-a-1 at the output of the AND gate.

Deductive Fault Simulation: Example




$L_{OG1}=\{I1_0, I1(G1)_0, OG1_1\}$
$L_{I1(G1)}=\{I1_0, I1(G1)_0\}$
G1
G3
$L_{I1}=\{I1_0\}$
$L_{G3}=\{I1(G1)_0, OG1_1, OG3_1\}$
$L_{I1(G3)}=\{I1_0, I1(G3)_0\}$

fault deduction at a 2-input AND gate with In1=0 and In2=1

Fault list detected at output of the AND gate comprise (i) all faults at In1 but not at In2 and (ii) s-a-1 at the output of the AND gate.

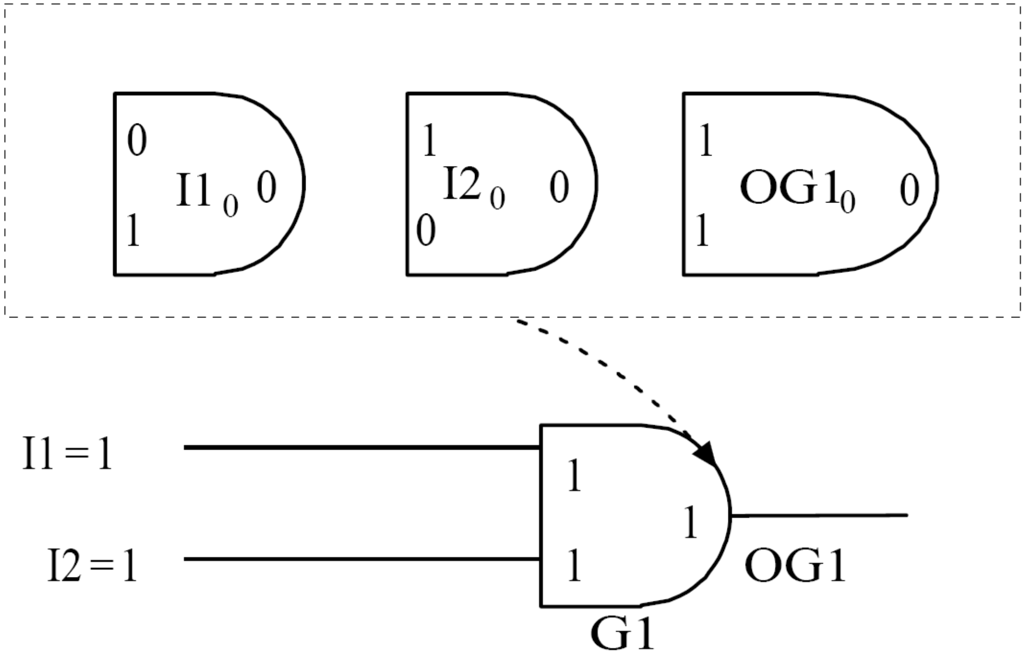
Fault deduction rules for logic gates

Gate Type	Inputs (In1 and In2)		Output (O)	Deductive Fault list at O
AND	0	0	0	$[L_{In1} \cap L_{In2}] \cup O_1$
	0	1	0	$[L_{In1} \cap \overline{L_{In2}}] \cup O_1$
	1	0	0	$[\overline{L_{In1}} \cap L_{In2}] \cup O_1$
	1	1	1	$[L_{In1} \cup L_{In2}] \cup O_0$
OR *(Dual of AND gate)*	0	0	0	$[L_{In1} \cup L_{In2}] \cup O_1$
	0	1	1	$[\overline{L_{In1}} \cap L_{In2}] \cup O_0$
	1	0	1	$[L_{In1} \cap \overline{L_{In2}}] \cup O_0$
	1	1	1	$[L_{In1} \cap L_{In2}] \cup O_0$
NOT	0	----	1	$[L_{In1}] \cup O_0$
	1	-----	0	$[L_{In1}] \cup O_1$
Fanout	0	----	0	$[L_{In1}] \cup O_1$
	1	------	1	$[L_{In1}] \cup O_0$

Concurrent Fault Simulation

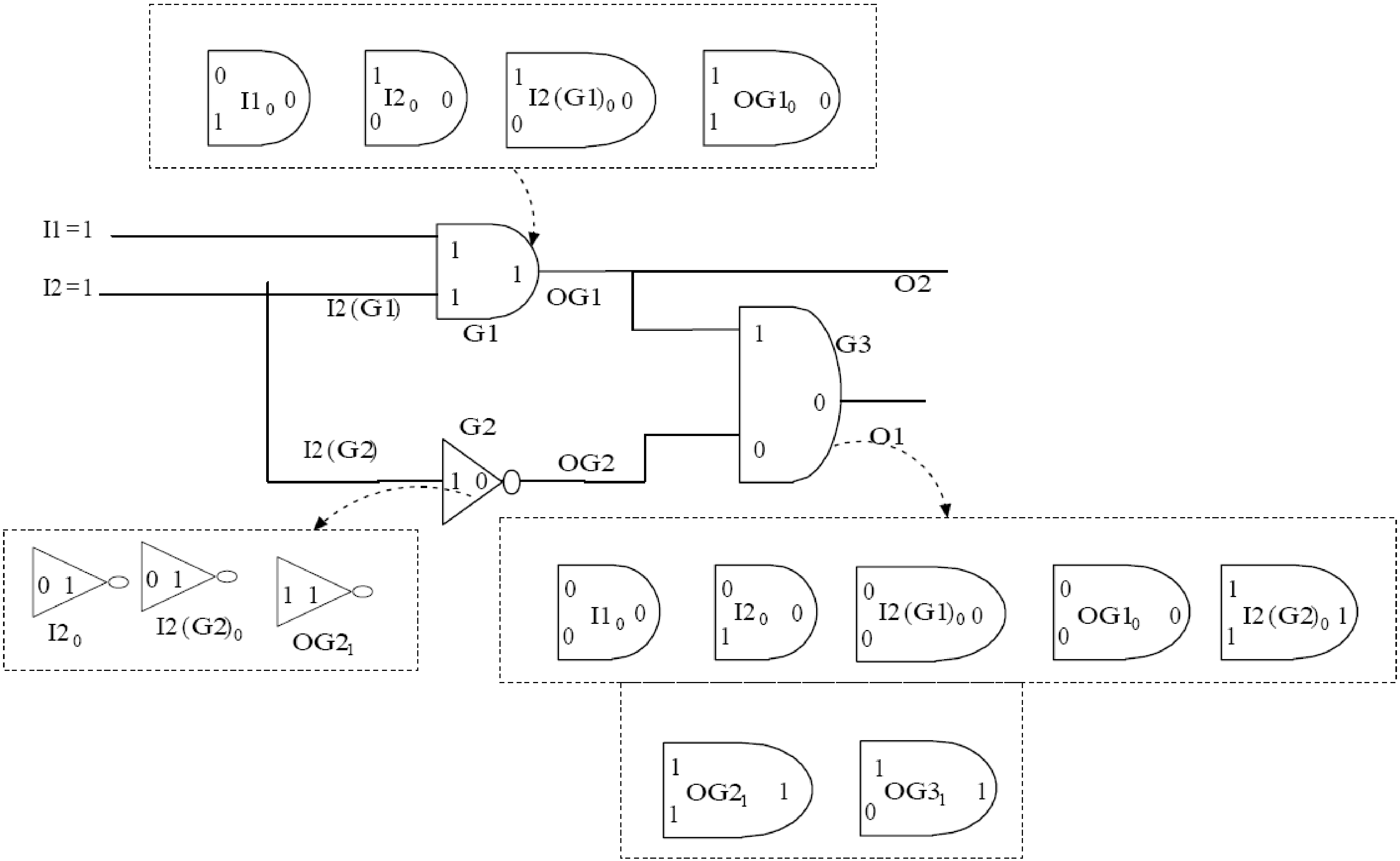
- Detective fault simulation can determine all the faults in one iteration detectable by a random pattern.
  - When a new random pattern is fed as input the whole process needs to be redone.
- “Concurrent Fault Simulation” is a technique similar to deductive fault simulation, however, retains information when moving from one random pattern to another.
  - In concurrent fault simulation when a new random pattern is fed it needs to compute only that information which got changed by the new pattern.
- So, concurrent fault simulation gets motivation from the advantages achieved by event driven simulation compared to compiled code simulation.

