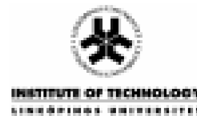
Test Generation

Abdil Rashid Mohamed, ESLAB
Sumant Sathe, ISY
Linköping University, Sweden



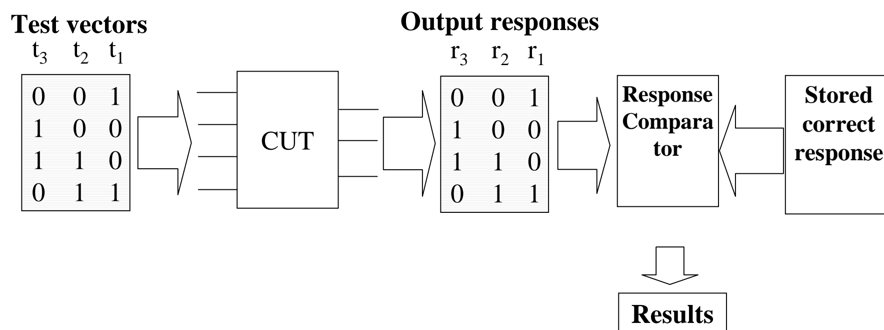
Outline

- General introduction on VLSI testing
- Description of test generation problem
- Test generation for combinational circuits
  - Selected algorithm PODEM
- Test generation for sequential circuits
  - Selected heuristic: Genetic algorithm
- Conclusions

What is VLSI Testing?

- Testing - A process to ensure that the physical device, manufactured from the synthesized VLSI design, has no manufacturing defect.
  - Verifies correctness of manufacturing process
- Two-steps involved in VLSI testing:
  - Test generation:
    - software process executed once during design
  - Test application:
    - electrical tests applied to hardware
    - test application is done on each hardware device.

Testing Process



Cost of Testing

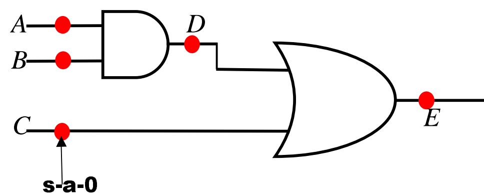
- *Design for testability* (DfT)
  - Chip area overhead and yield reduction
  - Performance overhead
- Software processes of test
  - **Test generation and fault simulation**
  - Test programming and debugging
- Manufacturing test
  - *Automatic test equipment* (ATE) capital cost
  - Test center operational cost
- Testing cost much $: ≈ 30-40% of total design cost

Fault Models

- Fault model: characterizes target faults to be tested
- Enables analysis of designs with respect to testability
- Common fault models:
  - **Single stuck-at fault model** (s-a-0, s-a-1), multiple stuck-at
  - Functional faults (used in testing processors)
  - Delay fault model: similar to stuck-at, but with timing info
    - Gate (transitions) delay fault:
      - *slow-to-rise* or *slow-to-fall* transitions or
      - interconnect signal with longer than normal propagation delay
    - path delay faults: accumulation of gate delay faults over the whole path
  - …

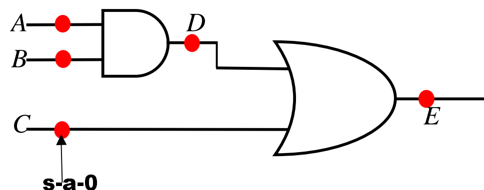
Single Stuck-at Fault Model

- Characteristics
  - Only one line is faulty at a time
  - The line is permanently stuck at 0 or 1
  - Considers faults only at the inputs and outputs of gates
- Example: 5-fault sites, 10 faults (*s-a-0, s-a-1*)



How to Generate a Test?

