Advanced Digital Logic Design – EECS 303

http://ziyang.eecs.northwestern.edu/eecs303/

Teacher:	Robert Dick
Office:	L477 Tech
Email:	dickrp@northwestern.edu
Phone:	847–467–2298

NORTHWESTERN
UNIVERSITY

Outline

Introduction to testing

- After fabrication, some circuits don't work correctly
- Determining which circuits contain faults requires testing
- Testing some types of circuits is easy, testing others is difficult
  - One can use automation to help solve this problem

Introduction to testing

- Determining the best way to test large circuits requires automation
- Building large circuits that are easy to test also requires automation
- Testing it spans all design, from system to physical level
- Today, I'll introduce some classical ideas at the logic level

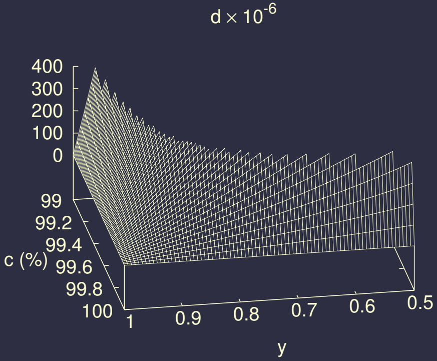
Section outline

Yield

- Yield, y, is the fraction of fault-free products
- Test fault coverage, c, is the fraction of faults that the set of applied tests detects
  - Low-yield is expensive because fabrication capacity is wasted
- Defect level, d, is the fraction of parts containing undetected faults
  - $d = 1 - y^{1-c}$
  - A high defect level is extremely expensive because it means circuits make their way into products, or worse, to consumers, before faults are discovered

Defect level

- Example acceptable defect level: ~ 0.0002
- Either yield must be very high or fault coverage must be very high



Section outline

Faults and failures

- A *fault* is a physical defect in a circuit
- A *failure* is the deviation of a circuit from its specified behavior
- Faults can cause failures but they don't always cause failures

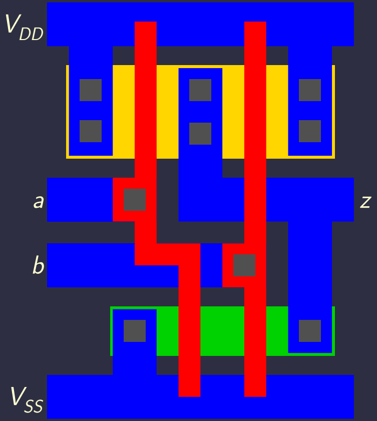
Functional testing

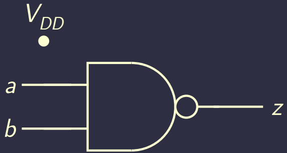
- No assumptions about types of fault
- No assumptions about structure or properties of Circuit Under Test (CUT)
- The CUT is a black box that is checked to determine whether it responds to all (or most) input (sequences) as specified
- However, ignoring structural information makes testing unnecessarily slow

Structural testing

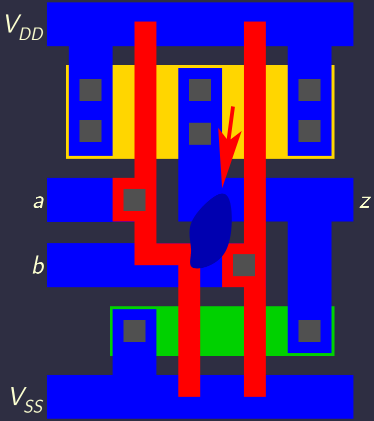
- Use information about the specific CUT
  - Types of faults that are likely to occur
  - Structure of circuit

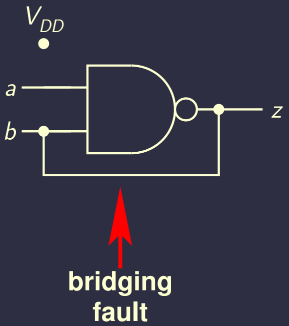
Fault models



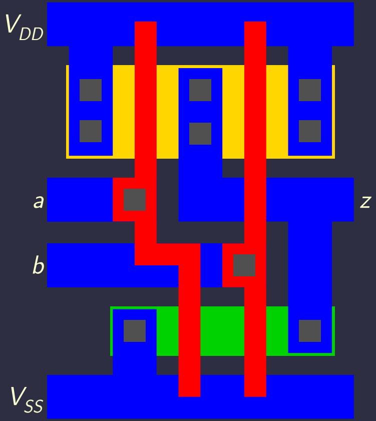


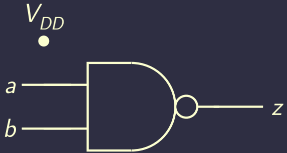
Fault models



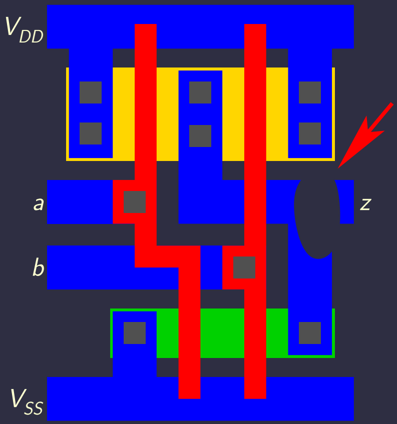


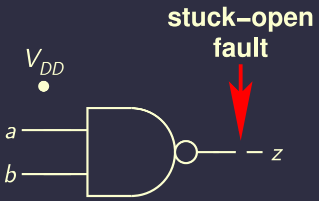
Fault models



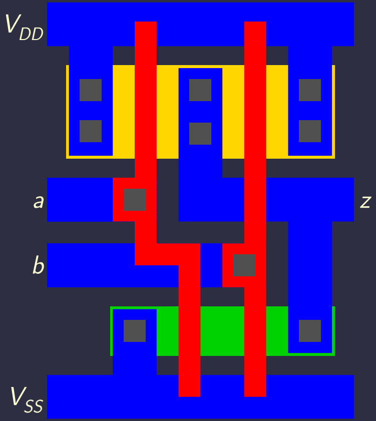


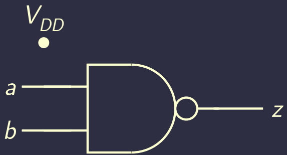
Fault models



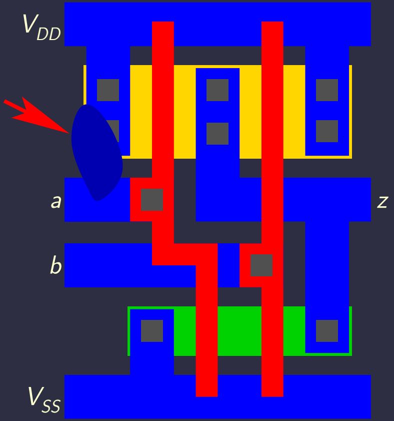


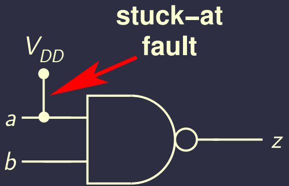
Fault models



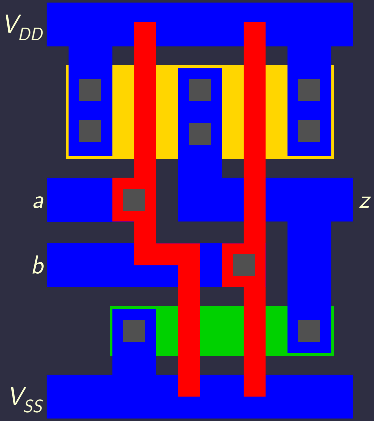


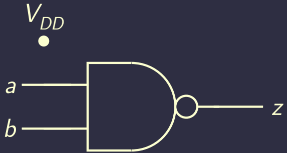
Fault models





Fault models





Singe stuck-at faults

- One of the simplest and most common fault models
  - S-a-0: Stuck-at 0
  - S-a-1: Stuck-at 1
- Relies on digital reinforcement
  - $0.8 \cdot V_{DD}$ is $1 \cdot V_{DD}$ one logic stage later
  - S-a-0.8 $\approx$ s-a-1
- For two-level logic, exhaustive test set for single stuck-at faults will detect all multiple stuck-at faults, too
  - Doesn't hold for multi-level logic

Single stuck-at faults

Consider a NAND3 gate

Inputs			fault	z							
			free	a		b		c		z	
A	B	C		0	1	0	1	0	1	0	1
0	0	0	1	1	1	1	1	1	1	0	1
0	0	1	1	1	1	1	1	1	1	0	1
0	1	0	1	1	1	1	1	1	1	0	1
0	1	1	1	1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	0	1	1	0	1
1	1	0	1	1	1	1	1	1	0	0	1
1	1	1	0	1	0	1	0	1	0	0	1

Single stuck-at faults

Consider a NAND3 gate

Inputs			fault free	z							
				a		b		c		z	
A	B	C		0	1	0	1	0	1	0	1
0	0	0	1	1	1	1	1	1	1	0	1
0	0	1	1	1	1	1	1	1	1	0	1
0	1	0	1	1	1	1	1	1	1	0	1
0	1	1	1	1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	0	1	1	0	1
1	1	0	1	1	1	1	1	1	0	0	1
1	1	1	0	1	0	1	0	1	0	0	1

Too expensive to use 2^n inputs ($2^3 = 8$ in this case)

Single stuck-at faults

Consider a NAND3 gate

Inputs			fault	z							
				a		b		c		z	
A	B	C	free	0	1	0	1	0	1	0	1
0	0	0	1	1	1	1	1	1	1	0	1
0	0	1	1	1	1	1	1	1	1	0	1
0	1	0	1	1	1	1	1	1	1	0	1
0	1	1	1	1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	0	1	1	0	1
1	1	0	1	1	1	1	1	1	0	0	1
1	1	1	0	1	0	1	0	1	0	0	1

Unate covering again

Fault coverage

- A test of all possible inputs for a NANDn gate will require 2^n tests
- Applying all possible tests (or sequences of tests) for complex circuits isn't practical
- However, for all NANDn gates, all single stuck-at faults (and implicitly multiple stuck-at faults) can be tested using only $n + 1$ tests

Fault equivalence

- The function of the circuit is identical in the presence of either fault
- E.g., any input of a NAND gate being s-a-0 or the output being s-a-1 $\longrightarrow$ output of 1
- In general, identifying equivalent faults, or *fault collapsing* is difficult
- However, can use knowledge about gates, and connectivities, to find most equivalent faults

Fault equivalence

Consider a NAND3 gate

Inputs			fault	z							
				a		b		c		z	
A	B	C	free	0	1	0	1	0	1	0	1
0	0	0	1	1	1	1	1	1	1	0	1
0	0	1	1	1	1	1	1	1	1	0	1
0	1	0	1	1	1	1	1	1	1	0	1
0	1	1	1	1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	0	1	1	0	1
1	1	0	1	1	1	1	1	1	0	0	1
1	1	1	0	1	0	1	0	1	0	0	1

Fault equivalence

Consider a NAND3 gate

Inputs			fault	z							
				a		b		c		z	
A	B	C	free	0	1	0	1	0	1	0	1
0	0	0	1	1	1	1	1	1	1	0	1
0	0	1	1	1	1	1	1	1	1	0	1
0	1	0	1	1	1	1	1	1	1	0	1
0	1	1	1	1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	0	1	1	0	1
1	1	0	1	1	1	1	1	1	0	0	1
1	1	1	0	1	0	1	0	1	0	0	1

Equivalent faults

Other fault models

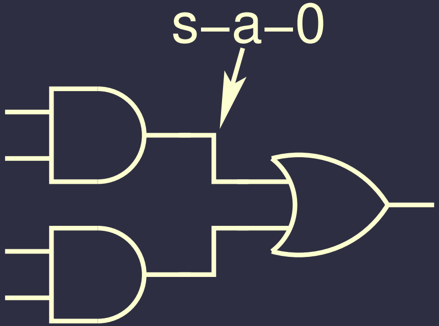
- $\sim {}^{2}\!/_{3}$ CMOS faults are not stuck-at faults
  - Luckily, stuck-at fault tests often detect them
- More advanced faults models
  - Allow higher theoretical coverage
  - Are more complicated
- In this lecture, we only have time to cover stuck-at-faults