Concurrent Fault Simulation: Concept



To each gate is associated a number of gates “affected by some fault in the circuit”. An affected gate is one whose at least one input or output is different from the ones in the original (normal gate).

Concurrent Fault Simulation: Example



Concurrent Fault Simulation

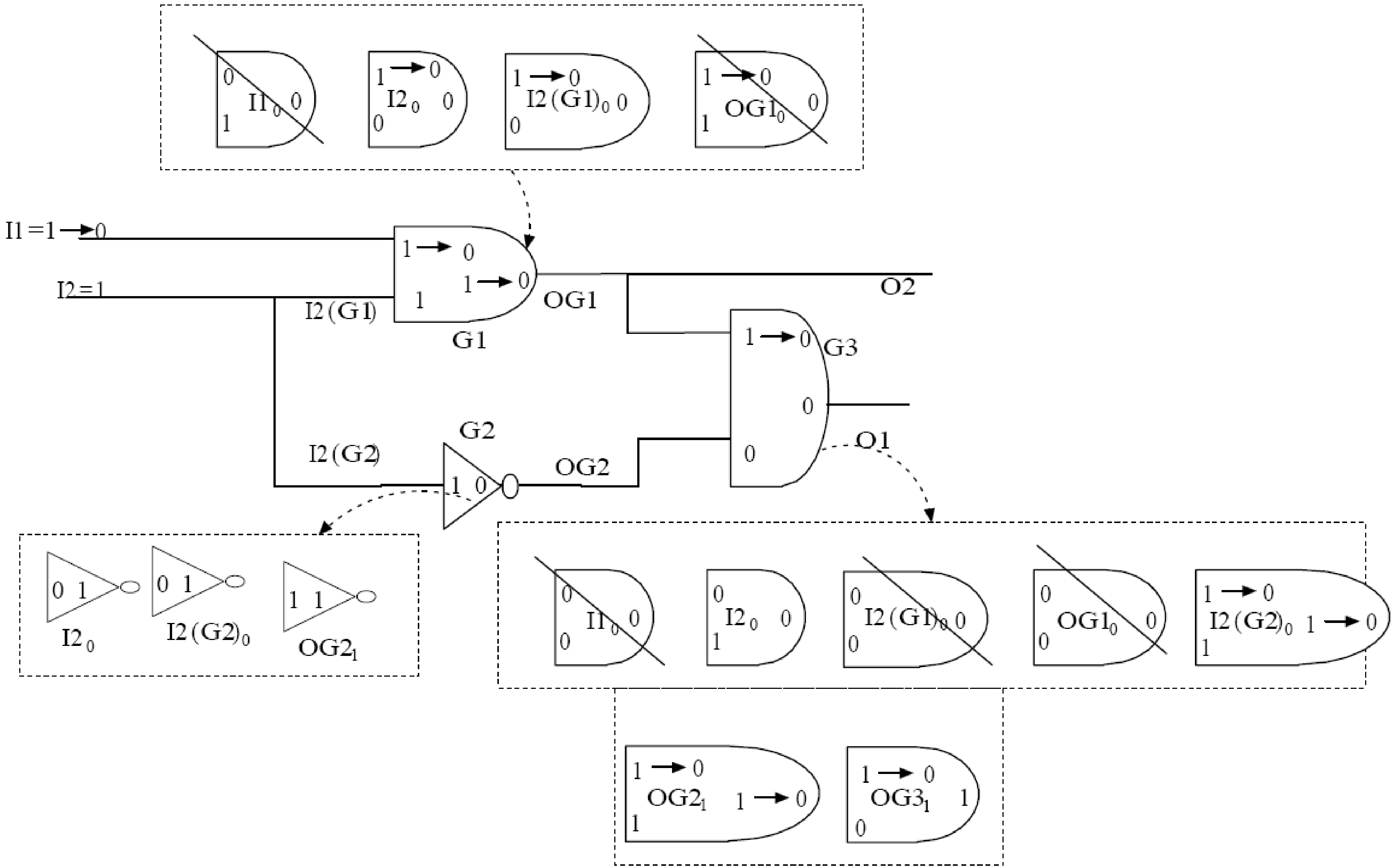
Given a circuit, level wise affected gates corresponding to all normal gates are created. Now, affected gates (in the list of normal gates) that drive some primary output are considered. Among those affected gates, the ones whose output signal value differs from that of the normal gate, correspond to faults being detected by the random pattern given as input.

Concurrent Fault Simulation

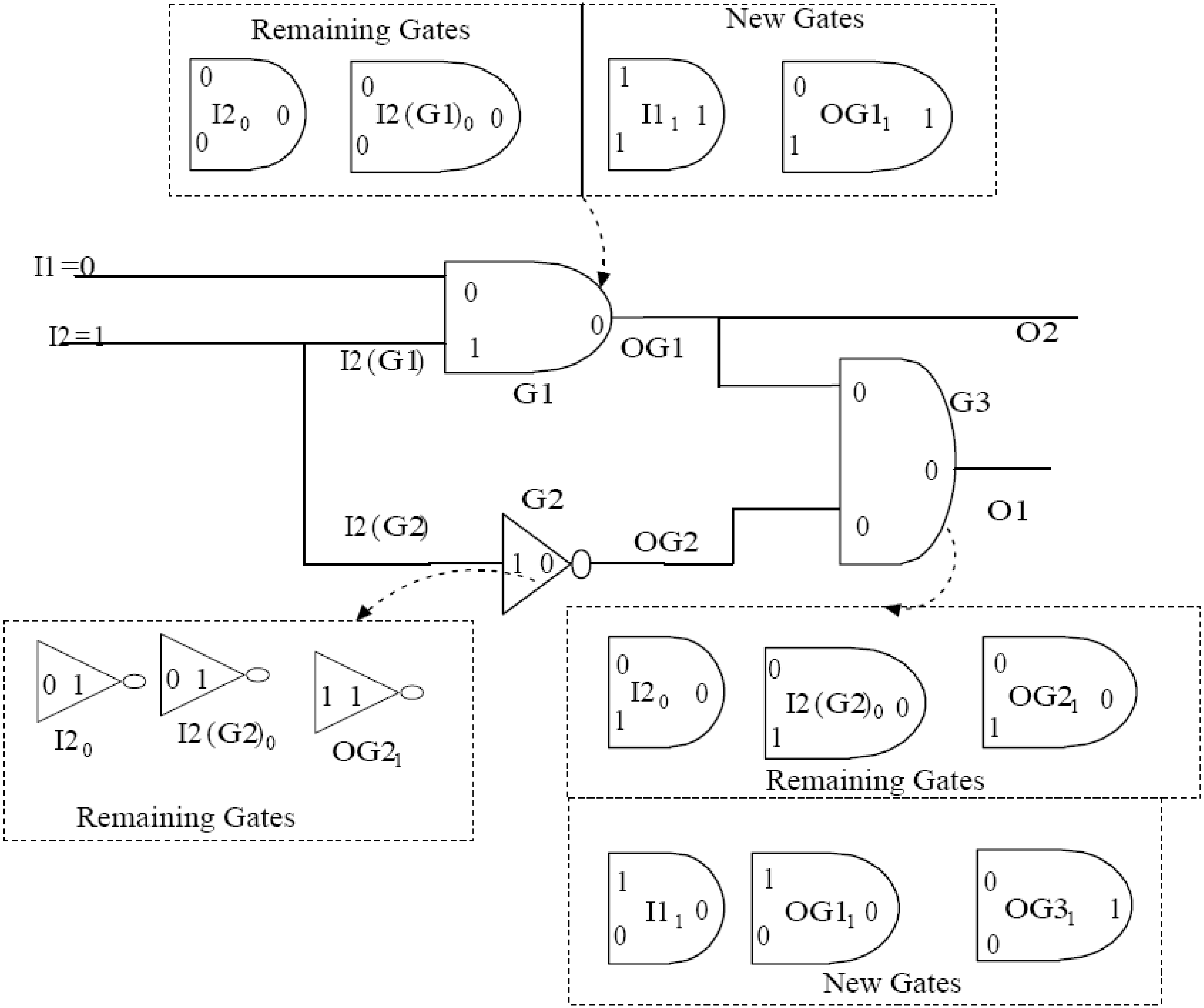
Only three gates correspond to faults being detected at OG3 (or primary output O1). So what is requirement of seven gates in the affected list?
Deductive fault simulation, which keeps information about only these three faults is better choice than concurrent fault simulation?