- C: s-a-0: to test force it to 1
- For C=1→ to arrive at E, set D=0
- D= 0 → A=0 or B=0
  →A=0,B=X or A=X,B=0
- Test vector: ABC → 0X1 or X01
- Good test result E=1, bad test result E=0



Test Generation Problem

- Input: CUT, usually a gate level netlist
- Objective & output: obtain a set of test vectors that will detect defects on a VLSI circuit.
- Requirements:
  - High fault coverage – detect as many faults as possible
  - The number of test vectors shall be as small as possible
- Potential defects are modeled using a fault model
  - Single stuck at fault model (most widely used)
  - A given fault model, say ***single stuck at*** models only a subset of real defects

Cost of Test Generation

- Cost of test generation depends on:
  - Complexity of the fault model
  - Complexity of the TG algorithm
  - Complexity of the DUT

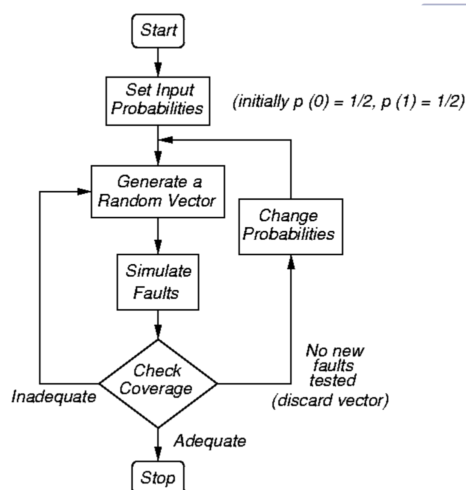
Test Generation: Classification

Type of Circuit
- Test pattern generation for combinational circuits
- Test pattern generation for Sequential circuits

Abstraction level
- Gate level test generation
- RT level test generation
- Hierarchical test generation

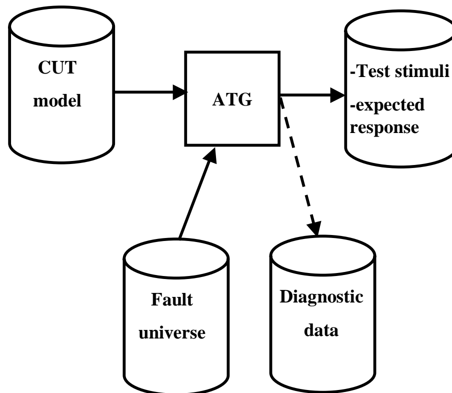
Generation method
- Random test generation
- Pseudo-random test generation
- Exhaustive test generation
- Deterministic test generation

Random Pattern Generation (RTG)



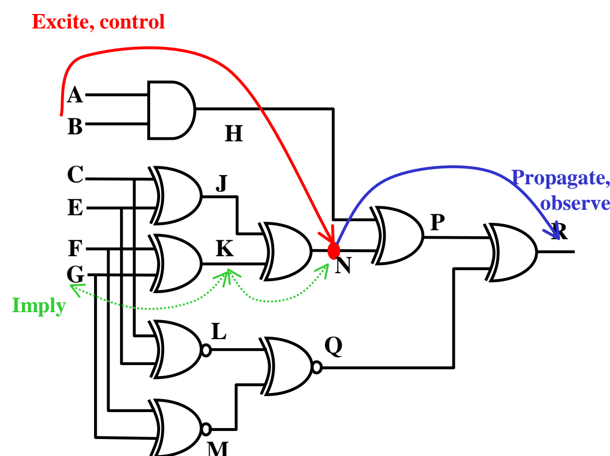
- Doesn't consider CUT structure
- Generate tests randomly
- RTG method is cheap
- Disadvantages:
  - too many test vectors
  - low quality tests (low FC)

Deterministic Test Pattern Generation (DTG)



- Considers CUT structure to generate tests
- DTG method is expensive
- Advantages:
  - fewer test vectors cmp. RTG
  - high quality tests (high FC)
- Can be ***Fault oriented*** or fault independent

Components of TG Process



- A fault ***n s-a-v*** *(e.g. s-a-1)*
- Activate (excite) a fault
  - Set PI values that causes line *n* to be set to a value *~v*
  - *controllability*
- Propagate the resulting error to a PO
  - *observability*

Roth's 5-valued Algebra

Symbol	Meaning	Good machine	Failing machine
D	1/0	1	0
$\overline{D}$	0/1	0	1
0	0/0	0	0
1	1/1	1	1
X	X/X	X	X