Section outline

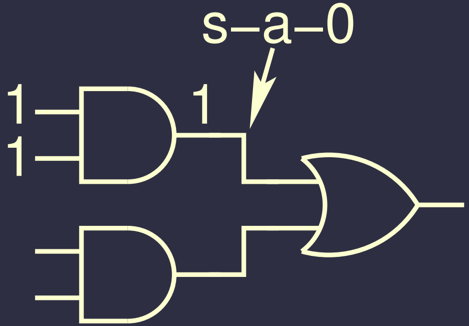
Test generation

- Can use algorithms to automatically generate a test for a specific fault
- Problem: Identify some test, t, such that the output of the circuit will deviate from its specified value if a particular fault, f, is present
- Excitation: Need to propagate a value to stuck-at fault that is contrary to the fault value
- Sensitization: Need to propagate the faulty value to an output of the circuit

Excitation and sensitization

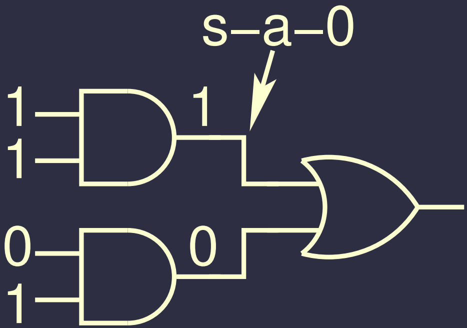


Excitation and sensitization



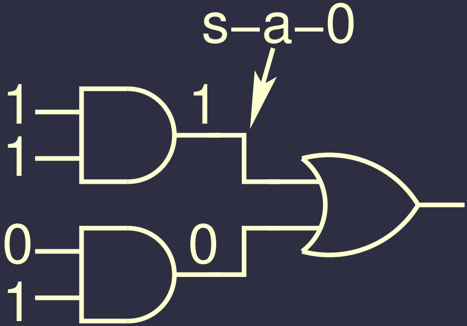
excitation

Excitation and sensitization



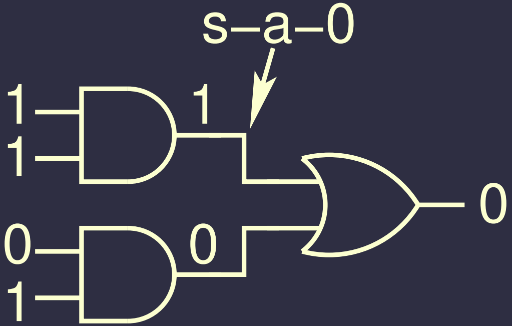
sensitization

Excitation and sensitization



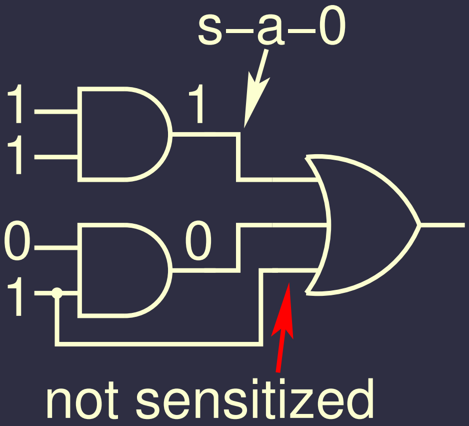
sensitization

Excitation and sensitization

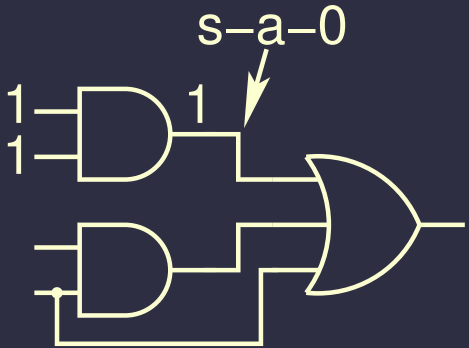


sensitization

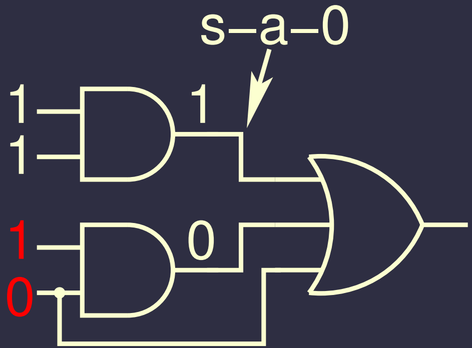
Backtracking



Backtracking

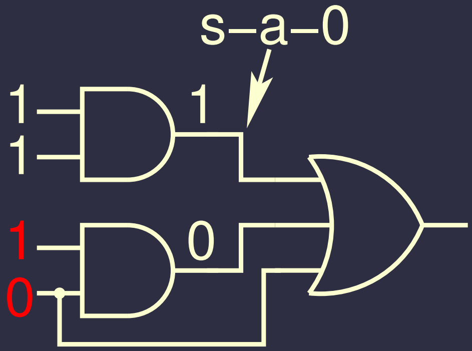


Backtracking



sensitized

Backtracking

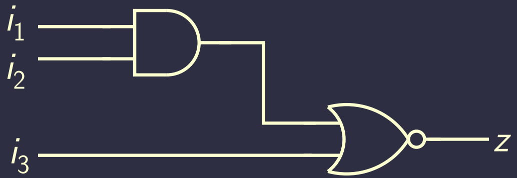


sensitized

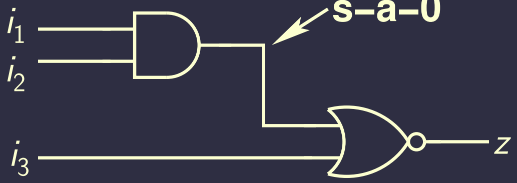
Automatic Test Pattern Generation (ATPG)

- Can automatically determine test for a particular fault
- Boolean difference
  - Generally considered too slow for use on large circuits
  - Easy to understand
- Excitation/sensitization based methods ($\mathcal{D}$-algorithm)

Boolean difference

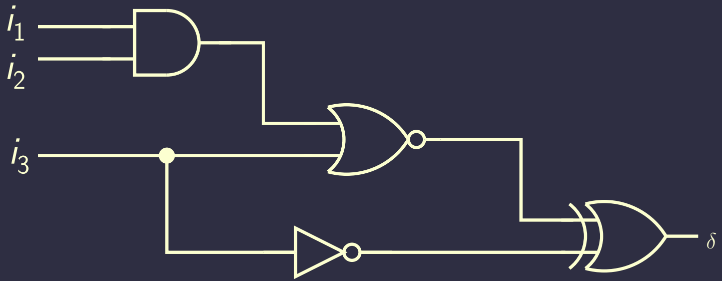


Boolean difference



In the presence of the fault, function is $\overline{i_3}$

Boolean difference



XOR faulty and specified functions

Boolean difference

- If the resulting function is 0, the fault can not be identified
- If the resulting function is 1, the fault can be identified
    - Use a test that satisfies (sets to 1) the function

Boolean difference

	00	01	11	10
0	1	0	0	1
1	1	0	0	0

	00	01	11	10
0	1	0	0	1
1	1	0	0	1

Boolean difference

	00	01	11	10
0	1	0	0	1
1	1	0	0	$\boxed{0}$

	00	01	11	10
0	1	0	0	1
1	1	0	0	$\boxed{1}$

Boolean difference

	00	01	11	10
0	1	0	0	1
1	1	0	0	0

	00	01	11	10
0	0	0	0	0
1	0	0	0	1

	00	01	11	10
0	1	0	0	1
1	1	0	0	1

Boolean difference

	00	01	11	10
0	1	0	0	1
1	1	0	0	0

	00	01	11	10
0	0	0	0	0
1	0	0	0	1

	00	01	11	10
0	1	0	0	1
1	1	0	0	1

$\mathcal{D}$-algorithm

- We have seen that, in order to test for a particular fault, we must
  - Excite the fault
  - Sensitize a path from the fault to the output
- The $\mathcal{D}$-algorithm is an automated method of finding a test that excites a fault and sensitizes a path to the circuit output
- J.P. Roth, IBM, 1966

Roth's extension to Boolean algebra

- α/β
  - α is the fault-free value
  - β is the value in the presence of the fault
- $0/0 = 0$
- $1/1 = 1$
- $1/0 = \mathcal{D}$
- $0/1 = \overline{\mathcal{D}}$
- $X/X = X$

(N)AND $\mathcal{D}$-cubes of failure

gate	detection input		$\mathcal{D}$-cube of failure			
			z			
	a	b		a	b	z
AND	1	1	s-a-0	1	1	$\mathcal{D}$
	0	X	s-a-1	0	X	$\overline{\mathcal{D}}$
	X	0	s-a-1	X	0	$\overline{\mathcal{D}}$
NAND	1	1	s-a-1	1	1	$\overline{\mathcal{D}}$
	0	X	s-a-0	0	X	$\mathcal{D}$
	X	0	s-a-0	X	0	$\mathcal{D}$

(N)OR $\mathcal{D}$-cubes of failure

gate	detection input		z	$\mathcal{D}$-cube of failure		
	a	b		a	b	z
OR	0	0	s-a-1	0	0	$\overline{\mathcal{D}}$
	1	X	s-a-0	1	X	$\mathcal{D}$
	X	1	s-a-0	X	1	$\mathcal{D}$
NOR	0	0	s-a-0	0	0	$\mathcal{D}$
	1	X	s-a-1	1	X	$\overline{\mathcal{D}}$
	X	1	s-a-1	X	1	$\overline{\mathcal{D}}$

(N)AND $\mathcal{D}$-cubes of propagation

input		AND	NAND
a	b	z	z
1	$\mathcal{D}$	$\mathcal{D}$	$\overline{\mathcal{D}}$
1	$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$	$\mathcal{D}$
$\mathcal{D}$	1	$\mathcal{D}$	$\overline{\mathcal{D}}$
$\overline{\mathcal{D}}$	1	$\overline{\mathcal{D}}$	$\mathcal{D}$
$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$	$\overline{\mathcal{D}}$
$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$	$\mathcal{D}$

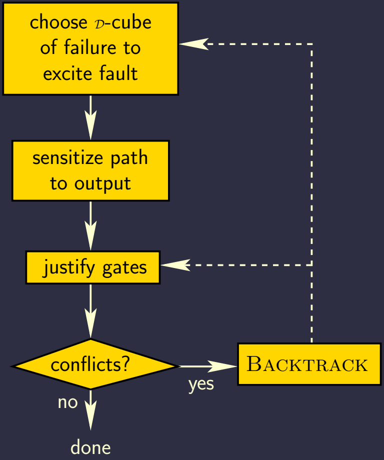
(N)OR $\mathcal{D}$-cubes of propagation

input		OR	NOR
a	b	z	z
0	$\mathcal{D}$	$\mathcal{D}$	$\overline{\mathcal{D}}$
0	$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$	$\mathcal{D}$
$\mathcal{D}$	0	$\mathcal{D}$	$\overline{\mathcal{D}}$
$\overline{\mathcal{D}}$	0	$\overline{\mathcal{D}}$	$\mathcal{D}$
$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$	$\overline{\mathcal{D}}$
$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$	$\mathcal{D}$

$\mathcal{D}$-algorithm

1. Initially, all the lines in the circuit are X (Don't Care)
   - Left blank in example
2. A gate with an input of $\mathcal{D}$ or $\overline{\mathcal{D}}$ and an output of X belongs to the *frontier*
3. An element whose assigned inputs do not imply the output is *unjustified*
4. We want to drive the frontier to a circuit output (sensitize)...
5. ...and justify the gate inputs (excite)