Next random pattern isI1=0, I2=1. In case of deductive fault simulation we need to repeat all the steps, while in case of concurrent simulation we will re-compute only the information that changes as a result of changes in signals triggered by I1 (from 1 to 0).

Concurrent Fault Simulation: Example



Concurrent Fault Simulation: Example



Conclusions

- To conclude, fault simulation algorithms help to determine patters that can test a subset of faults in a circuit.

- Broadly speaking, after about 90% of faults being detected by random patters and fault simulation, we need to go for ATPG by sensitization–propagation -justification approach.

- Now, if there was a scheme that could tell which 90% of faults are easy to test (by random patterns) and which are difficult to test, then fault simulation algorithms could be more focused. In other words, fault simulation algorithms would stop when most of the easy faults were covered.

Questions and Answers

What is the main advantage of compiled code simulator versus event driven simulator?

Answer: In case of compiled code simulator the whole circuit (i.e., all gates) needs to be simulated when input pattern changes. In case of event driven simulation, only those gates are required to be simulated whose inputs change because of change in the input. So, event driven simulation is more time efficient compared to compiled code simulation.

Questions and Answers

Which fault simulation algorithm is most dependent on architecture of the computer simulating it?

Answer: Concurrent fault simulator is mostly dependent on the architecture of the computer simulating it because speed (i.e., number of faults simulated per iteration) depends on the bit width of the word of the computer. If the bit width is *w,* then *w-1* faults can be simulated in an iteration.

Questions and Answers

When the procedure for test pattern generation by fault simulation is stopped and ATPG by sensitization–propagation –justification approach is taken?

Answer: Under two cases, test pattern generation by fault simulation is stopped.

A high percentage of faults are detected by fault simulation; it implies that most of the easy to test faults are covered and the ones remaining are difficult to test faults.

For a significant number of random patterns at a stretch, new faults covered are nominal; it implies that in the circuit a substantial percentage of faults are difficult to test and one should resort to sensitization–propagation –justification approach.

Thank You

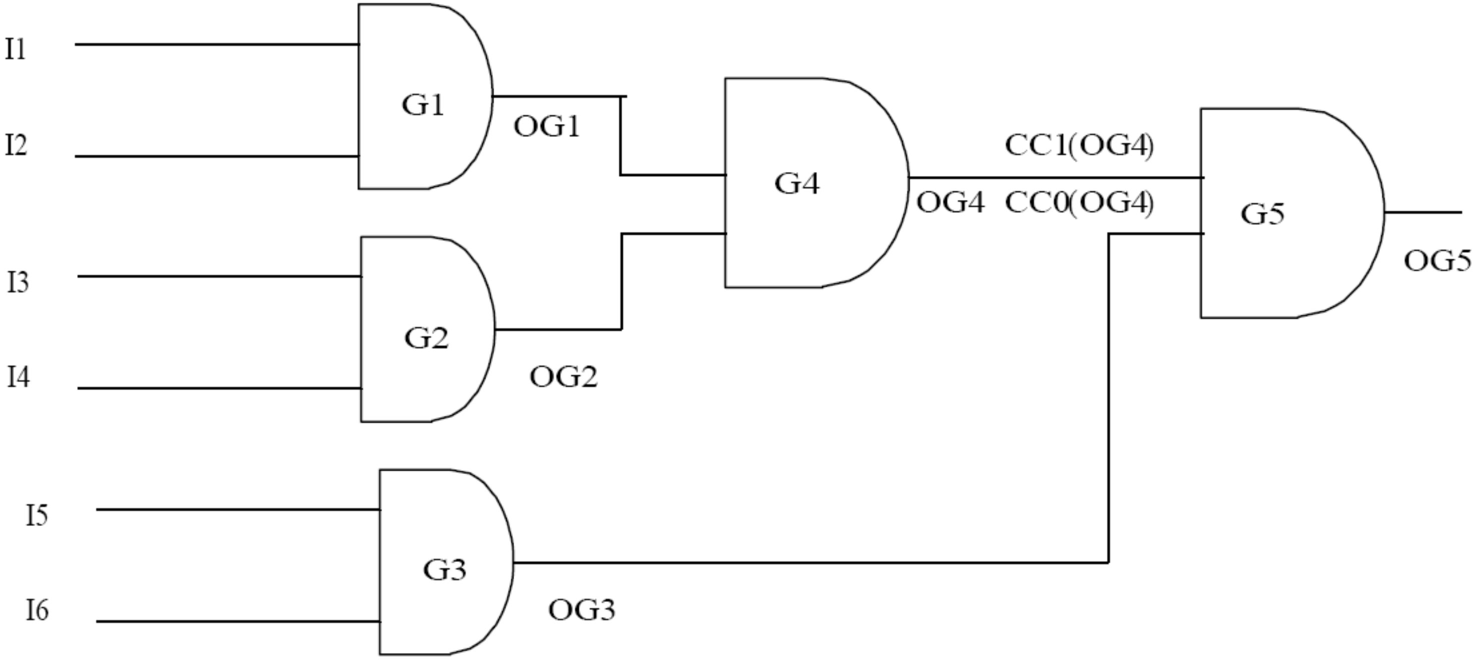
Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

Module-VIII

Lecture-IV

Testability Measures (SCOAP)

Testability Measures (SCOAP): Introduction



CC1(OG4)----I1=1,I2=1,I3=1,I4=1

CC0(OG4)----Any combination except
I1=1,I2=1,I3=1,I4=1

A quick heuristic based algorithm that can rank the faults by their difficulty in testing.

CC0 (OG4)=15/16, CC1(OG4)=1/16

So, s-a-0 fault at OG4 is more difficult to test than s-a-1 fault at OG4.

Testability Measures (SCOAP): Introduction

This procedure determined which fault is more difficult to test, but at the same time also found a pattern to test it.

s-a-0 fault at OG4, test pattern is I1=1,I2=1,I3=1,I4=1,I5=1,I6=1;
I1=1,I2=1,I3=1,I4=1 makes OG4 to 1 --*Sensitization*
OG3 is 1—*Propagation*
I5=1,IG=1—*Justification*
The scheme rank faults on basis of their difficulty in testing is as complex as ATPG by Sensitization- Propagation-Justification.

Approximate but computationally simple Algorithm and can order all the faults (by their difficulty in testing) in two iterations of the circuit.
This algorithm is called SCOAP--*Sandia Controllability/Observability Analysis Program.*

SCOAP Procedure: Introduction

In SCOAP, each net *l* say, has three values associated,

CC0(*l*): controllability of 0 at *l*

CC1(*l*): controllability of 1 at *l*

CO(*l*): observability of *l* at a primary output

Each net *l* has associated with it positive integers n_1,n_2,n_3 and represented as is (n_1,n_2)n_3, where CC0(l)= n_1, CC1(l) =n_2,CO(l)=n_3.