- Needed for simultaneous analysis of: good & faulty circuit

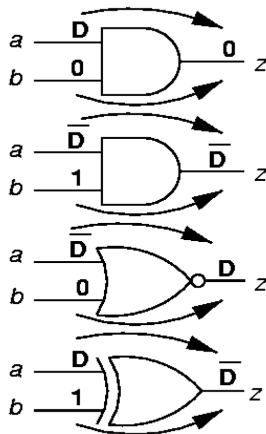
D-frontier

- Set of all gates whose output value is currently *x* but have one or more error signals D or $\overline{D}$ on their inputs
- Error propagation –
  - select a gate from D-frontier
  - Assign values to unspecified inputs to make output D or $\overline{D}$
- An empty D-frontier implies no error propagation is possible
  - Backtracking is needed

X-path

- Path whose lines have value *X*.
- *Fact:* if a gate G is on D-frontier, the errors at its inputs can propagate to PO Z only if there is at least one x-path between G and Z.
- X-path identification –
  - Avoids decisions that are bound to failure
  - Avoids unnecessary backtracking
  - Its a look ahead strategy of PODEM to prune decision tree

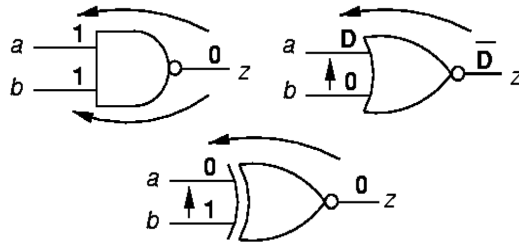
Forward Implication



- Results in logic gate inputs that are significantly labeled so that output is uniquely determined

Backward Implication

- Uniquely determine all gate inputs when the gate output and some of the inputs are given



General Outline of a TG Algorithm

SolveTG ()

Begin

If *Imply_and_check()=FAILURE* **then return** *FAILURE;*
If (error at PO) **and** all lines are justified
then return SUCCESS;
If (no error can be propagated to a PO)
then return FAILURE;
Select an unsolved problem; // *justification or propagation*
Repeat
Select one untried way to solve it; *//line justification, error propagation*
If *SolveTG() = SUCCESS* **then return** *SUCCESS;*
Until all ways to solve it have been tried;
Return FAILURE;

End

Combinational Circuit TG- Heuristics

- Structural search methods: Perform search on Boolean space guided by the circuit topology
  - D-algorithm (Roth)
  - PODEM (Goel)
  - FAN (Fujiwara, Shimono)
  - Socrates (Schultz et al.)
- SAT based methods
  - Larrabee
  - Stephan

Combinational Circuit TG - Heuristics

- Symbolic and Algebraic methods: Elegant abstract formulation (in practice is infeasible)
  - ENF (Armstrong)
  - Boolean differences
  - Pooge's method

PODEM: Path Oriented Decision Making: Facts

- Start search for a test pattern at PI
- Start: select an objective *(l,v)* – a specific fault to be detected
- Create a search tree in which two choices are available for PIs
- Choice is random
- Evaluate implications of the choice on subsequent gates to the output
- If it controls the fault site to intended value (achieved objective) accept it and select another PI
- If inconsistency occurs, backtrack and select another input combination
- Stop the search when a pattern is generated and no patterns are possible (undetectable fault)

PODEM

Objective

- A value to be justified at a line ***l*** is an objective ***(l;v)*** to be achieved through *PI assignment*
  - Target fault is ***l s-a-v***
  - Assign values to PIs only
  - Implicitly examine all possible PIs
  - Terminate when a test is found

PODEM: Objective Selection

if (gate *g* is unassigned) return (*g*, *v*);
select a gate *P* from the D-frontier;
select an unassigned input *l* of *P*;
if (gate *P* has controlling value)
    ***c* = controlling input value of *P*;**
else if (0 value easier to get at input of XOR/EQUIV gate)
    ***c* = 1;**
else *c* = 0;
return (*l*, *c*);

PODEM

Backtracing:

- Maps a desired objective into a PI assignment that has a chance to contribute to achieving the objective
  - No values are assigned during backtracing.
  - Values are assigned only by simulating PIs assignments
    - only through forward implications of PI assignments.