$\mathcal{D}$-algorithm overview



$\mathcal{D}$-algorithm

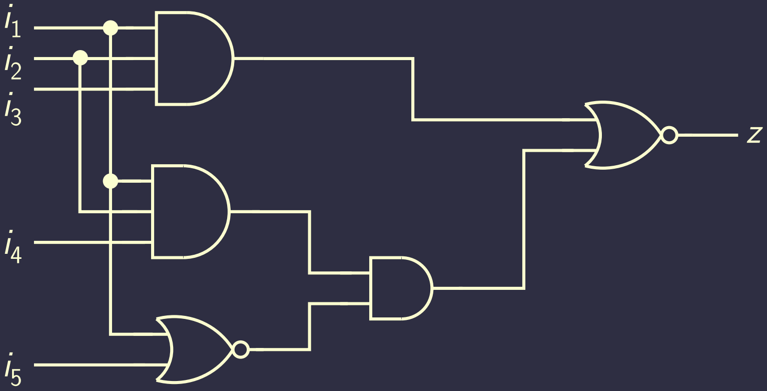
```
Select a $\mathcal{D}$-cube of failure to excite the fault
while frontier has not yet reached an output do
    Perform the implications of the most recent assignment
    if the frontier is empty then
        BACKTRACK
    end if
    Choose a signal, $s$, s.t. $s$ can't reached from the fault
    Assign $s$ to a value in order to propagate the fault forward
end while
for each unjustified line, $l$ do
    if $l$ can be justified then
        Justify $l$
    else
        BACKTRACK
    end if
end for
A test has been found
```

$\mathcal{D}$-algorithm notes

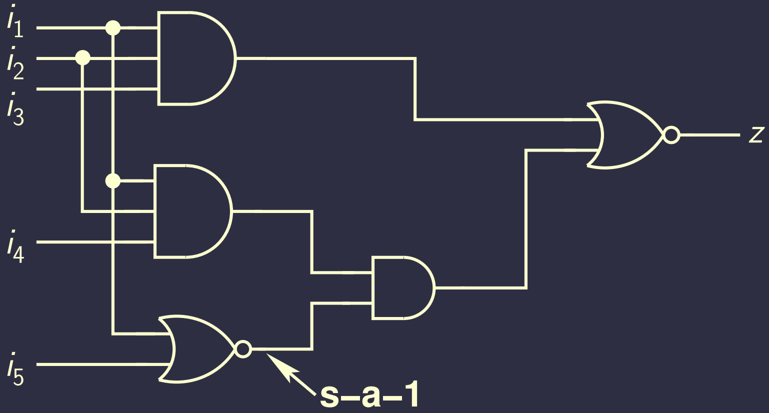
There are a few details not explained in the pseudocode

- Signals should be selected in order to drive the frontier forward
- Backtracking is the action of backing up and taking the closest previously unexplored branch in the decision tree