SCOAP Procedure

1. First, all the primary inputs are directly assigned CC0=1 and CC1 =1; it is assumed that as the primary inputs are directly controllable, to make their signal values 0 or 1 require “effort proportional to 1”.
2. Following that, CC0 and CC1 are determined level wise for the circuit using SCOAP rules for logic gates
3. In a similar way, combinational observability (CO) of all primary outputs is assumed 0; as primary outputs are directly observable, to see their signal values require an “effort proportional to 0”.
4. Following that, CO values are determined level wise for the circuit (now moving from primary outputs to primary inputs) using SCOAP rules.

SCOAP Analysis

Now, given a net *l* , difficulty to test a s-a-0 fault in it is proportional to CC1(*l*) (as we need to apply 1 in *l*) plus CO(*l*) (as we need to observe the value of *l* at a primary output).

In a similar way, difficulty to test a s-a-1 fault at *l* is proportional to CC0(*l*) plus CO(*l*).

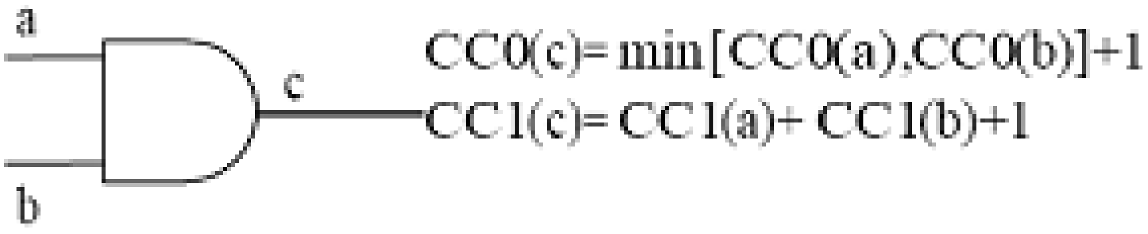
SCOAP Rules to Compute CC0 and CC1

AND gate with two inputs as **a**, **b** and output as **c**.

To compute CCO(c)/CC1(c) we need to know CC0(a),CC1(a) and CC0(b),CC1(b).

To control the value of **c** to 0 (CCO(c)), either **a** is to be 0 or **b** is to be 0. So, CCO(c) is minimum of difficulty to control **a** to 0 or **b** to 0. Also we add 1 to CCO(c) as we progress by a level when we go from input to output of a gate. So, CCO(c) = min[CCO(a),CC0(b)] +1.

To control the value of **c** to 1 (CC1(c)), both **a** and **b** are to be 1. So, CC1(c) is sum of difficulty to control **a** and **b** to 1. Also we add 1 to CC1(c) as we progress by a level when we go from input to output of a gate. So, CC1(c) = CC1(a)+CC1(b) +1.

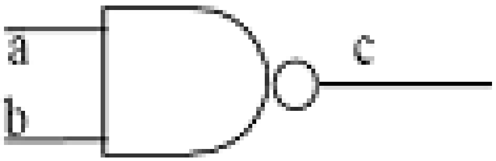


SCOAP Rules to Compute CC0 and CC1

CCO(c)/CC1(c) for NAND gate (two inputs as **a**, **b** and output as **c**) can be computed as follows

To control the value of **c** to 0 (CC0(c)), both **a** and **b** are to be 1. So, CCO(c) is sum of difficulty to control **a** and **b** to 1, plus 1 for change in level. So, CC0(c) = CC1(a)+CC1(b) +1.

To control the value of **c** to 1 (CC1(c)), either **a** is to be 0 or **b** is to be 0. So, CC1(c) = min[CCO(a),CC0(b)] +1.



$CC0(c)=CC1(a)+CC1(b)+1$

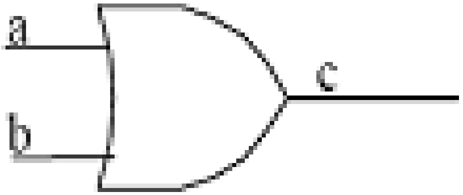
$CC1(c)=\min[CC0(a),CC0(b)]+1$

SCOAP Rules to Compute CC0 and CC1

CCO(c)/CC1(c) for OR gate (two inputs as **a**, **b** and output as **c**) can be computed as follows

To control the value of **c** to 0 (CC0(c)), both **a** and **b** are to be 0. So, CC0(c) = CC0(a)+CC0(b) +1.

To control the value of **c** to 1 (CC1(c)), either **a** is to be 1 or **b** is to be 1. So, CC1(c) = min[CC1(a),CC1(b)] +1.



CC0(c)= CC0(a)+ CC0(b)+1

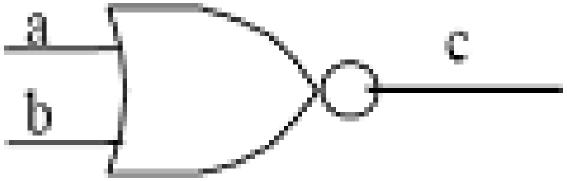
CC1(c)= min[CC1(a),CC1(b)]+1

SCOAP Rules to Compute CC0 and CC1

CCO(c)/CC1(c) for NOR gate (two inputs as **a**, **b** and output as **c**) can be computed as follows

To control the value of **c** to 0 (CCO(c)), either **a** is to be 1 or **b** is to be 1. So, CC0(c) = min[CC1(a),CC1(b)] +1.

To control the value of **c** to 1 (CC1(c)), both **a** and **b** are to be 0. So, CC1(c) = CC0(a)+CC0(b) +1.



$CC0(c)= \min[CC1(a),CC1(b)]+1$

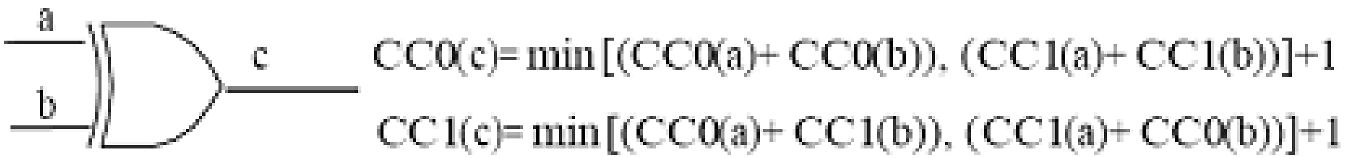
$CC1(c)= CC0(a)+ CC0(b)+1$

SCOAP Rules to Compute CC0 and CC1

CCO(c)/CC1(c) for XOR gate (two inputs as **a**, **b** and output as **c**) can be computed as follows

To control the value of **c** to 0 (CC0(c)), either both **a** and **b** are to be 0 or both **a** and **b** are to be 1. So, CC0(c) = min[(CC0(a)+CC0(b)), (CC1(a)+CC1(b))] +1.

To control the value of **c** to 1 (CC1(c)), either **a** is 1 and **b** is 0 or **a** is 0 and **b** is 1. So, CC1(c) = min[(CC0(a)+CC1(b)), (CC1(a)+CC0(b))] +1.



CC0(c)= min [(CC0(a)+ CC0(b)), (CC1(a)+ CC1(b))]+1

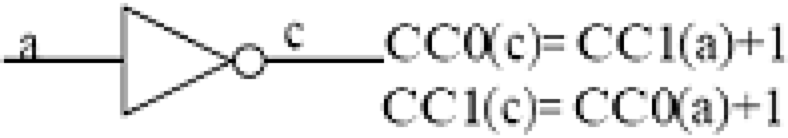
CC1(c)= min [(CC0(a)+ CC1(b)), (CC1(a)+ CC0(b))]+1

SCOAP Rules to Compute CC0 and CC1

CCO(c)/CC1(c) for NOT gate (input as **a** and output as **c**) can be computed as follows

To control the value of **c** to 0 (CC0(c)), **a** is to be 1. So, CC0(c) = CC1(a)+1.

