ECE 586

FAULT DETECTION IN DIGITAL CIRCUITS

BONUS PROJECT REPORT

AUTOMATIC TEST PATTERN GENERATION

Harshita Ravi Shankar

A20327797

INTRODUCTION

The objective of this project is to create a software program for Automatic Test Pattern Generation (ATPG). The ISCAS89 benchmark circuits, representing synchronous sequential circuits are used as a basis for this project, which supports the DFT (Design For Testability) methodology. The software code is modeled in *"Python"* which has the easiest way of indexing multiple input and outputs.

The algorithm should provide good fault coverage, with reduced fault set obtained from checkpoints using fault equivalence and fault dominance.

AUTOMATIC TEST PATTERN GENERATION

The test pattern generation algorithm to be programmed here focuses on the combinational part of any sequential circuit. The benchmark circuit files from ISCAS89 which are used for VLSI CAD design testing are utilized here to test the algorithm. The ATPG algorithm initiates by finding all the check points in the combinational circuit. The stuck-at faults at each check point are then identified and the concept of fault equivalence and fault dominance are used to help reduce the fault set. With the reduced set of faults, an initial set of test vectors are defined and the coverage of faults are checked continuously throughout the execution of algorithm.

Once the desired fault coverage is obtained for a benchmark circuit, other benchmark circuits are tested with the same algorithm to obtain the same coverage. Hence in an iterative process, all the faults in a circuit are modeled, and an input test vector is obtained automatically, by selecting a specific fault and creating a test vector to detect that fault.

The algorithm is represented in a simpler manner in Figure 2.1 below. This algorithm initially creates a net list data structure with four tables namely, E*lement table*, *Signal table*, *FANIN and FANOUT table.* Using the index value of these tabulations, we obtain the check points and can implement the levelization algorithm.

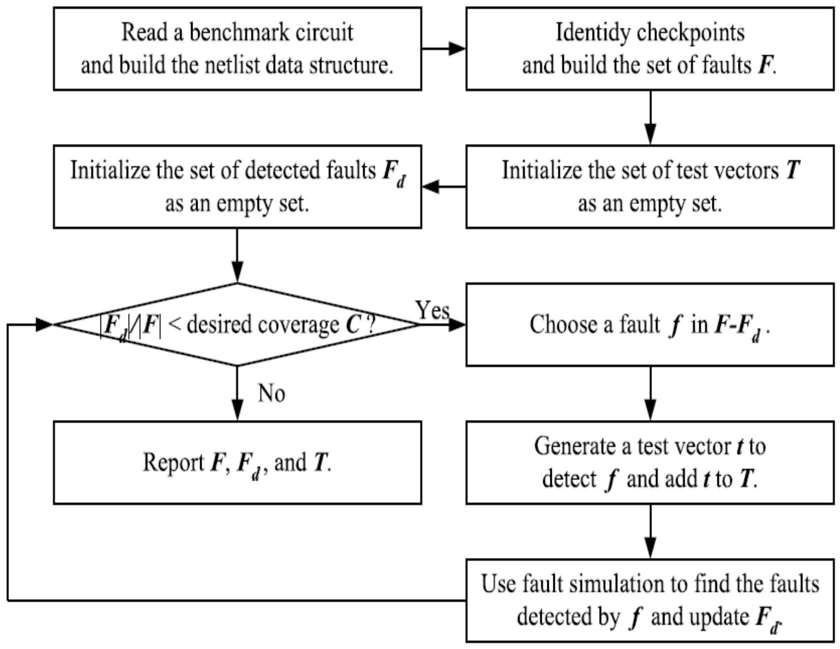

Read a benchmark circuit and build the netlist data structure.
Identidy checkpoints and build the set of faults F.
Initialize the set of test vectors T as an empty set.
Initialize the set of detected faults F_d as an empty set.
$|F_d|/|F|$ < desired coverage C ?
Yes
No
Choose a fault f in F-F_d.
Report F, F_d, and T.
Generate a test vector t to detect f and add t to T.
Use fault simulation to find the faults detected by f and update F_d.

Figure 2.1 – ATPG Algorithm.

IMPLEMENTATION OF PROJECT

This project has been implemented using Python coding language which is a powerful programming language. The high-level data structures are efficient and simple approach that implements object-oriented programming. The systematic syntax and the dynamic typing, along with the interpreted nature, makes python an ideal scripting language and has applications in most platforms.

The checkpoints in the benchmark files have to be found. The primary inputs and branching signals are together the checkpoints. The algorithm that has been used to generate the checkpoints is as given below.

Algorithm for Checkpoint Generation

1. Start.

2. Initialize all the lists required for the program.

3. Initialize the variables required for the program.

4. Open the benchmark file (Input file).

5. Split each line till end of file and append it to a masterList. Now the masterList contains all the lines of code.

6. Close the file.

7. Scan for the term ‘inputs\n’ in the master list and extract the number of inputs. Repeat the same for ‘outputs\n’.

8. Separate out the Input signals by matching the term INPUTS(G0) [The first input is G0].

9. Store each input as primary input in list 1.

10. Similarly extract all the outputs. Put it into a new output list.

11. Count the number of occurrences of the output signal from the output list.’

12. If the count of the counter is more than or equal to 3, the signal is treated as a checkpoint.

13. End.

The test vectors need to be generated in order to detect the faults. Depending on the number of inputs, all possible combinations are generated. The algorithm to generate the test vectors is shown below. Once it is done and the expressions for both the original and the faulty circuits are obtained, we give these test vectors as the primary inputs. The test vectors which propagate the stuck at faults in the faulty circuit to the output are the test vectors.

