VLSI testing - Assignment 6

309510133 - Cheng-Cheng Lo

Part.a - backtrack number

Run

```
./atpg -bt [backtrack_number] -output [pattern] [circuit]

# Example
./atpg -bt 1 -output b17_1_vectors b17.bench
```

Results

-bt	# patterns	fault coverage (%)	CPU run time (s)	actual # backtrack
1	41647	55.00	838.08	68413
10	72511	82.09	1235.09	376982
100	83711	90.01	1520.50	1821691
1000	86025	91.62	3767.59	12664818
no limit	87963	93.00	22627.57	95087176

Discussion

When the number of backtrack increases, fault coverage increases but backtrack number increases and running time also becomes longer.

Part.b - ATPG with checkpoint theorem

Run

```
# ATPG based on fault list using checkpoint theorem
./atpg -output -check_point [pattern] [circuit]

# ATPG based on total fault list
./atpg -output [pattern] [circuit]

# run fault simulation
./atpg -input [pattern] [circuit]
```

	# patterns	fault coverage (%) - while atpg	fault coverage (%) - using fsim
original	87963	93.00	97.34%
checkpoint	53071	92.84	97.28%

Discussion

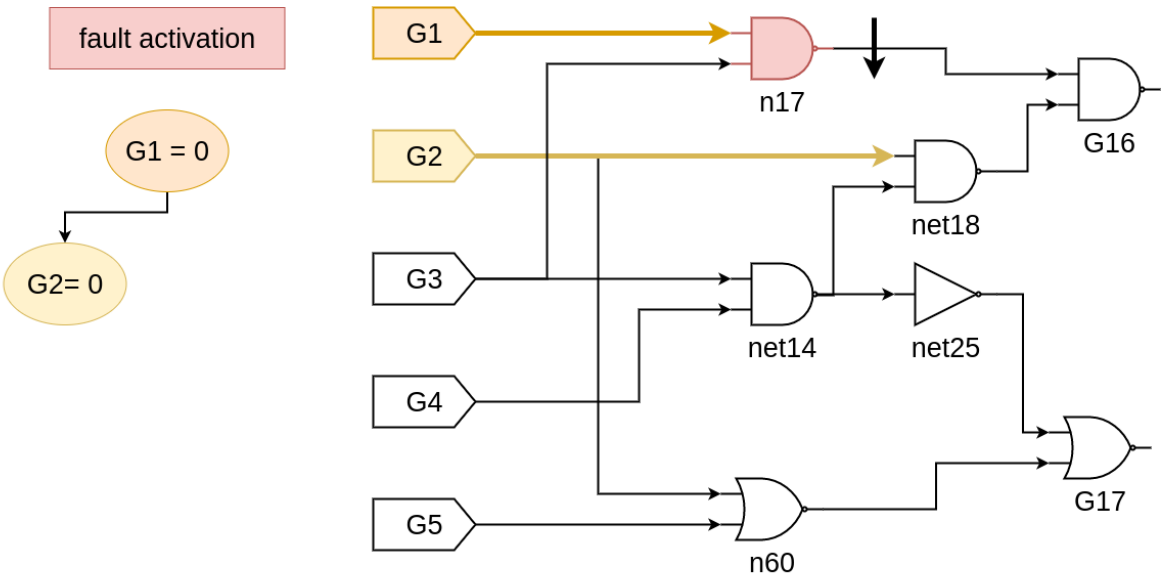
Using checkpoint theorem will get almost the same fault coverage but with less patterns.

Part.c - PODEM

net17 stuck-at-0

For net17 s-a-0, it does decisions on the following sequences:

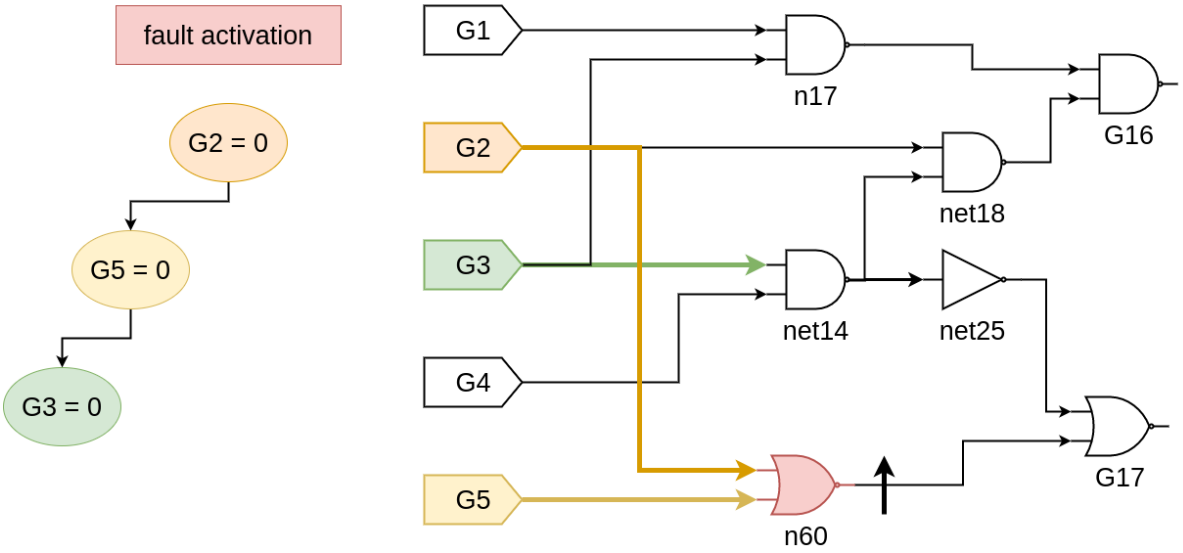
1. G1 = 0
2. G2 = 0



n60 stuck-at-1

For n60 s-a-1, it does decisions on the following sequences:

1. G2 = 0
2. G5 = 0
3. G3 = 0



Since c17.bench is a simple circuit, no conflicts found during search. The decisions are not flipped after making.

Part.d - random pattern before ATPG

```
./atpg -random_pattern -output [output_pattern] [circuit_name]

# example
./atpg -random_pattern -output c17pat c17.bench
```

The number before -> is the fault coverage after generating 1000 patterns or reach 90%; the number behind it is the fault coverage generated by ATPG.

b17.bench

- original ATPG

fault coverage (%)	# patterns	total CPU time (sec)
93.00	87963	21711.50

- random pattern generation + original ATPG

fault coverage (%)	# patterns	total CPU time (sec)
47.52 -> 93.72	1000 + 40156	9389.91

s35932_com.bench

- original ATPG

fault coverage (%)	# patterns	total CPU time (sec)
89.64	77	50.03

- random pattern generation + original ATPG

fault coverage (%)	# patterns	total CPU time (sec)
89.69 -> 89.69	1000 + 0	50.60

s38417_com.bench

- original ATPG

fault coverage (%)	# patterns	total CPU time (sec)
99.68	1373	13.91

- random pattern generation + original ATPG

fault coverage (%)	# patterns	total CPU time (sec)
89.38 -> 99.67	1000 + 1121	15.72

s38584_com.bench

- original ATPG

fault coverage (%)	# patterns	total CPU time (sec)
95.57	856	7.19

- random pattern generation + original ATPG

fault coverage (%)	# patterns	total CPU time (sec)
89.02 -> 95.57	1000 + 593	10.16

Discussuin

From above results, I found that for a large circuit, random pattern generation greatly reduces the number of patterns and time. It is necessarily true for a small one.

Part.e - ATPG for bridging faults

In this part, I tried to transform the original problem, ATPG for bridging faults to ATPG for single stuck-at faults.

Consider two neighboring gates A and B, for type-AND bridging faults, it can be viewed as A s-a-0 when B is 0 or B-s-a-0 when A is ; for type-OR, similarly, it can be viewed as A s-a-1 when B is 1 or B s-a-1 when A is 1. For each type of fault, if one of two conditions is satisfied, we say that the fault can be detected.

Run

```
./atpg -bridging_atpg -output [output_pattern] [circuit_name]

# example
./atpg -bridging_atpg -output c17pat c17.bench
```

Results

	fault coverage (%)	# patterns
c17.bench	100.00	5