To control the value of **c** to 1 (CC1(c)), **a** is to be 0. So, CC1(c) = CC0(a)+1.

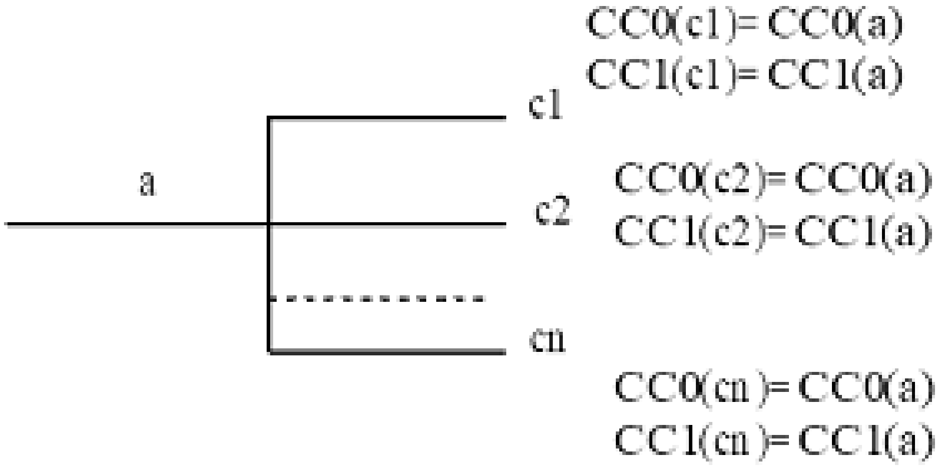


SCOAP Rules to Compute CC0 and CC1

CC0(c1), CC0(c2)…. CC0(cn)/ CC1(c1), CC1(c2)…. ,CC1(cn) for a fanout net with stem as **a** and **c1,c2,…,cn** as branches can be computed as follows

As all the values of the branches can be controlled by the value at the stem CC0(c1)=CC0(c2)…. =CC0(cn)=**CC0(a)**.

Similarly, CC1(c1)=CC1(c2)…. =CC1(cn)=**CC1(a).** As there is no change in level, 1 is not added.

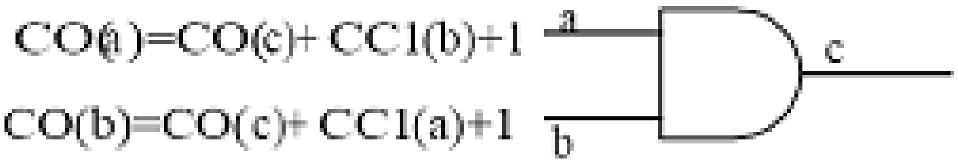


SCOAP Rules to CO

AND gate with two inputs as **a**, **b** and output as **c**. To compute CO(a) and CO(b) we need to know CO(c).

To observe the value of **a** at a primary output (CO(a)), **b** is to be 1. So, CO(a) is the difficulty to control **b** to 1 plus the observability of **c**. Also we add 1 to CO(a) as we progress by a level when we go from output to input of a gate. So, CO(a) = CC1(b) + CO(c)+1.

To observe the value of **b** at a primary output (CO(b)), **a** is to be 1. Similar to computation of CO(a), CO(b) = CC1(a) + CO(c)+1.



SCOAP Rules to CO

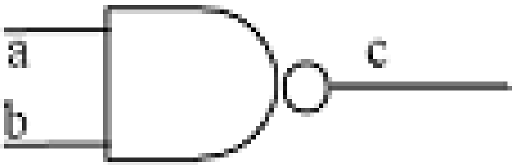
CO(a), CO(b) for NAND gate (two inputs as **a**, **b** and output as **c**) can be computed as follows

To observe the value of **a** at a primary output (CO(a)), **b** is to be 1. So, CO(a) = CC1(b) + CO(c)+1.

To observe the value of **b** at a primary output (CO(b)), **a** is to be 1. Similar to computation of CO(a), CO(b) = CC1(a) + CO(c)+1.

$$CO(a)=CO(c)+CC1(b)+1$$

$$CO(b)=CO(c)+CC1(a)+1$$

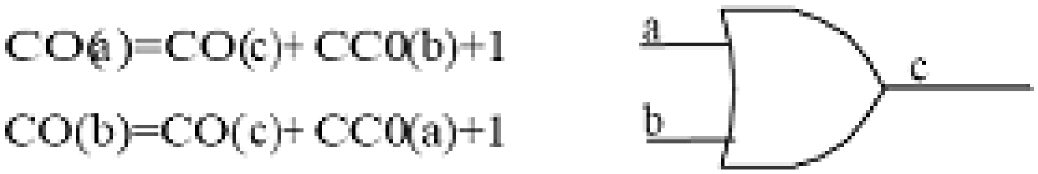


SCOAP Rules to CO

CO(a), CO(b) for OR gate (two inputs as **a**, **b** and output as **c**) can be computed as follows

To observe the value of **a** at a primary output (CO(a)), **b** is to be 0. So, CO(a) = CC0(b) + CO(c)+1.

To observe the value of **b** at a primary output (CO(b)), **a** is to be 0. Similar to computation of CO(a), CO(b) = CC0(a) + CO(c)+1.



CO(a)=CO(c)+CC0(b)+1

CO(b)=CO(c)+CC0(a)+1

SCOAP Rules to CO

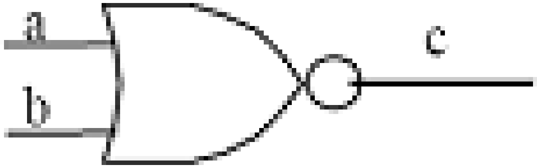
CO(a), CO(b) for NOR gate (two inputs as **a**, **b** and output as **c**) is same as that of OR gate and can be computed as follows

CO(a) = CC0(b) + CO(c)+1.

CO(b) = CC0(a) + CO(c)+1.

CO(a)=CO(c)+ CC0(b)+1

CO(b)=CO(c)+ CC0(a)+1



SCOAP Rules to CO

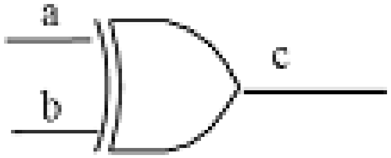
CO(a) and CO(b) for XOR gate (two inputs as **a**, **b** and output as **c**) can be computed as follows

To observe the value of **a** at a primary output (CO(a)), **b** is to be 0 or 1; if **b** is 1 then **c**=**a** and if **b** is 0 then **c**=NOT(**a)**. So, CO(a) is the minimum of (difficulty to control **b** to 1 or difficulty to control **b** to 0) plus the observability of **c**. Also we add 1 to CO(a) as we progress by a level when we go from output to input of a gate. So, CO(a) = min(CC0(b),CC1(b)) + CO(c)+1.

Similar to computation of CO(a), CO(b) = min(CC0(a),CC1(a)) + CO(c)+1.

CO(a)=CO(c)+ min[CC0(b),CC1(b)]+1

CO(b)=CO(c)+ min[CC0(a),CC1(b)]+1

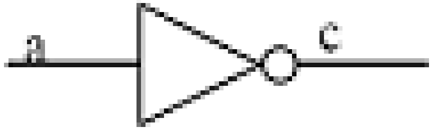


SCOAP Rules to CO

CO(a) for NOT gate (input as **a** and output as **c**) can be computed as follows

There is only one input in a NOT gate and it is always observable at the output (i.e., **c**=NOT(**a**)); so C0(a) = CO(c)+1.