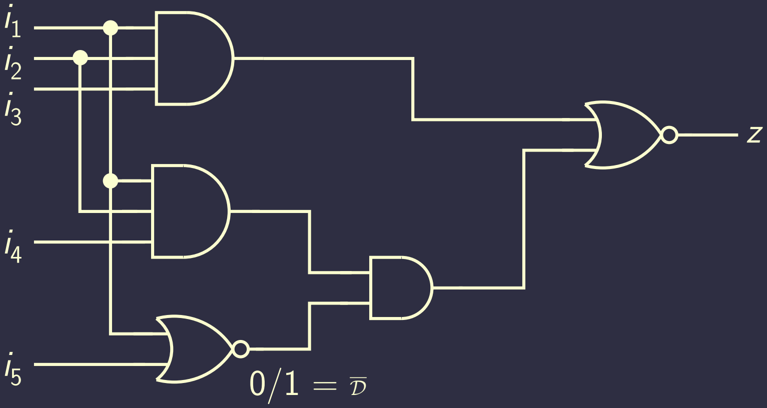
$\mathcal{D}$-algorithm example



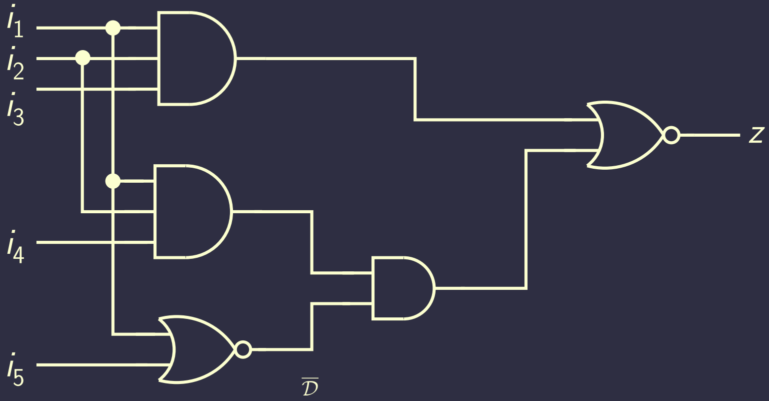
$\mathcal{D}$-algorithm example



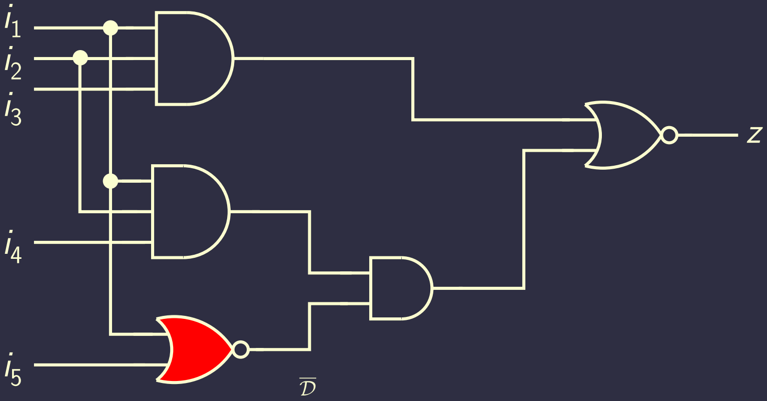
$\mathcal{D}$-algorithm example



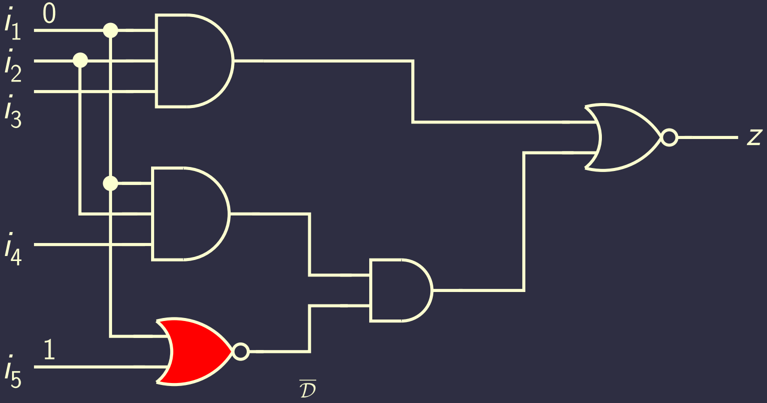
$\mathcal{D}$-algorithm example



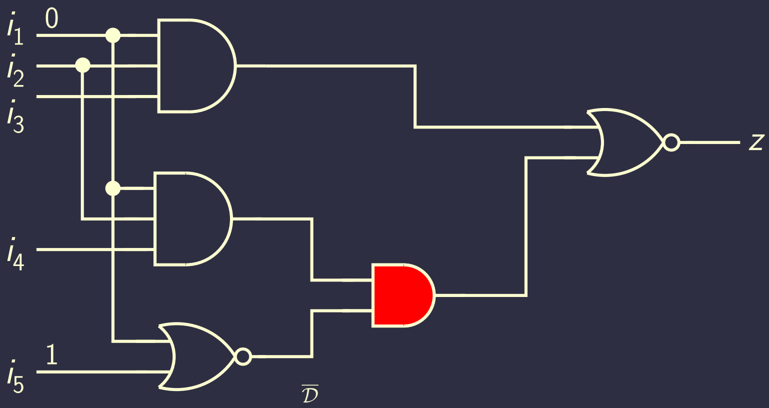
$\mathcal{D}$-algorithm example



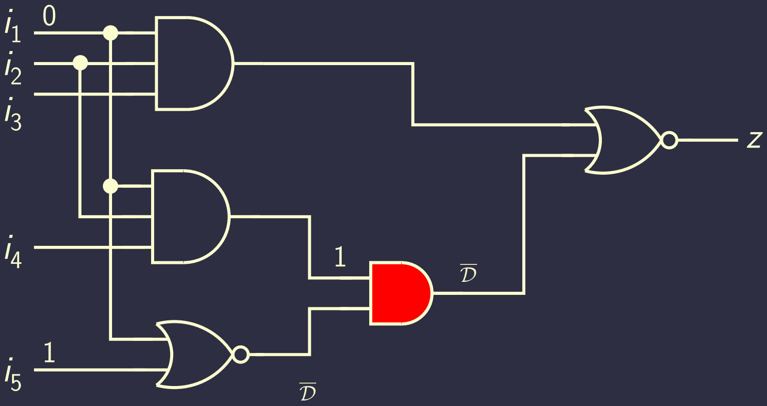
$\mathcal{D}$-algorithm example



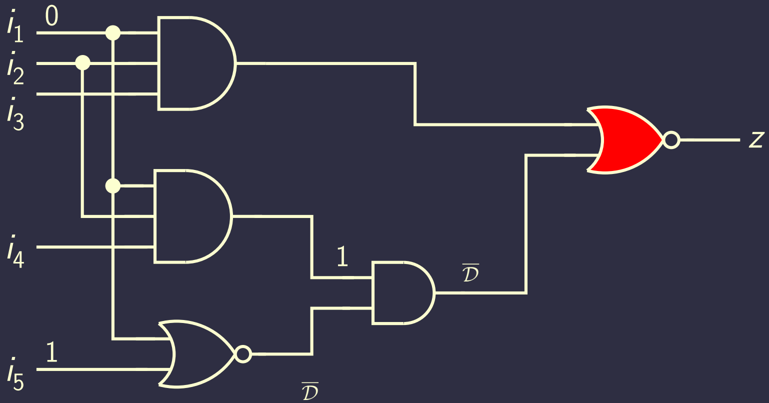
$\mathcal{D}$-algorithm example



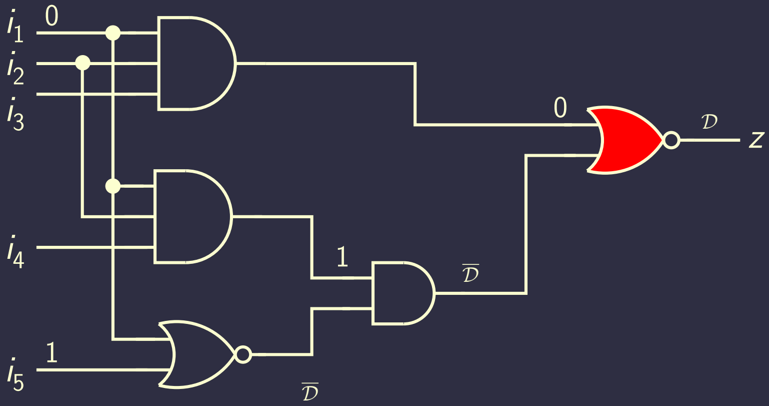
$\mathcal{D}$-algorithm example



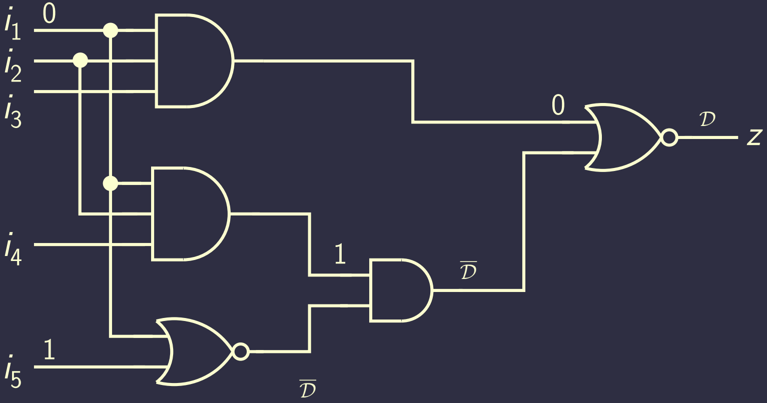
$\mathcal{D}$-algorithm example



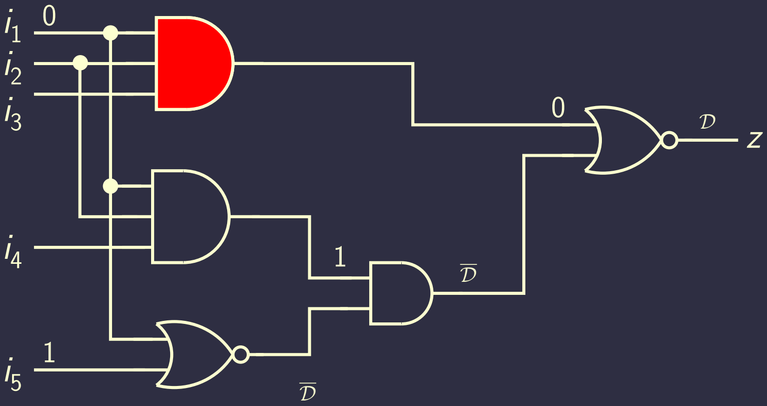
$\mathcal{D}$-algorithm example



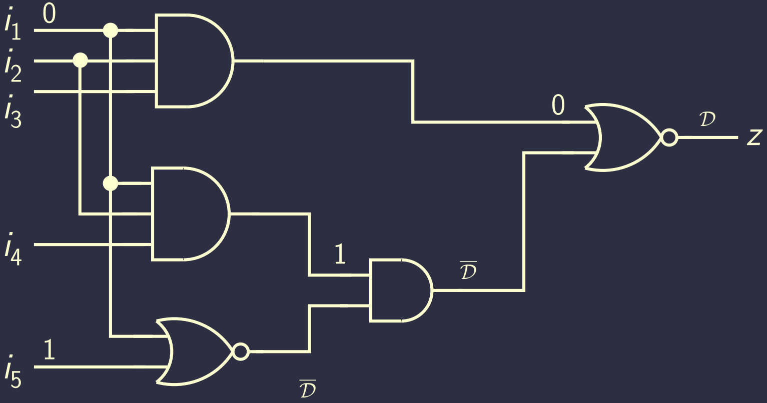
$\mathcal{D}$-algorithm example



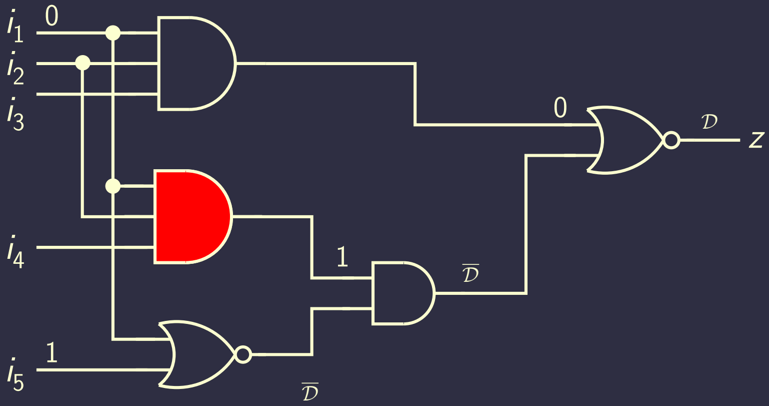
$\mathcal{D}$-algorithm example



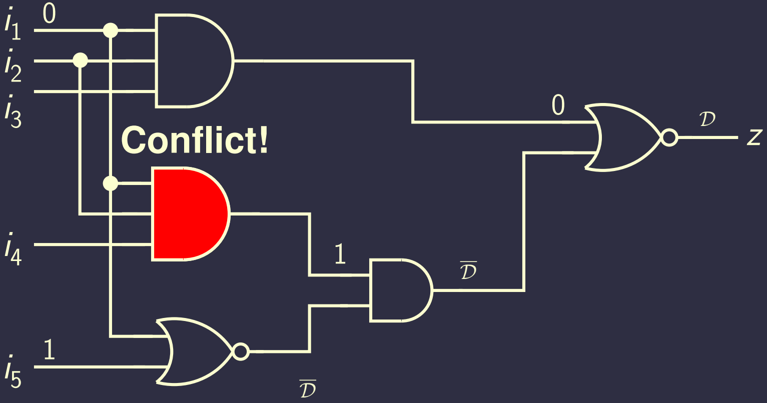
$\mathcal{D}$-algorithm example



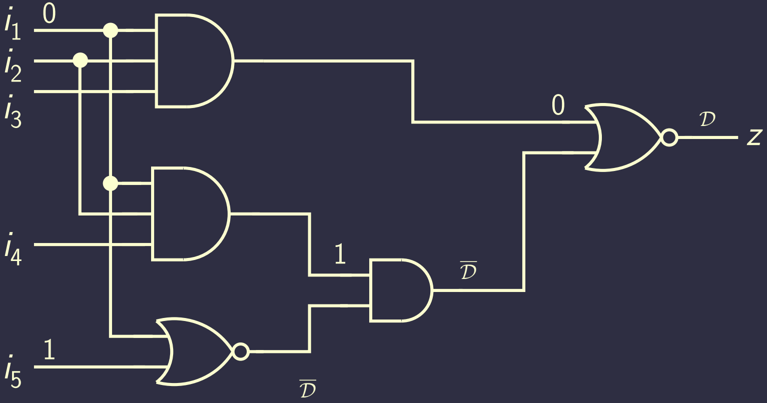
$\mathcal{D}$-algorithm example



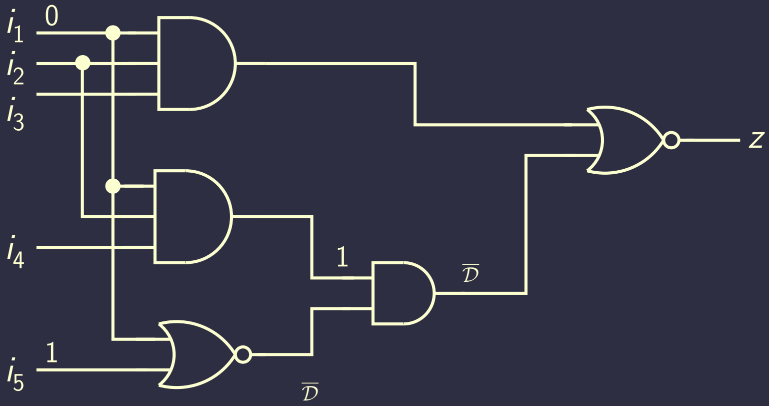
$\mathcal{D}$-algorithm example



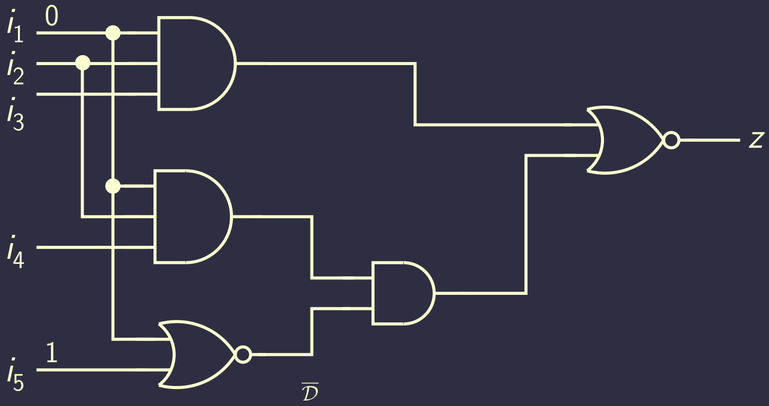
$\mathcal{D}$-algorithm example



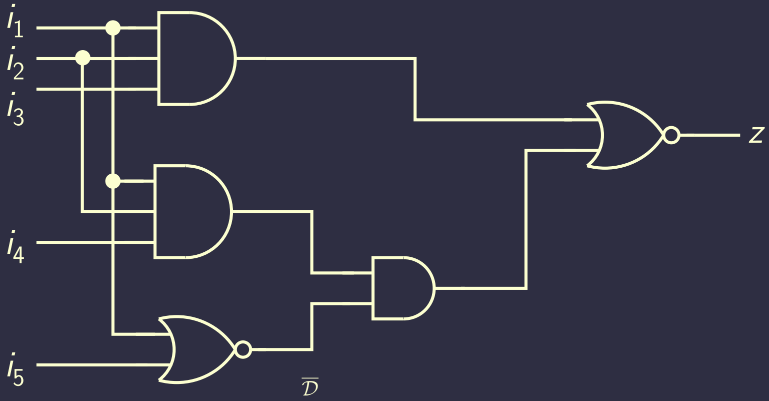
$\mathcal{D}$-algorithm example



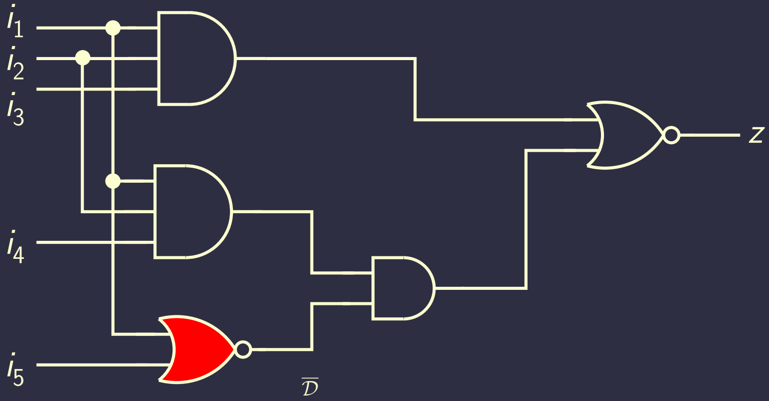
$\mathcal{D}$-algorithm example



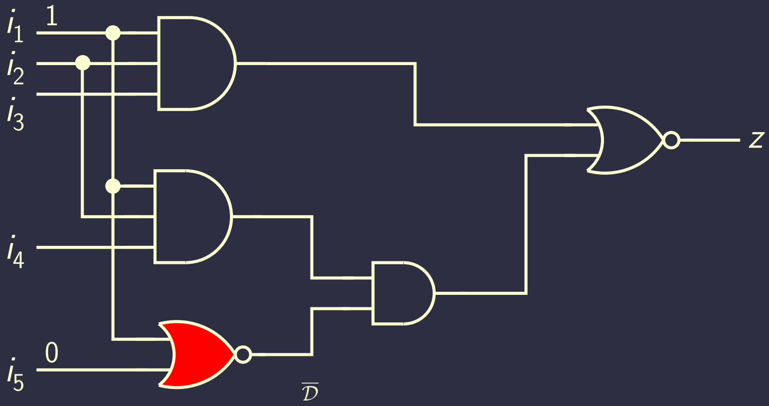
$\mathcal{D}$-algorithm example



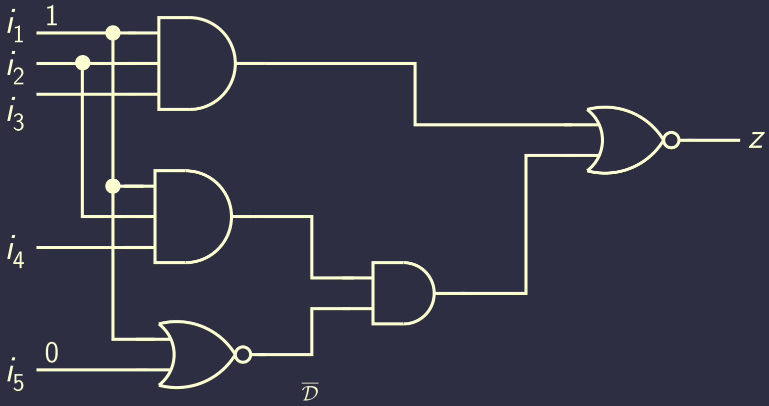
$\mathcal{D}$-algorithm example



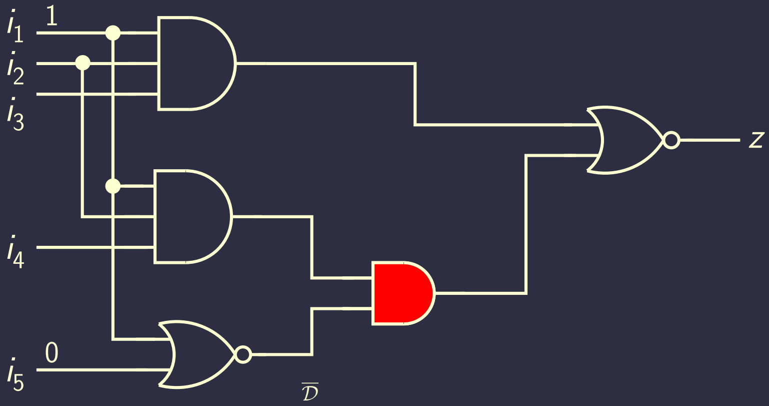
$\mathcal{D}$-algorithm example



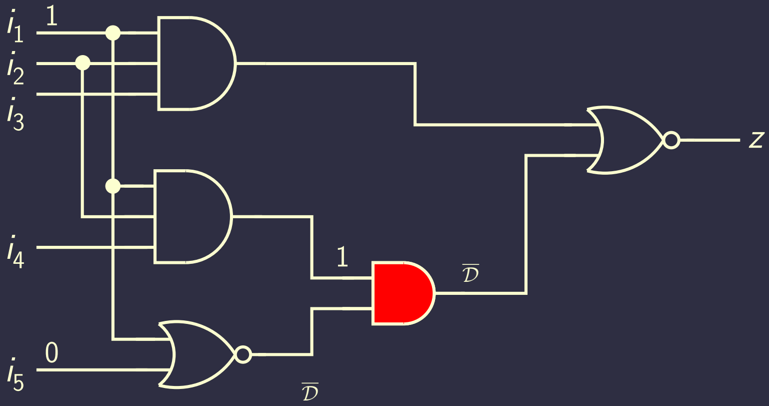
$\mathcal{D}$-algorithm example



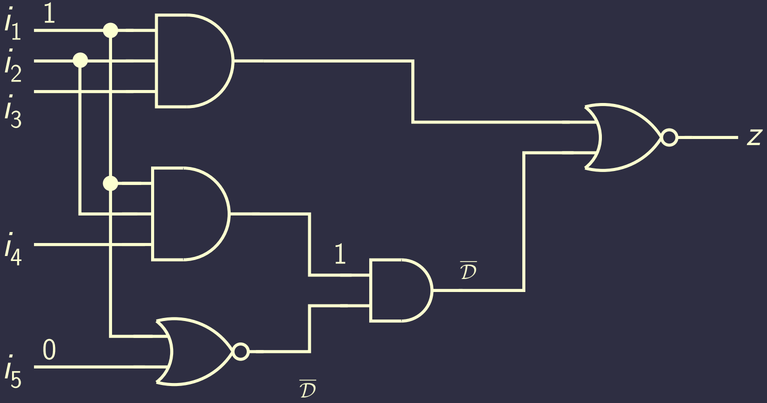
$\mathcal{D}$-algorithm example



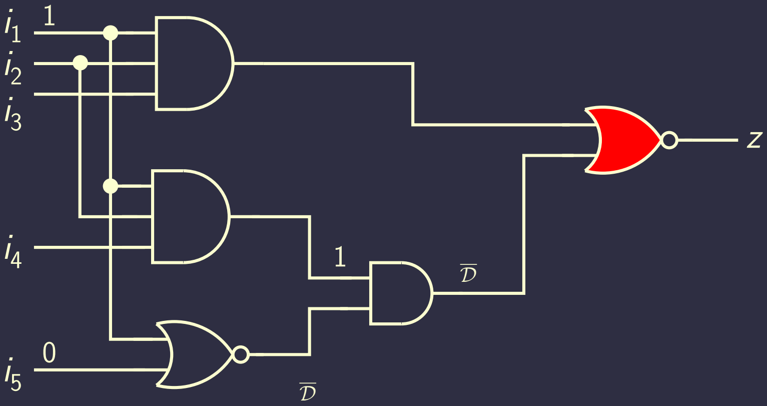
$\mathcal{D}$-algorithm example



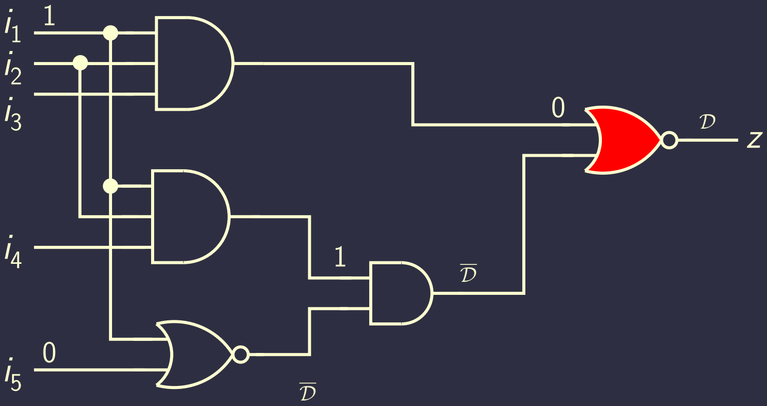
$\mathcal{D}$-algorithm example



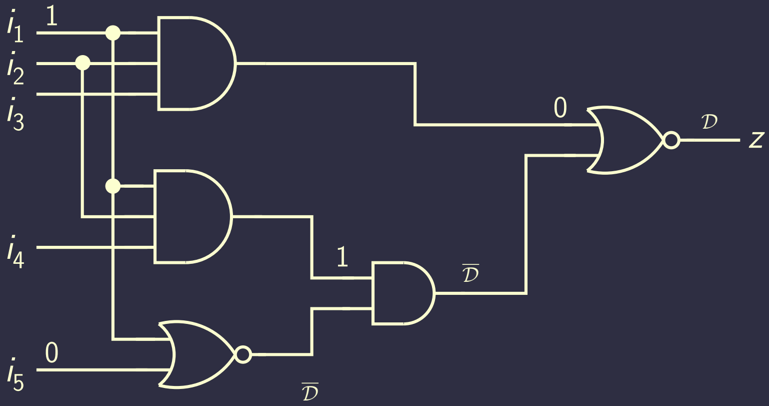
$\mathcal{D}$-algorithm example



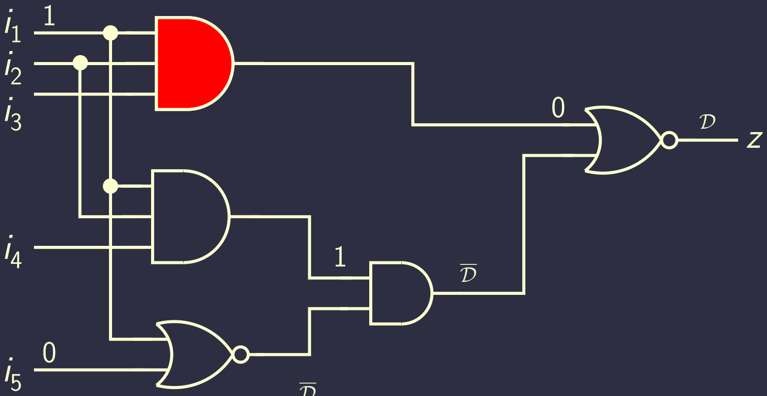
$\mathcal{D}$-algorithm example



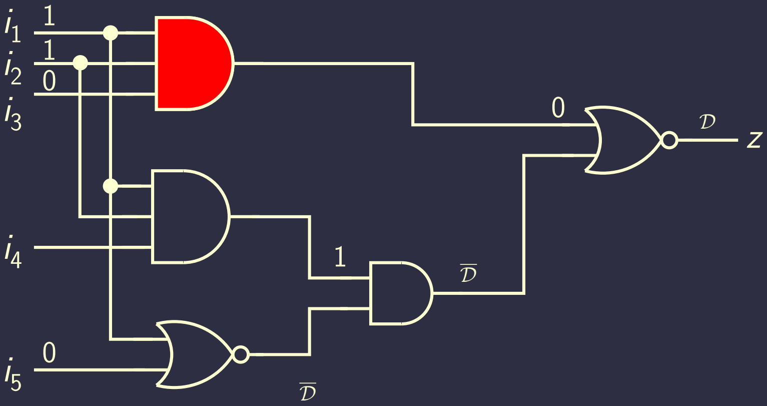
$\mathcal{D}$-algorithm example



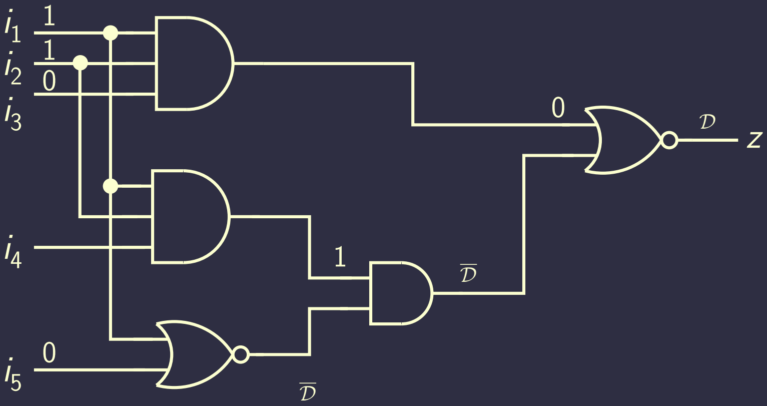
$\mathcal{D}$-algorithm example



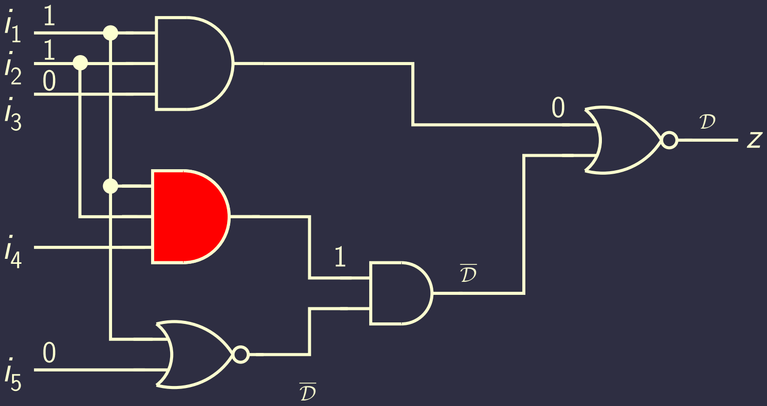
$\mathcal{D}$-algorithm example



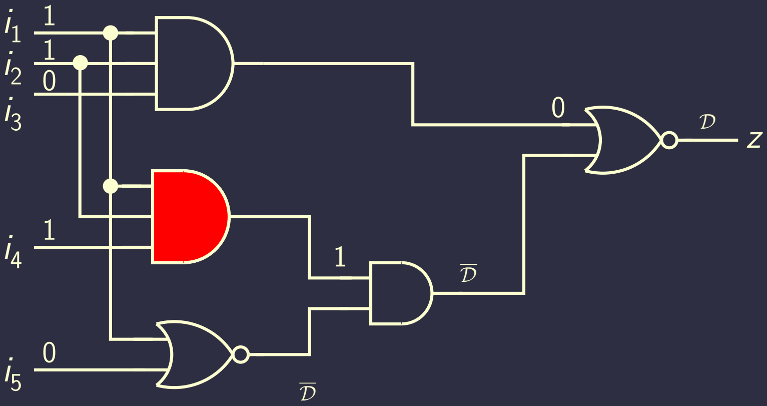
$\mathcal{D}$-algorithm example



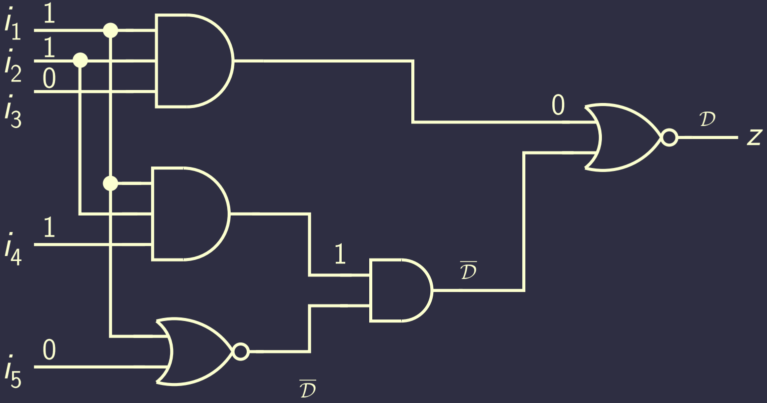
$\mathcal{D}$-algorithm example



$\mathcal{D}$-algorithm example



$\mathcal{D}$-algorithm example



NOR $\mathcal{D}$-cubes of failure

gate	detection input		$\mathcal{D}$-cube of failure			
			z			
	a	b		a	b	z
NOR	0	0	s-a-0	0	0	$\mathcal{D}$
	0	1	s-a-1	0	1	$\overline{\mathcal{D}}$
	1	0	s-a-1	1	0	$\overline{\mathcal{D}}$
	1	1	s-a-1	1	1	$\overline{\mathcal{D}}$

NOR $\mathcal{D}$-cubes of failure

gate	detection input		$\mathcal{D}$-cube of failure			
			z			
	a	b		a	b	z
NOR	0	0	s-a-0	0	0	$\mathcal{D}$
	0	1	s-a-1	0	1	$\overline{\mathcal{D}}$
	1	0	s-a-1	1	0	$\overline{\mathcal{D}}$
	1	1	s-a-1	1	1	$\overline{\mathcal{D}}$

AND $\mathcal{D}$-cubes of propagation

input		AND
a	b	z
1	$\mathcal{D}$	$\mathcal{D}$
1	$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$
$\mathcal{D}$	1	$\mathcal{D}$