PODEM: Backtrace(s, v_s)

```
v = v_s;
while (s is a gate output)
    if (s is NAND or INVERTER or NOR) v = v;
    if (objective requires setting all inputs)
        select unassigned input a of s with hardest
          controllability to value v;
    else
        select unassigned input a of s with easiest
          controllability to value v;
    s = a;
return (s, v)
/* Gate and value to be assigned – s is a PI */;
```

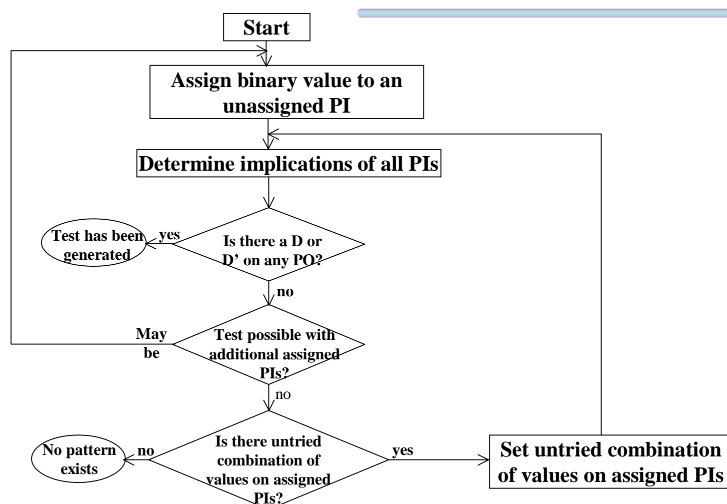

PODEM

- 1. Initialization: Assign x (don't cares) to all PIs.
- 2. Assign a 0 or a 1 to a PI (for a given objective)
- 3. Determines whether the current input combination constitutes a test. Stop if a test is found
- 4. (Backtracing):If it is possible to generate a test with additional assigned PIs, go to 2.
- 5. (Backtracing):
  - if there is an input pattern which has not been examined as a possible test, go to 3,
  - else the fault is redundant (undetectable)

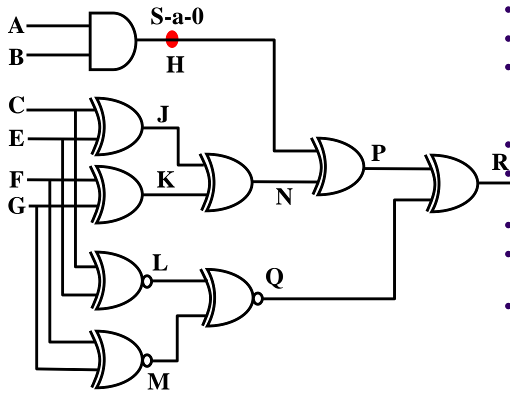
PODEM: Algorithm

```
while (no fault effect at POs)
    if (xpathcheck (D-frontier))
        (l, v_l) = Objective (fault, v_fault);
        (pi, v_pi) = Backtrace (l, v_l);
        Imply (pi, v_pi);              // Imply(pi, v_pi)
        if (PODEM (fault, v_fault) == SUCCESS) return (SUCCESS);
        (pi, v_pi) = Backtrack ();     // reverse decision
        Imply (pi, v_pi);              // Imply(pi, ~v_pi)
        if (PODEM (fault, v_fault) ==  SUCCESS) return (SUCCESS);
        Imply (pi, "X");               // Imply(pi, x)
        return (FAILURE);
    else if (implication stack exhausted)
        return (FAILURE);              // test not possible
    else Backtrack ();
return (SUCCESS);
```