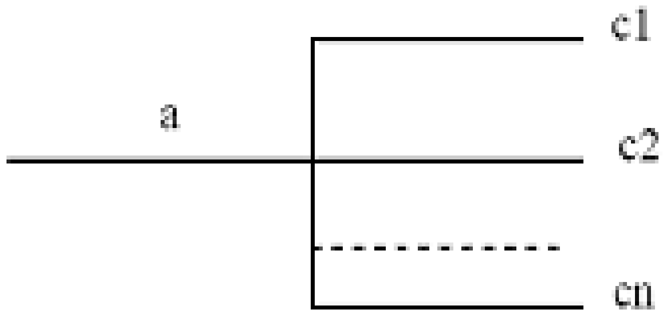
$$CO(a)=CO(c)+1$$
$$CO(a)=CO(c)+1$$



SCOAP Rules to CO

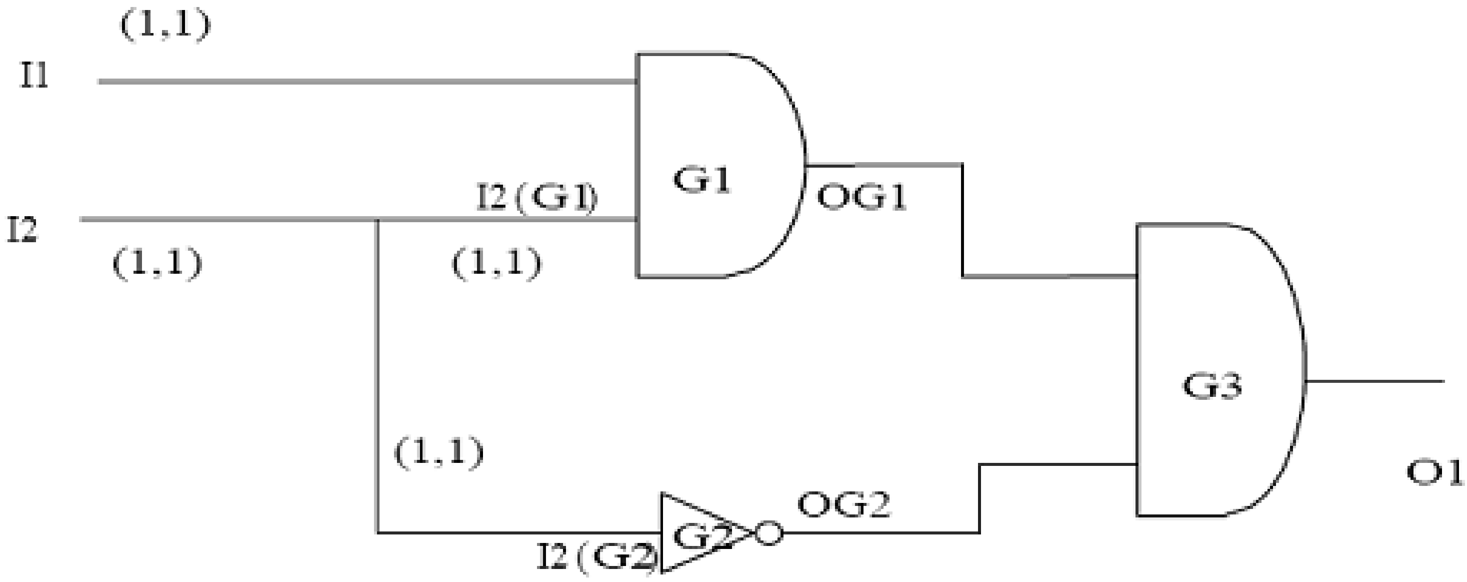
CO(a) for a fanout net with stem as **a** and **c1,c2,…,cn** as branches can be computed as follows

As all the values of the branches can be observed at the stem, C0(a) = min[CO(c1), CO(c2)…,CO(cn)]. As there is no change in level, 1 is not added.



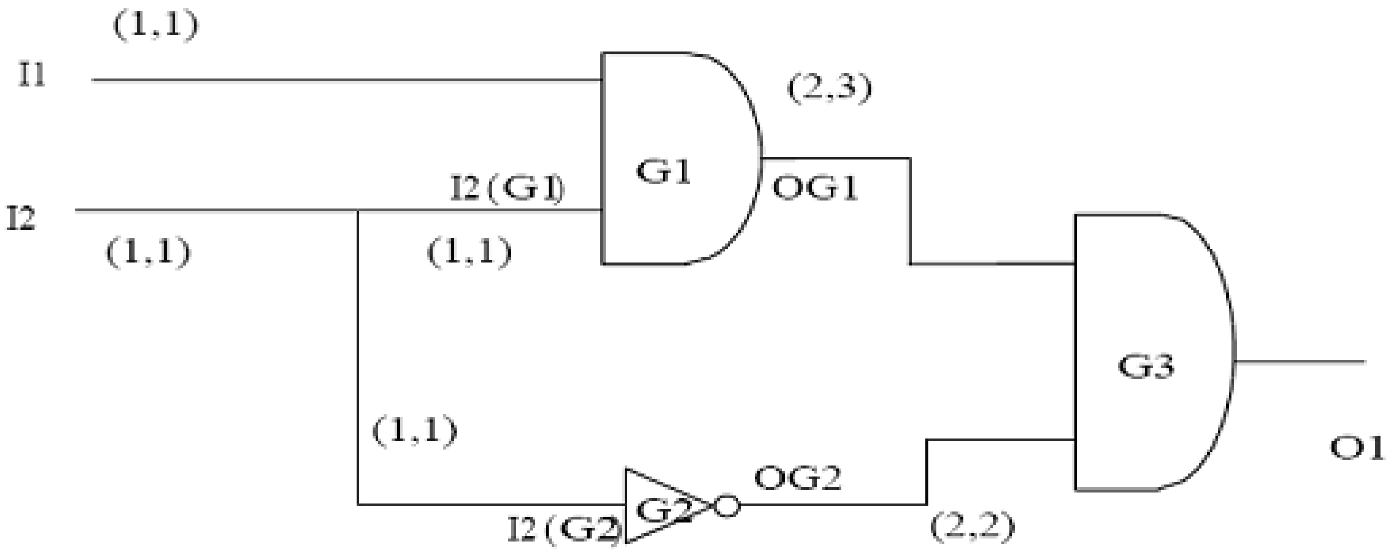
$$CO(a)=\min[CO(c1),CO(c2),\ldots CO(cn)]$$