$\overline{\mathcal{D}}$	1	$\overline{\mathcal{D}}$
$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$
$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$

AND $\mathcal{D}$-cubes of propagation

input		AND
a	b	z
1	$\mathcal{D}$	$\mathcal{D}$
1	$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$
$\mathcal{D}$	1	$\mathcal{D}$
$\overline{\mathcal{D}}$	1	$\overline{\mathcal{D}}$
$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$
$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$

NOR $\mathcal{D}$-cubes of propagation

input		NOR
a	b	z
0	$\mathcal{D}$	$\overline{\mathcal{D}}$
0	$\overline{\mathcal{D}}$	$\mathcal{D}$
$\mathcal{D}$	0	$\overline{\mathcal{D}}$
$\overline{\mathcal{D}}$	0	$\mathcal{D}$
$\mathcal{D}$	$\mathcal{D}$	$\overline{\mathcal{D}}$
$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$	$\mathcal{D}$

NOR $\mathcal{D}$-cubes of propagation

input		NOR
a	b	z
0	$\mathcal{D}$	$\overline{\mathcal{D}}$
0	$\overline{\mathcal{D}}$	$\mathcal{D}$
$\mathcal{D}$	0	$\overline{\mathcal{D}}$
$\overline{\mathcal{D}}$	0	$\mathcal{D}$
$\mathcal{D}$	$\mathcal{D}$	$\overline{\mathcal{D}}$
$\overline{\mathcal{D}}$	$\overline{\mathcal{D}}$	$\mathcal{D}$

NOR $\mathcal{D}$-cubes of failure

gate	detection input			$\mathcal{D}$-cube of failure		
			z			
	a	b		a	b	z
NOR	0	0	s-a-0	0	0	$\mathcal{D}$
	0	1	s-a-1	0	1	$\overline{\mathcal{D}}$
	1	0	s-a-1	1	0	$\overline{\mathcal{D}}$
	1	1	s-a-1	1	1	$\overline{\mathcal{D}}$

NOR $\mathcal{D}$-cubes of failure

gate	detection input		$\mathcal{D}$-cube of failure			
			z			
	a	b		a	b	z
NOR	0	0	s-a-0	0	0	$\mathcal{D}$
	0	1	s-a-1	0	1	$\overline{\mathcal{D}}$
	1	0	s-a-1	1	0	$\overline{\mathcal{D}}$
	1	1	s-a-1	1	1	$\overline{\mathcal{D}}$

$\mathcal{D}$-algorithm summary

- If a test for a fault exists, guaranteed to generate it
    - Full backtracking
- However, much faster than Boolean difference

Caveats

- This lecture has only introduced some topics in testing
  - If you intend to do research in this area, take a few courses on testing
- Single stuck-at fault model is becoming obsolete

Caveats

- $\mathcal{D}$-algorithm is actually more complicated
  - Avoiding backtracking is important for performance
    - Implications
    - Heuristics to make best decisions first
  - Should understand derivation of $\mathcal{D}$-cubes for arbitrary fault models

$\mathcal{D}$-algorithm reference

Stanley Hurst. *VLSI Testing: Digital and Mixed Analogue/Digital Techniques*. Institution of Electrical Engineers, U.K., 1998

- This is one of the better references in the library
- Beware: The $\mathcal{D}$-algorithm examples contain small errors

CAD survey reference

Melvin A. Breuer, Majid Sarrafzadeh, and Fabio Somenzi. Fundamental CAD algorithms. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 18(12):1449–1475, December 2000

- Recommended by a student
- Introduces a number of important logic-level and physical-level CAD algorithms
- A useful supplement to the material in Hachtel and Somenzi's, and Sherwani's books
- Introduces the $\mathcal{D}$-algorithm

Outline

1. Combinational testing
2. Sequential testing

Section outline

$\mathcal{D}$-algorithm review

- Given a fault, the $\mathcal{D}$-algorithm will generate a test for that fault