PODEM: Simplified

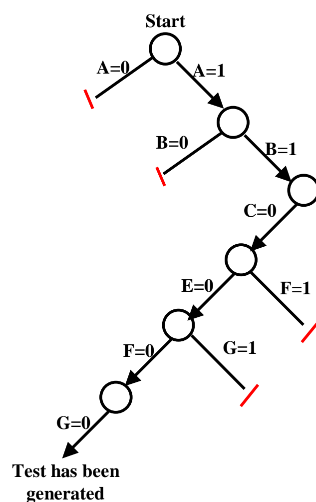


PODEM: Example



- Assign X to all inputs
- Set objective: (H,1)
- A=0 → H=0. Bad choice doesn't further objective, discard it.
  - Choose A = 1
- 3. In a simular way: rejects B= 0, choose B=1
- 4. C=0. Its implication neither further nor block the objective
- 5. E=0 → J=0, L=1
- 6. Similarly select F=0, G=0 which implies that K=0, M=1, N=0, Q=1
- 7. D can be propagated on P, then D' can be propagated on the PO R.
  - A test is generated: ABCDEFG = 1100000

PODEM: Search Tree



- 1. Assign X to all inputs
- 2. A=0 → H=0. Bad choice doesn't further objective, discard it.
  - Choose A = 1
- 3. In a simular way: rejects B= 0, choose B=1
- 4. C=0. Its implication neither further nor block the objective
- 5. E=0 → J=0, L=1
- 6. Similarly select F=0, G=0 which implies that K=0, M=1, N=0, Q=1
- 7. D can be propagated on P, then D' can be propagated on the PO R.
  - A test is generated: ABCDEFG = 1100000

PODEM: Advantages

- No consistency check, since conflicts never occur
- No J-frontier, since no values that require justification
- No backward implication, since values are propagated only forward.
- Backtracking done by simulation
- Faster than D-algorithm

TG Complexity

- Exponential in worst case
- In practice to limit computation effort set limit on
  - CPU time spent
  - Number of incorrect decisions made

Sequential Circuit Test Generation

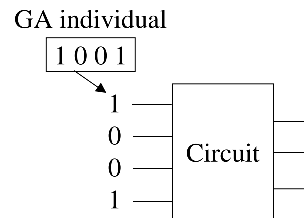
- Sequential circuit:
  - combinational part + memory
- A test in sequential circuit is a vector which:
  - Initializes the circuit to a known state
  - Activates the fault, and
  - Propagates the fault effect to a PO
- Sequential circuit test generation methods
  - Time frame expansion methods
  - Simulation based methods
  - Genetic algorithm

Genetic Algorithm: Basic idea

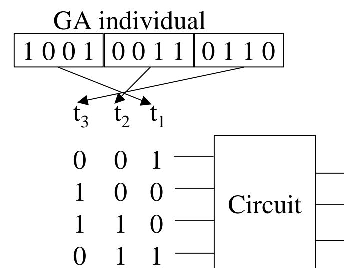
- **Population** solution space, set of possible tests
- **Individual** one possible test. Coded as binary string
- **Fitness** determines suitability of the solution.
  - user defined and problem specific
  - does it lead to a good test vector?
- **Selection** how to select individuals that can produce better ones?
- **Cross-over** how to combine portions of individuals to create new offsprings
- **Mutation** an incremental change made to each individual
  - done with low probability
  - introduce new features

Alphabet Encoding

- **Test vector generation**:
  - Map each gene of an *individual* string to a PI bit
  - Thus, the string represents a test vector
  - Evaluate fitness of each string by fault simulation



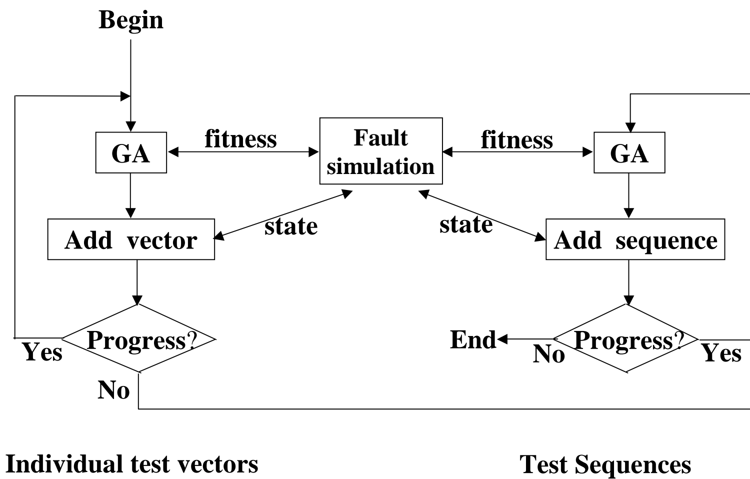
- **Test sequence generation**:
  - Individual represents a sequence of vectors to be applied to the PIs in successive clock cycles (t_1, t_2, t_3)
  - Place individual vectors in a sequence in adjacent positions on a single string