Combinational Controllability Calculation for a Simple Circuit



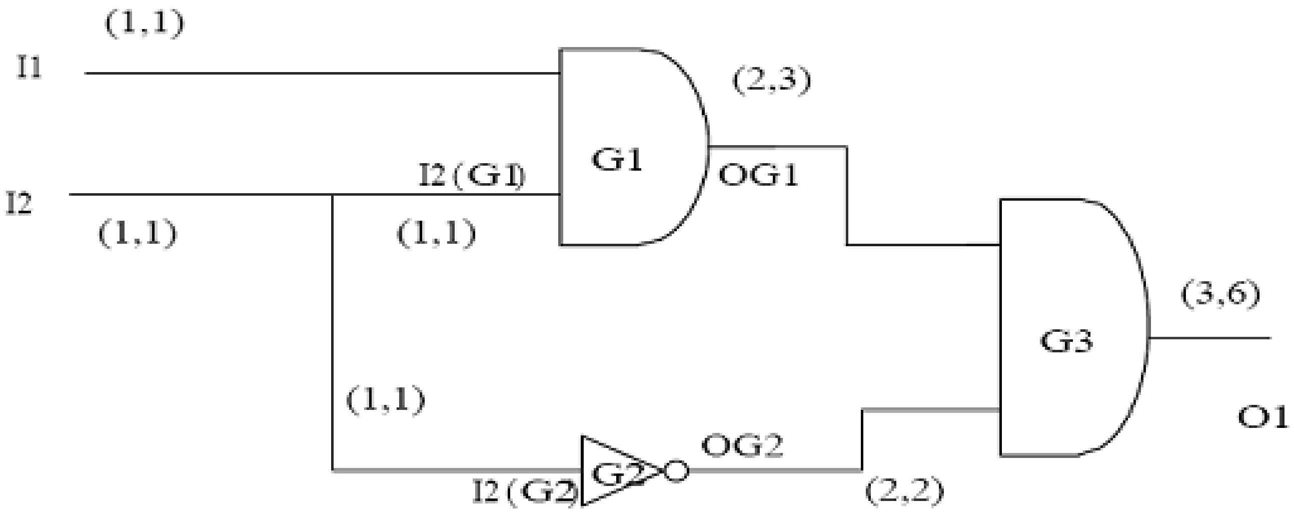
(A) Level1 computation

Combinational Controllability Calculation for a Simple Circuit



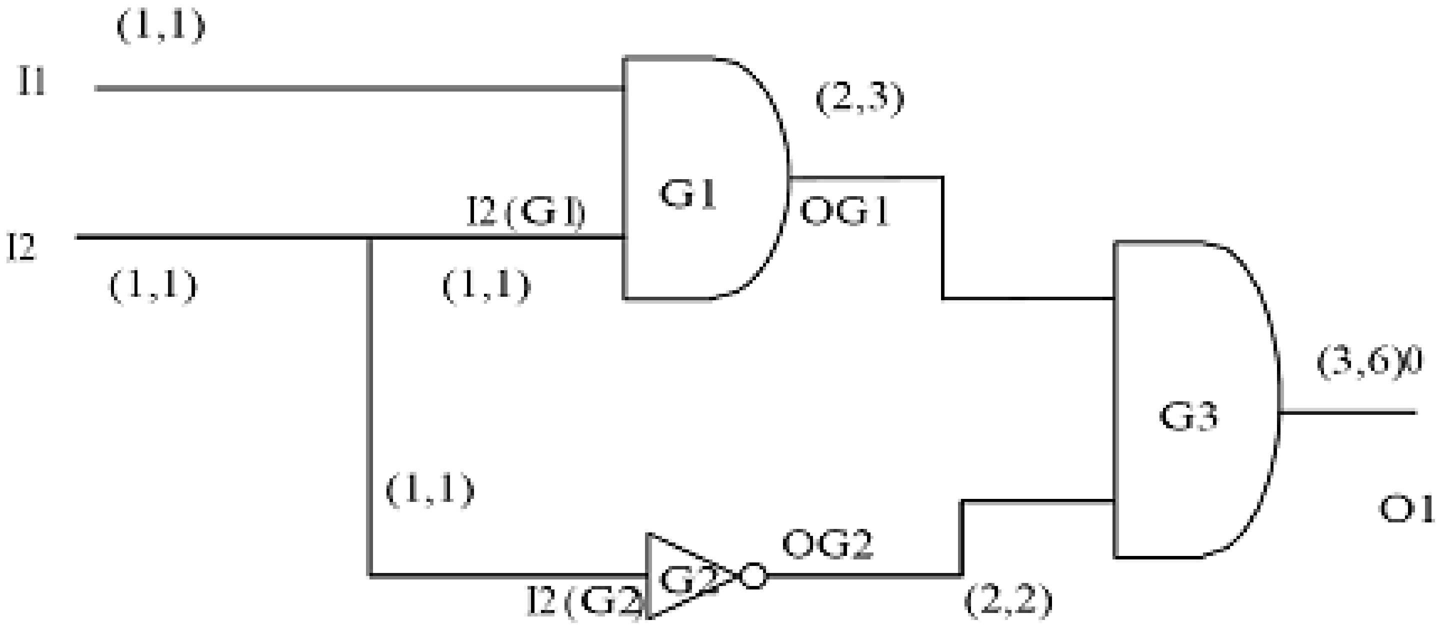
(B) Level2 computation

Combinational Controllability Calculation for a Simple Circuit



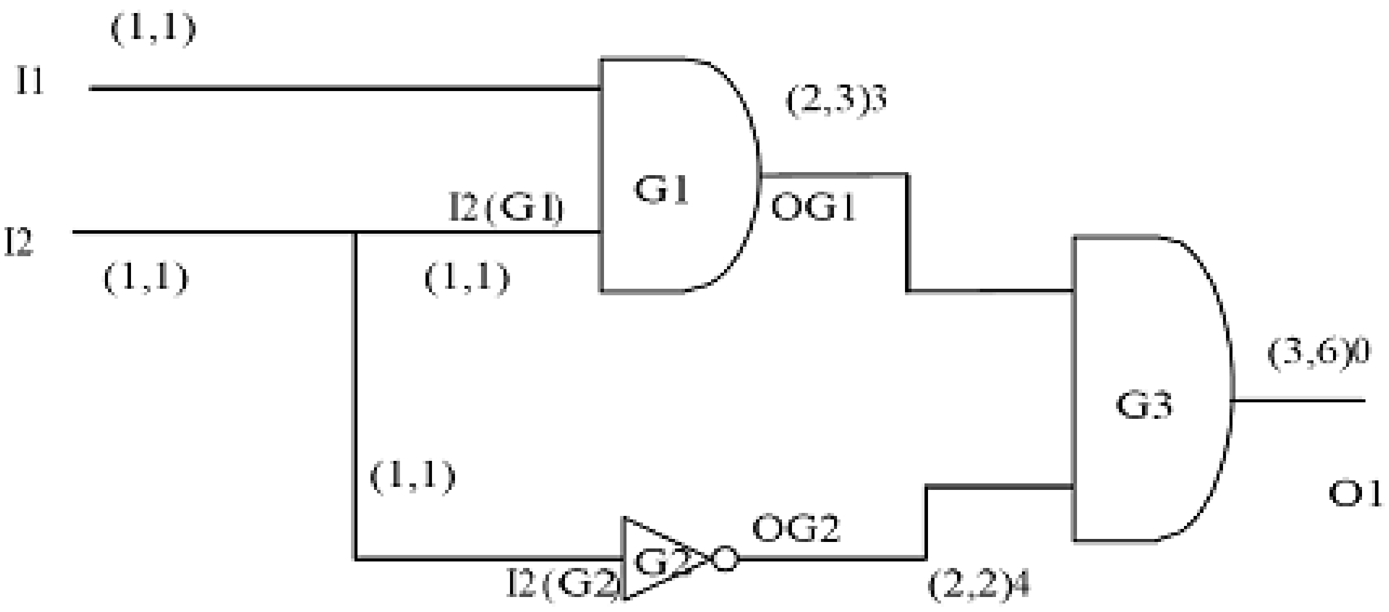
(C) Level3 computation

Combinational Observability Calculation for a Simple Circuit



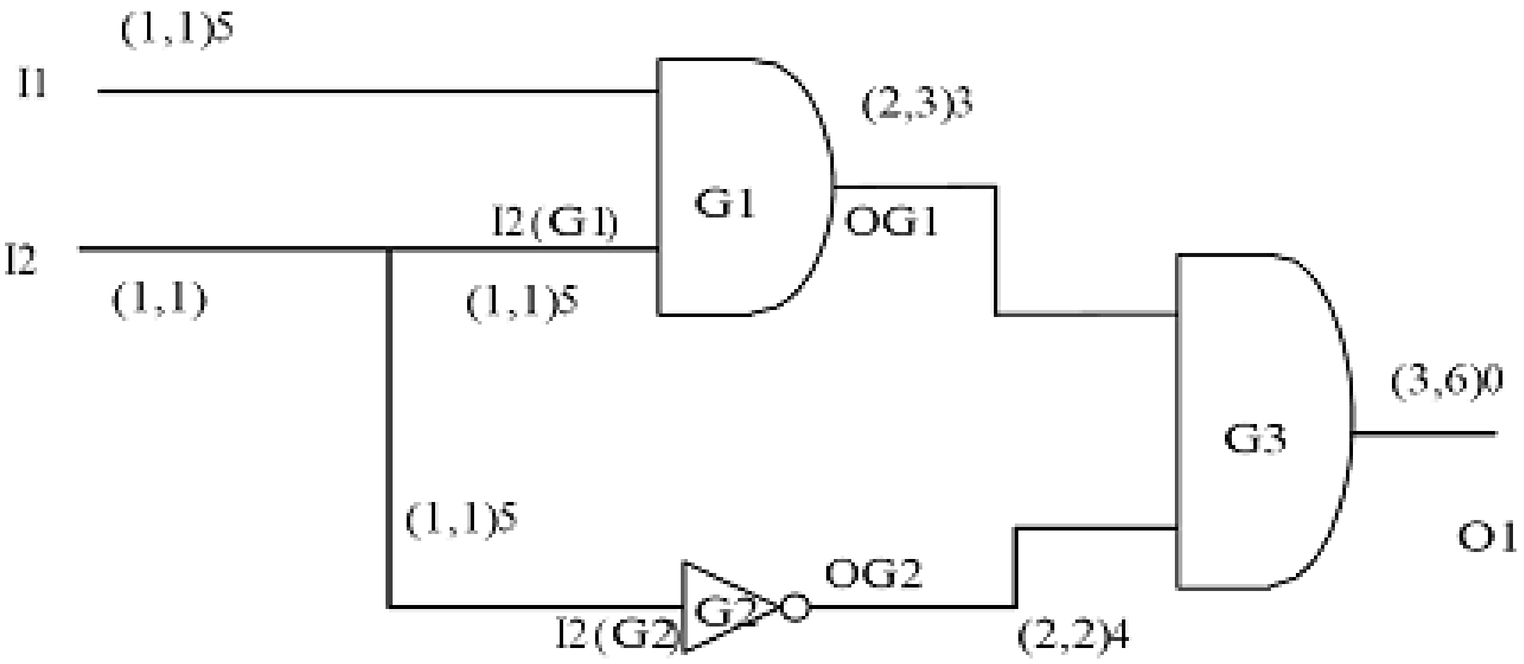
(A) Level1 computation

Combinational Observability Calculation for a Simple Circuit



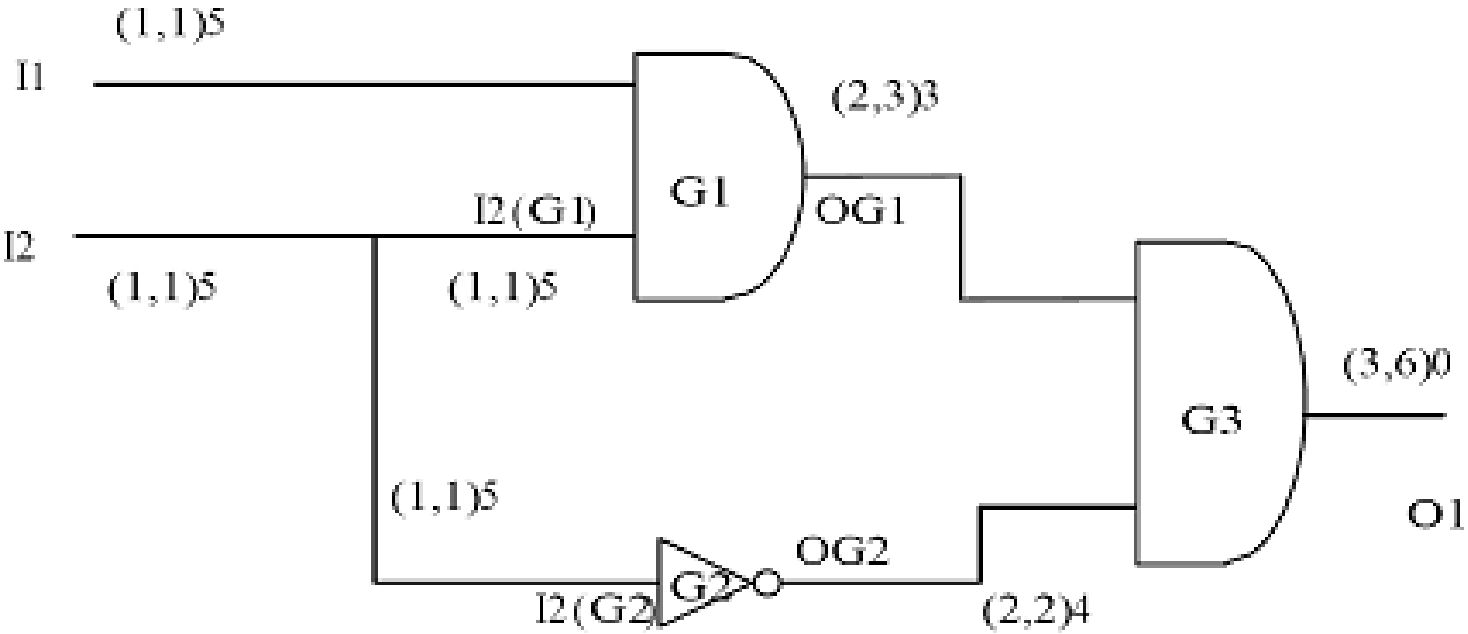
(B) Level2 computation

Combinational Observability Calculation for a Simple Circuit