- Selects a $\mathcal{D}$-cube of failure to excite the fault
- Propagates a $\mathcal{D}$ value to the circuit outputs be selection $\mathcal{D}$-cubes of propagation for gates
- Justifies gate inputs
- Uses full backtracking on $\mathcal{D}$-cubes of failure and gate justification

Fault simulator

- Fault simulation is the process of determining which faults a test detects
- Test generation is complicated and time-consuming
- Fault simulation is quick

Test sequence generation

The $\mathcal{D}$-algorithm gives us a way to generate a test for a given fault
However, we need a sequence of tests for all (or most) faults

```
Determine all stuck-at faults for the circuit
Eliminate equivalent faults
while untested faults remain do
    Pick a fault and use an ATPG to generate a test
    Append the test to a list
    Use a fault simulator to determine all faults the test detects
    Eliminate detected faults
end while
```

Test sequence generation

- However, this approach can still be too expensive
- Recall
  - Fault simulation is fast
  - ATPG is slow
- Take advantage of fault simulator

Test sequence generation

```
while fault coverage is less than 90% do
    Generate a random pattern – extremely fast
    Use a fault simulator to determine if new faults are detected
    if so then
        Append pattern to a list
        Eliminate detected faults
    end if
end while
while fault coverage is less than 99.5% do
    Pick a fault and use an ATPG to generate a test
    Append the test to a list
    Use a fault simulator to determine all faults the test detects
    Eliminate detected faults
end while
```

Section outline

Redundancies

- Sometimes a s-at-0 fault can't be detected
  - Implication: Given any set of inputs to the circuit, that node can be set to 0 and the output will match the specifications
- Converse for s-a-1 faults
- Even if a portion of the circuit is not able the switch, the same function is realized
- The presence of an undetectable fault implies redundancy

Redundancies

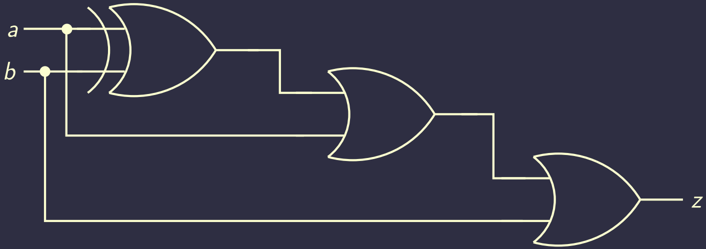
- Sometimes a s-at-0 fault can't be detected
  - Implication: Given any set of inputs to the circuit, that node can be set to 0 and the output will match the specifications
- Converse for s-a-1 faults
- Even if a portion of the circuit is not able the switch, the same function is realized
- The presence of an undetectable fault implies redundancy
- How can the $\mathcal{D}$-algorithm can be used to simplify logic?

Redundancies

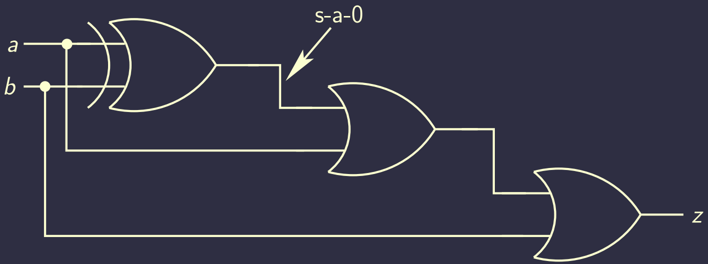
Logic terminating in the location of the stuck-at fault can be removed from the circuit and the same function will be realized

- Conventional wisdom suggests removal
  - Simplify logic
  - Make circuit more fully testable

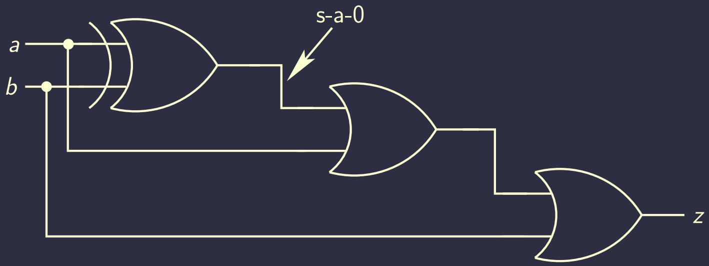
Redundancy example



Redundancy example

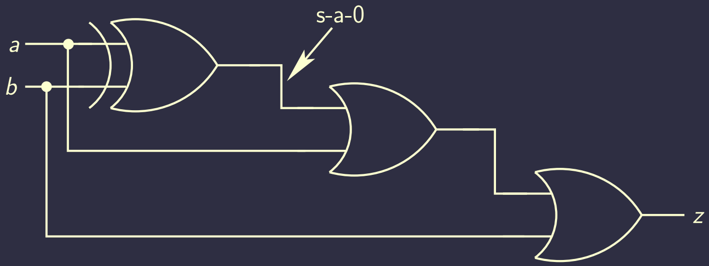


Redundancy example



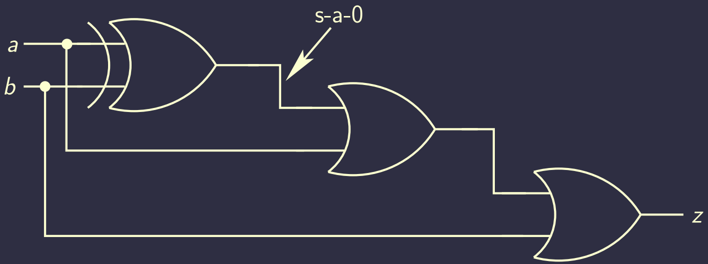
Excite: $a \neq b$

Redundancy example



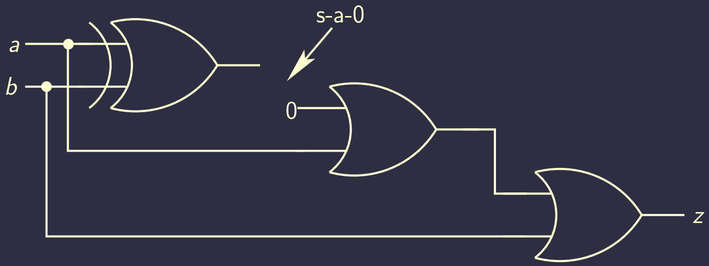
Excite: $a \neq b$
Sensitize: $a = b = 0$

Redundancy example



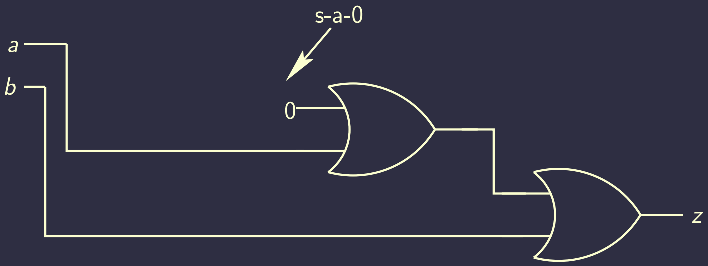
Excite: $a \neq b$
Sensitize: $a = b = 0$
Can't excite and propegate faulty value to z!