GA for Test Generation

Generate initial population; *// initial random set of possible strings(tests)*
Evaluate(Population); *// compute fitness function*
For i ← 1, 2, …, NumGenerations **do**
    NewPopulation := ϕ;
    for j ←1 , 2, …, PopulationSize/2 **do**
        select p_1 and p_2 from the Population;
        crossover(p_1, p_2, c_1, c_2); *// create new possible test vectors*
        mutate(c_1); *mutate*(c_2); *// improve/change quality*
        Add c_1 and c_2 to NewPopulation;
    end for
    Evaluate(NewPopulation);
    Population := NewPopulation;
    bestIndividual := bestOf(Population U bestIndividual); // *rank based on fitness*
End for
Solution := bestIndividual; *// return a test vector*

Framework of GA based Test Generation



GA

- 1. Generate individual test vectors until no more improvements in FC
- 2. If sequential circuit generate test sequences until no more improvements in FC is achieved
  - GA with random initial population generates vectors
  - Sequential circuit fault simulator evaluates fitness of each candidate test vector
  - Best test in any generation is added to the test set
  - Fault simulator is then used to to update the state of the circuit and to drop detected faults
  - Generation of individual test vectors is repeated until a given number of vectors are successively generated that do not

GA Parameter Selection

- To provide good combinations of genes (characters)
  - Large population size - adequate diversity
  - All characters in the alphabet should be present at every string position
  - Set a reasonable limit on population to limit computation
- Large number of generations - allow useful combination of characters from several strings
  - Limit to 8 generations → to reduce run time
  - For combinational circuits →20 generations to expand search space
- Cross-over probabilities – cross 2 individuals to get 2 new ones
- Mutation probabilities
  - Avoids loss of key characters at various string positions.
  - Can also destroy good combinations of characters

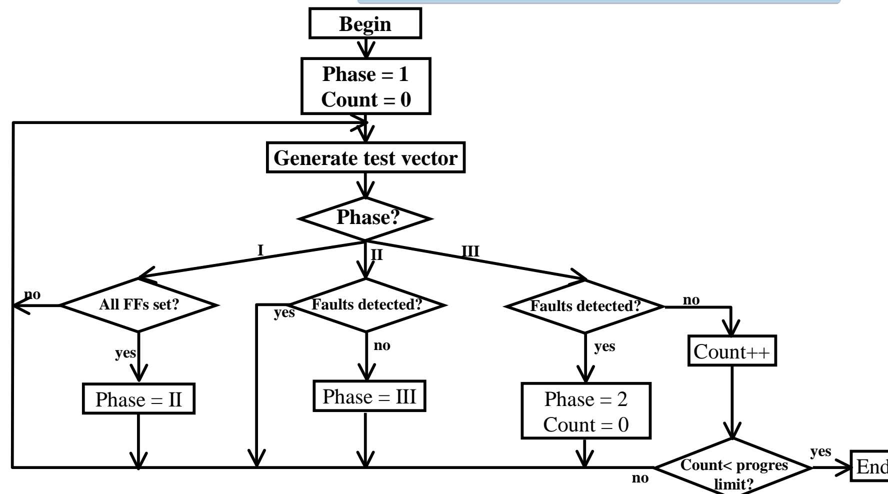
Suitable Paramaters for GA Test Generation

- **Individual test vector generation**:

Vector length (L)	Population size	Mutation probability
< 4	8	1/8
4 –16	16	1/16
17 – 49	16	1/L
50 – 63	24	1/L
64 – 99	24	1/64
> 99	32	1/64

- **Test sequence generation**: for all circuits
  - Population size = 32
  - Mutation probability = 1/64