(C) Level3 computation

Combinational Observability Calculation for a Simple Circuit

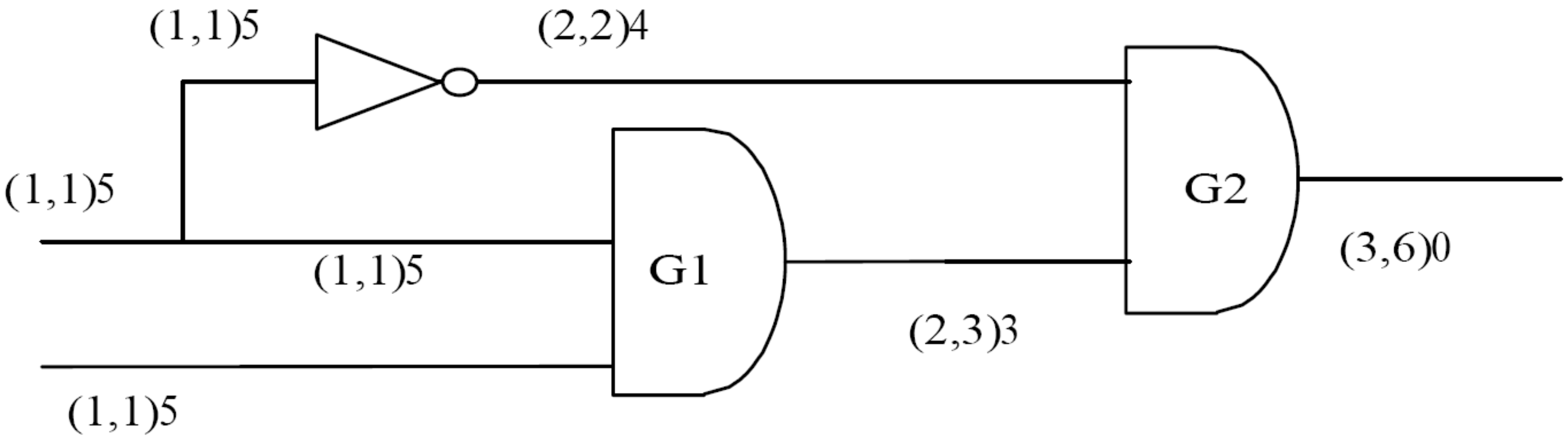


(D) Level4 computation

Questions and Answers

Question: SCOAP is a fast heuristic to compute difficulty in controlling and observing signals in a net in a circuit. Why is the method approximate/not accurate?

Answer: In SCOAP all the branches of a fanout net are considered independent. In the circuit given below, primary output (G2's output) can never have the value of 1 because of dependency of inputs of G2, which in turn arises because of the fanout at the input. However, CC1(G2) is 6 and not infinity.
Similar type of inadequacies arise for observabilities.



Thank You