Redundancy example



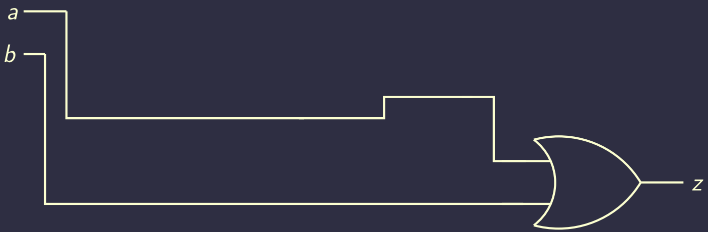
Excite: $a \neq b$
Sensitize: $a = b = 0$
Can't excite and propegate faulty value to z!
Replace s-a-0 logic with 0

Redundancy example



Excite: $a \neq b$
Sensitize: $a = b = 0$
Can't excite and propegate faulty value to z!
Replace s-a-0 logic with 0
Eliminate unused logic

Redundancy example



Excite: $a \neq b$

Sensitize: $a = b = 0$

Can't excite and propegate faulty value to z!

Replace s-a-0 logic with 0

Eliminate unused logic

Simplify logic

ATPG for logic simplification

Can use ATPG algorithm to simplify logic

1. Find a fault that is untestable due to redundancy
2. Remove the redundant logic
3. Repeat

Redundancy removal not always safe

- Logical equivalence may not imply true equivalence
- What happens if redundancy is intentional?
  - To reduce power consumption
  - To ensure signal stability for use in asynchronous circuits
- Removal can
  - Increase power
  - Introduce races

Section outline

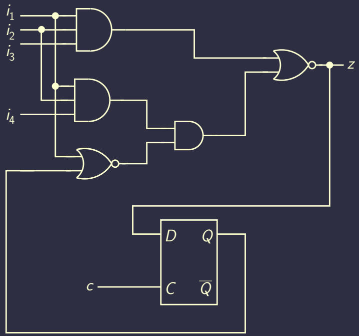
Sequential testing

- Can convert to a version of combinational testing
  - Iterative array expansion
- Simulation-based

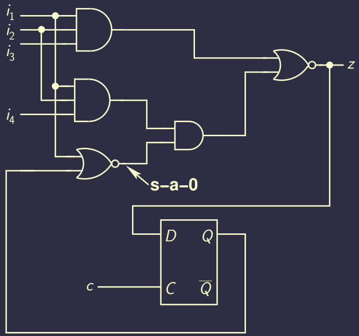
Iterative array expansion

- Make a copy of the combinational logic for each time step
- However, now the problem converts ATPG for multiple faults
- Iterative array expansion increases the size of the combinational problem
- More importantly, changes the required fault model
    - Requires ATPG for multiple faults

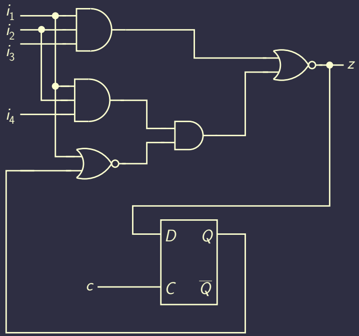
Iterative array expansion

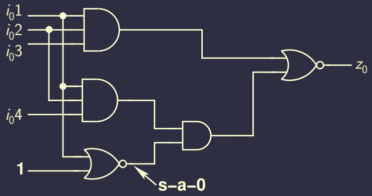


Iterative array expansion

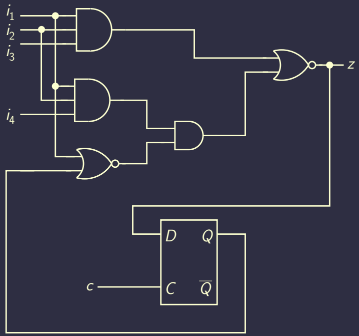


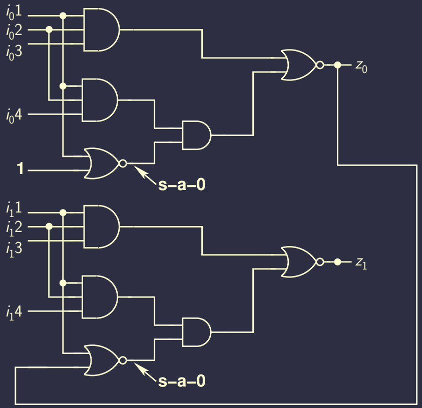
Iterative array expansion



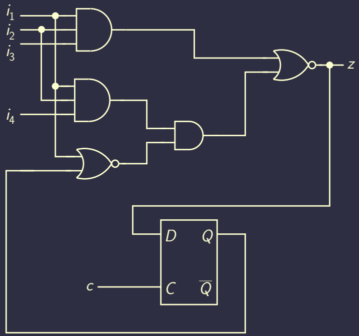


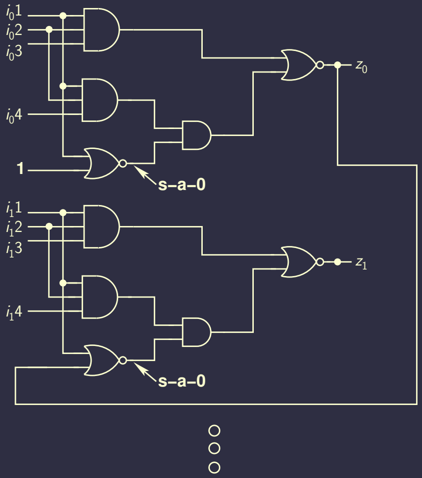
Iterative array expansion





Iterative array expansion





Modified $\mathcal{D}$-algorithm for iterative array

- Propagate $\mathcal{D}$-frontier forward
  - Generate new time frames as necessary
- Propagate gate justification backward
  - Generate new time frames as necessary
- However, this algorithm is not certain to find a test if one exists
  - Roth's algebra not suited to sequential circuits

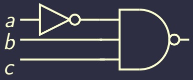
Muth's 9-valued logic

- $\mathcal{D}$-algorithm can't adequately model the possible states in sequential circuits
  - Repeated effect of fault can't be modeled
- Roth's values
  - 0/0, 0/1 ($\overline{\mathcal{D}}$), 1/0 ($\mathcal{D}$), 1/1, X/X
- Muth's values
  - 0/0, 0/1, 0/X, 1/0, 1/1, 1/X, X/0, X/1, X/X

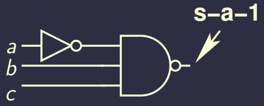
Simulation-based sequential testing

- Generate a vector
- Determine whether that vector brought the circuit state nearer to or farther from fault excitation and sensitization
- If the state moved near to detection, append the pattern to a list
- Many of the probabilistic optimization algorithms in the third lecture can be applied for simulation-based testing

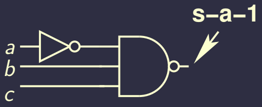
Simulation-based sequential testing



Simulation-based sequential testing

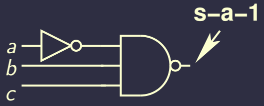


Simulation-based sequential testing

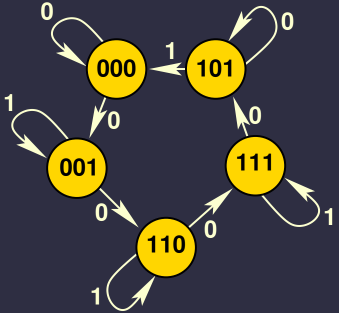


What if *a*, *b*, and *c* are state variables?

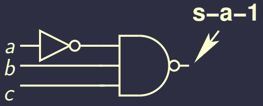
Simulation-based sequential testing



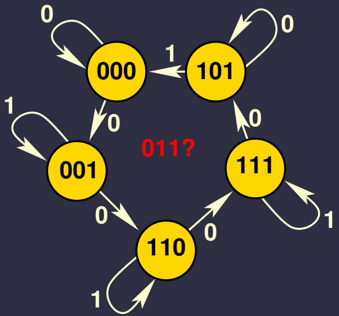
What if *a*, *b*, and *c* are state variables?



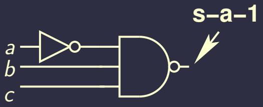
Simulation-based sequential testing



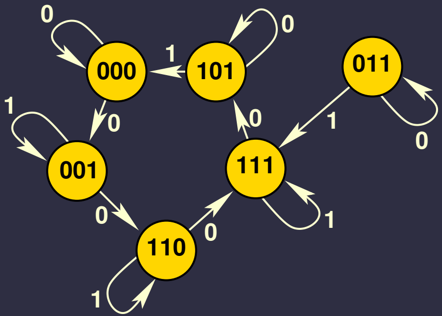
What if *a*, *b*, and *c* are state variables?



Simulation-based sequential testing



What if *a*, *b*, and *c* are state variables?



Sequential test generation problems

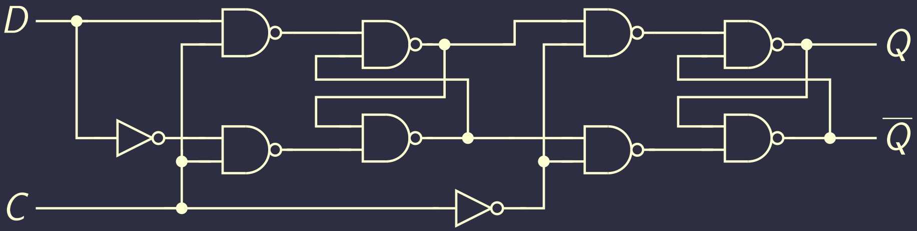
- Sequential test generation intractable for large circuits
- Some continue to work on improving test generation
- Others modify circuit structure to make test generation easier. . .