Generating Individual test Vectors



Fitness Function

- **Phase I**: test vectors to initialize FFs
  - Fitnes is a measure of the no. Of FFs set to a known state (0,1) and fraction of FFs changing values since the previous time frame

    fitness = # total flip flops set + fraction of flip flops changed
  - Good circuit simulation is needed to get FF state information
- **Phase II**: test vectors to maximize the no. of faults detected
  - Fitness of a candidate test is the no. of faults it detects
  - In fitness function, included also is the no. of fault effects propagated to FFs

$$fitness = \#faults\ detected + \frac{\#faults\ propagated\ to\ FFs}{(\#faults\ simulated)(\#flip\ flops)}$$

Fitness Function

- **Phase III**: count the no. of non-contributing test vectors
  - good and faulty circuit activity levels encourage evolution of useful vectors
  - vectors that activate more faults and propagate more fault effects will have higher fitness values
  - GA is likely to evolve a vector that can propagate the effects of some faults to PO

$$fitness = \#faults\ detected + \frac{\#faults\ propagated\ to\ FFs}{(\#faults)\ (\#flip\ flops)} + \frac{2\ (\#good\ \&\ faulty\ circuit\ events)}{(\#circuit\ nodes)(\#faults)}$$

- **Phase IV:** Test sequence geneartion

$$fitness = \#faults\ detected + \frac{\#faults\ propagated\ to\ FFs}{(\#faults)(\#flip\ flops)(sequence\ length)}$$

Run Time

- String manipulations (selection, cross-over, mutation)
  - not very time consuming
- Fitness function and fault simulation
  - accurate fitness function is needed ☺
  - dominates computation cost ☹
  - solution: ☺
    - approximate the fitness of a test by using a small sample of faults
    - avoid fault simulation by choosing a fitness function based on good circuit simulation
  - Disadvantages – less accurate fitness: ☹
    - results in more test vectors
    - lower fault coverage

Observations and Experimental Results

- ISCAS89 seq. circuits:
  - Selection & cross over schemes → significant impact on FC
    - tournament selection without replacement, uniform cross over → best results
  - Variations in mutation rate have smaller effect on FC
  - Non-overlapping populations give highest FC
  - Overlapping populations give 1.3 times speedups, only 0.4% drop in FC
- Problem specific knowledge → best results for all circuits
  - target hard to test faults one at a time in a 2-phase strategy → highest FC
    - Phase I: excites a fault and propagates fault effects to FFs
    - Phase II: drives the fault effects from the FFs to the POs.
  - for wide AND or OR gates
    fitness = 1 if fault is excited or 0 otherwise + fraction of AND inputs set to 1 + fraction of OR inputs set to 0

Performance Comparisons

Algorithm	Estimated speedup over D-ALG	Year
D-ALG	1	1966
PODEM	7	1981
FAN	23	1983
TOPS	292	1987
SOCRATES	1574	1988
Waicukauski et al.	2189	1990
EST	8765	1991
TRAN	3005	1993
Recursive learning	485	1995
Tafertshofer et al.	25057	1997
Genetic algorithm	?	

References

- A Book "*Digital Systems Testing and Testable designs*", by M. Abramovic, M. Breuer, A. Friedman
  - Chapter 6: Testing for Single Stuck Faults
- A Book "*Genetic Algorithms for VLSI Design, Layout & Test Automation*", by P. Mazumder, E. M. Rudnick
  - Chapter 6: Automatic Test Generation
- Lecture notes and a book "*Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI*" by M. L. Bushnell, V. D. Agrawal.
  - http://www.ece.wisc.edu/~va/BOOK/books.html
- A Book "*Principles of Testing Electronic Systems*", by Samiha Mourad, Yervant Zorian
  - Chapter 6: Automatic Test Pattern Generation

Conclusions

- Introduced the test generation problem
  - Several algorithms exist in the literature
- Illustrated with 2 algorithms
  - Classic PODEM - for combinational circuits
  - Genetic algorithm heuristic – for sequential circuits

Supplementary points

- ***J-Frontier***
  - Set of all gates whose output value is known but not implied by its input values
  - Keep track of currently unsolved line justification problem
- ***Imply_and_check()***
  - Compute all values that can be uniquely determined by implication
  - Check for consistency
  - Maintain the D-frontier and J-frontier