Section outline

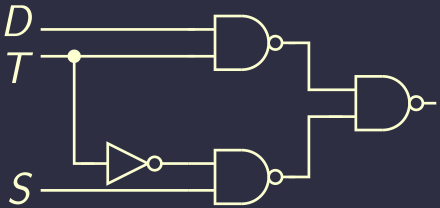
Scan-based design for test (DFT)

- If testability is considered during design or synthesis, can be made easy
- Instead of forcing ATPG to deal with synchronous testing, insert a scan chain
- Chain together (all) flip-flops and scan through an input if and only if a testing input is activated

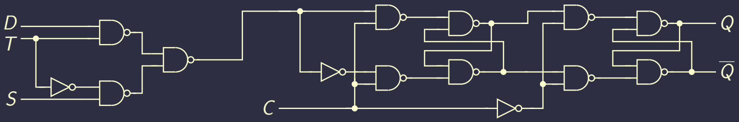
Scan D flip-flop



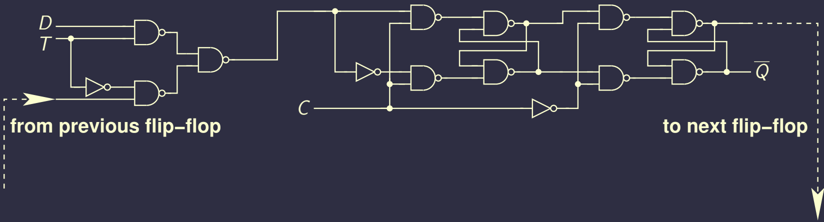
Scan D flip-flop



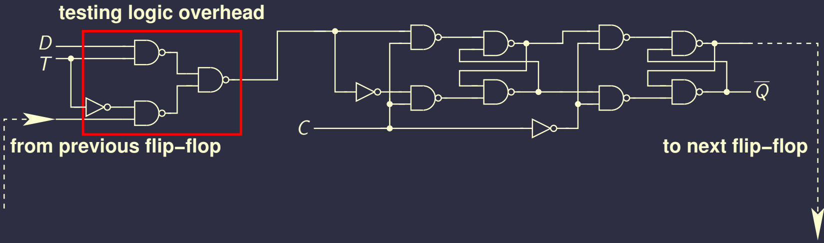
Scan D flip-flop



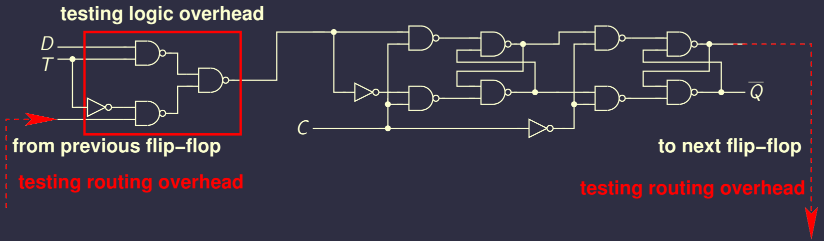
Scan D flip-flop



Scan D flip-flop



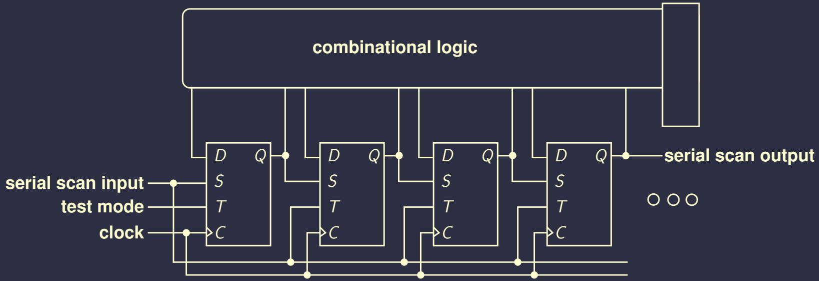
Scan D flip-flop



Full scan

- Greatly simplifies testing
  - Converts sequential testing to combinational testing
- Testing can be slow if I/O pins limited
- Significantly increases chip area and price
  - More complicated flip-flops
  - More complicated routing
  - I/O requirements, especially if speed required

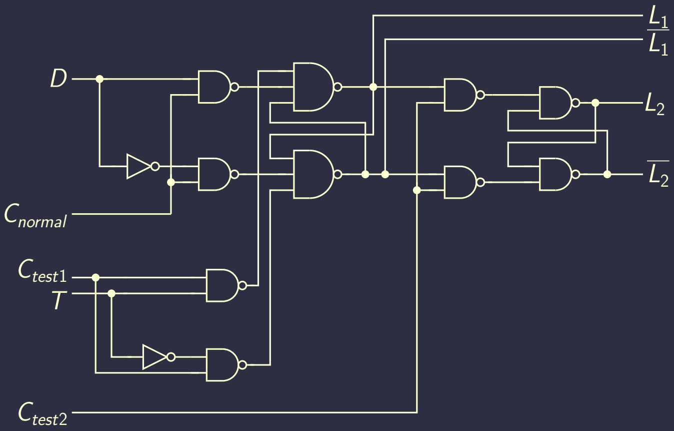
Scan chain


combinational logic
D
Q
S
T
C
serial scan input
test mode
clock
serial scan output

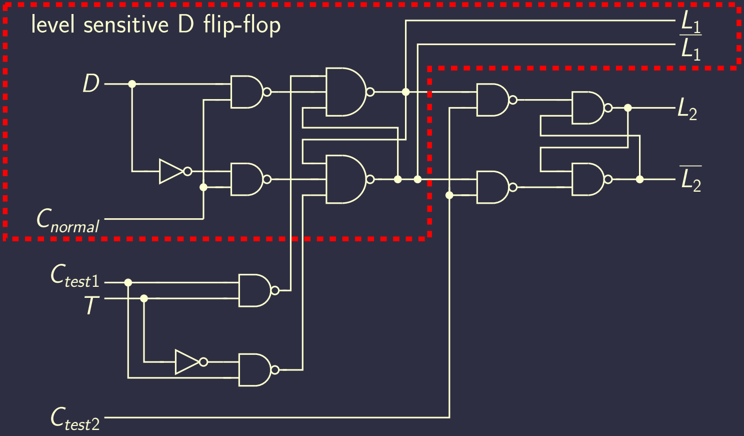
Level-sensitive scan design (LSSD)

- Less delay than scan-path during normal operation
- Used commercially, especially within IBM
- Pre-processing step replaces all latches with LSSD latches
- Other companies have related testing approaches
  - Scan path – NEC
  - Scan/set – Sperry Univac
  - Random access scan – Fujitsu

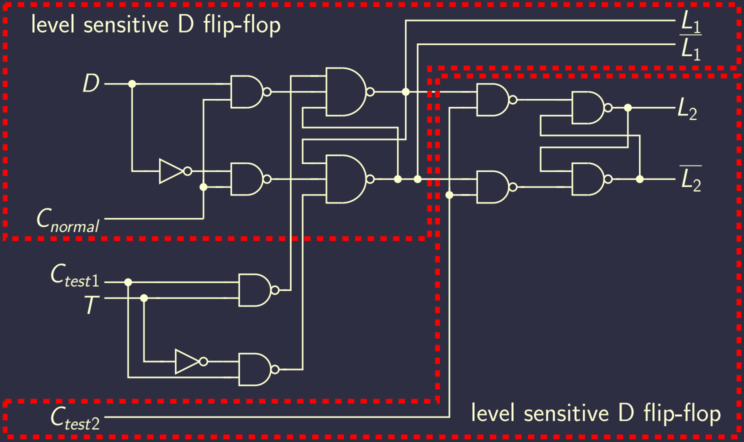
LSSD



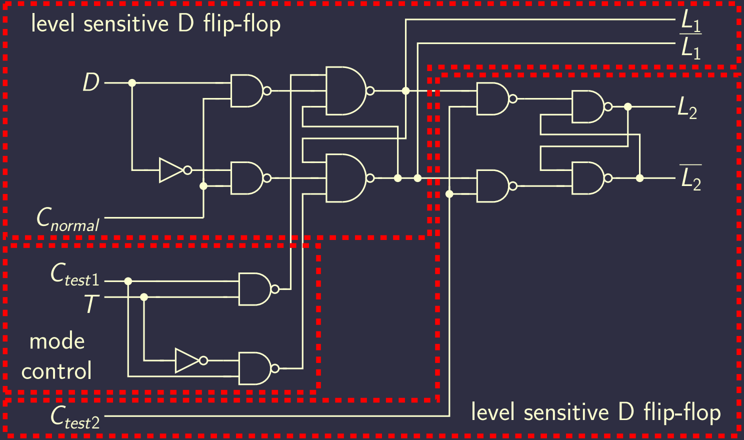
LSSD


level sensitive D flip-flop
D
C_{normal}
C_{test1}
T
C_{test2}
L_1
$\overline{L_1}$
L_2
$\overline{L_2}$

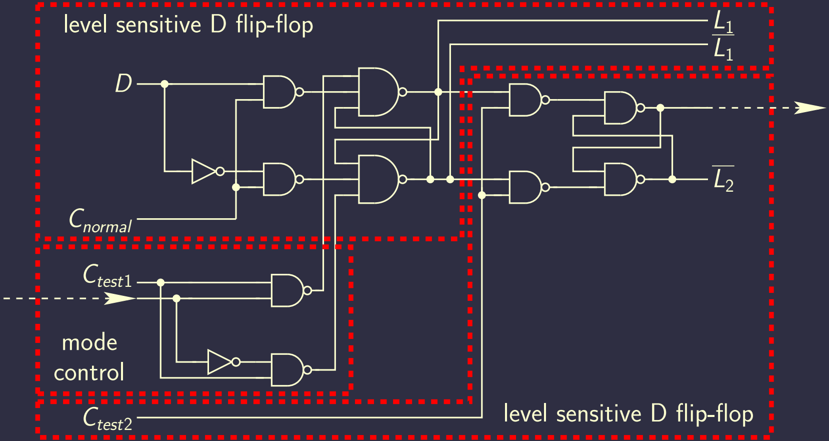
LSSD


level sensitive D flip-flop
D
C_{normal}
C_{test1}
T
C_{test2}
L_1
$\overline{L_1}$
L_2
$\overline{L_2}$
level sensitive D flip-flop

LSSD


level sensitive D flip-flop
D
C_{normal}
C_{test1}
T
mode control
C_{test2}
L_1
$\overline{L_1}$
L_2
$\overline{L_2}$
level sensitive D flip-flop

LSSD


level sensitive D flip-flop
D
C_{normal}
L_1
$\overline{L_1}$
$\overline{L_2}$
C_{test1}
mode control
C_{test2}
level sensitive D flip-flop

Full scan disadvantages

- Increased area
  - 5%–15%
- Increased delay
  - Depends on critical path average combinational logic depth
- Slow when implemented serially
  - Need to serially clock through every register in IC
- High pin requirements to speed up

Partial scan

- Instead of scanning all latches, scan a subset
- Need to carefully select scanned latches based on
  - Test coverage effects
  - Testing speed

Advanced DFT/SFT techniques

- Testing core-based systems-on-chip (SoC)
  - Related to board-level boundary-scan
- Fault injection
  - Reliability evaluation
  - Insert fault into system and determine reaction
- RTL synthesis for testability
- Software fault modeling and testing
  - Extremely difficult problem

Personal observations on testing

- Despite continued research on advanced testing techniques, industry continues to use full-scan
  - In order for industry to accept a more complicated testing technique, the advantages must be tremendous
  - Companies don't like complexity
- Testing is a fairly mature field
- Some people in academia have shifted their interest away from testing
  - It remains a huge practical problem for industry

Personal observations on testing

- Companies don't want increased complexity
- Researchers want to see their ideas used
- Therefore, reduction in interest in testing among researchers
- Therefore, reduction in interest in testing among companies
- However, companies still have huge testing problems
- Therefore, high demand for MS and Ph.D. graduates with testing experience

One option: target a research problems involving testing and another area, e.g., synthesis

Testing summary

- Low defect rate requires high test coverage
- Functional testing requires too many patterns
- Use structural testing based on a fault model, e.g., single stuck-at
- Can use fault equivalence to reduce patterns
- Can automatically generate a test for a specific fault
- Combine ATPG with fault simulator to generate a set of tests with high coverage

Testing summary

- Can use ATPG for logic simplification – redundancy
- Sequential testing difficult for large circuits
- Use DFT to make testing easier
- Use design automation to make DFT easier

Sequential testing references

- Sujit Dey, Anand Raghunathan, and Rabindra K. Roy. Considering testability during high-level design. In *Proc. Asia & South Pacific Design Automation Conf.*, February 1998
  - Survey on high-level design/synthesis for test techniques
- Kwang-Ting Cheng. Gate-level test generation for sequential circuits. *ACM Trans. Design Automation Electronic Systems*, 1(4):405–442, October 1996
  - Survey on sequential test generation