Algorithm for Test Vector Generator

1. Start.

2. Initialize all the lists required for the program.

3. Initialize the variables required for the program.

4. Open the benchmark file (Input file).

5. Split each line till end of file and append it to a masterList. Now the masterList contains all the lines of code.

6. Close the file.

7. Scan for the term ‘inputs\n’ in the master list and extract the number of inputs.

8. Depending on the number of inputs, generate the binary number for each number less than (input*input)

9. Append each binary number to a testGen List for further use in the program.

10. End.

The original circuits are the benchmark files. The faulty circuit is obtained from the original circuit. The stuck at faults are randomly put at certain checkpoints of each of the benchmark files. The input of any gate or any of the primary inputs may be pulled down to zero for an s-a-0 fault or pulled up for an s-a-1 fault at that signal. Once the original and the faulty circuits are obtained, the expression for both have to be generated.

In order to generate the expression for both the circuits, I tried an online code from github. Each of the gates are defined as separate functions and connectors are used to give inputs and obtain the outputs. Once the expressions are obtained, the outputs of the circuits have to be compared and faults have to be detected and simulated.

MiniSAT is a minimalistic, open source SAT solver which helps the programmers to get started on SAT. I tried using Mini-SAT to run the simulation logic for the input benchmark file in order to obtain the comparison between the faulty circuit and the original circuit. I installed the MiniSAT solver but was not able to configure the setup on my system. The screenshot has been provided in the appendix.

RESULTS

A few of the benchmark files were successfully parsed to test the code and the necessary information is extracted. All the checkpoints for the circuits have been identified. The test vectors have been generated. Also I have tried to obtain the expression for the original and the faulty circuits and compare them using the MiniSAT solver in order to detect and simulate the faults in the circuits.

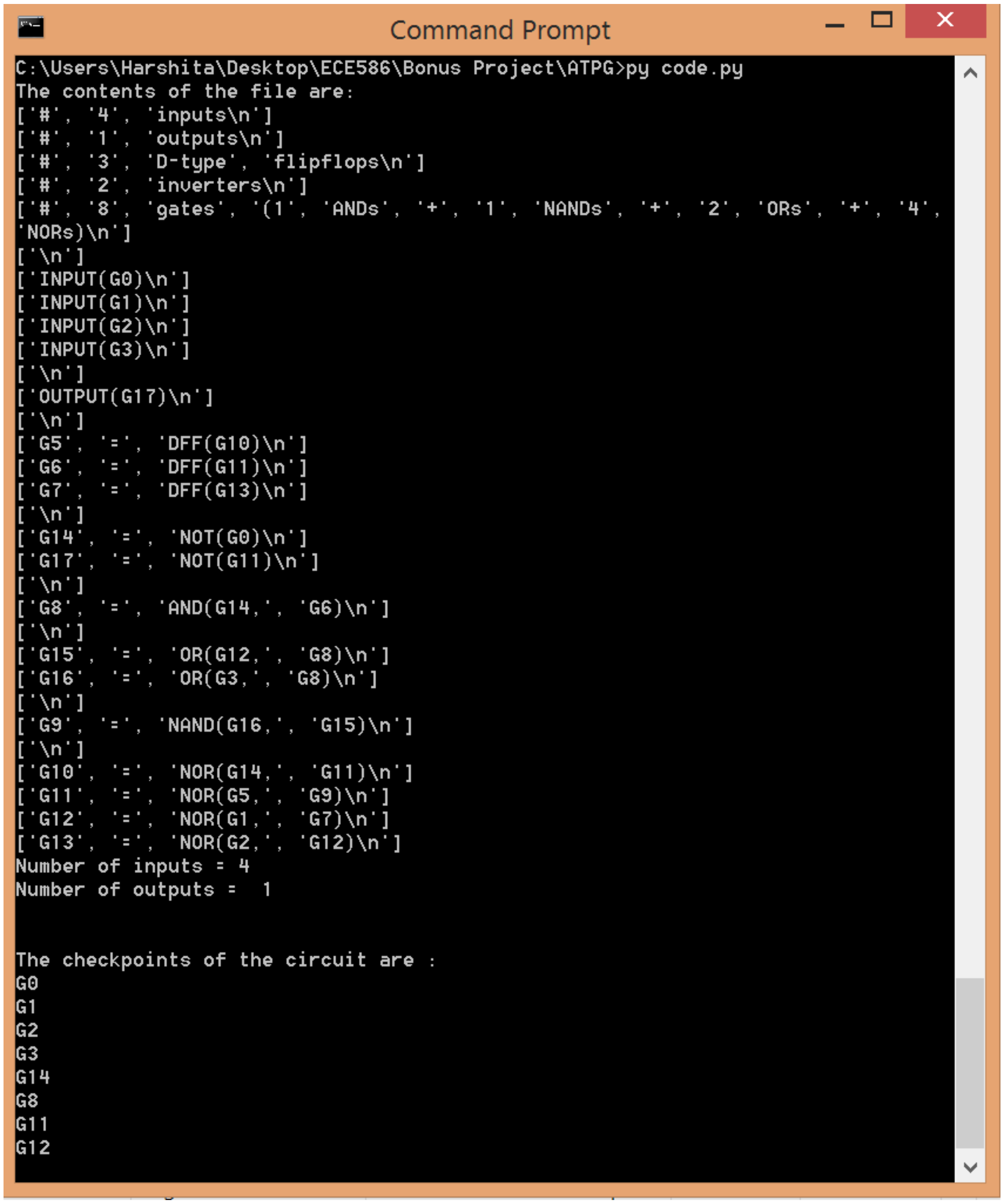
CONCLUSION

Therefore, the ATPG algorithm has been successfully implemented as explained above. The code of both the checkpoint generation and the test vector generation has been attached. The screenshots for 3 benchmark files for both the checkpoint generation and the test vector generation have been attached in the appendix below.

APPENDIX

1. S27.bench

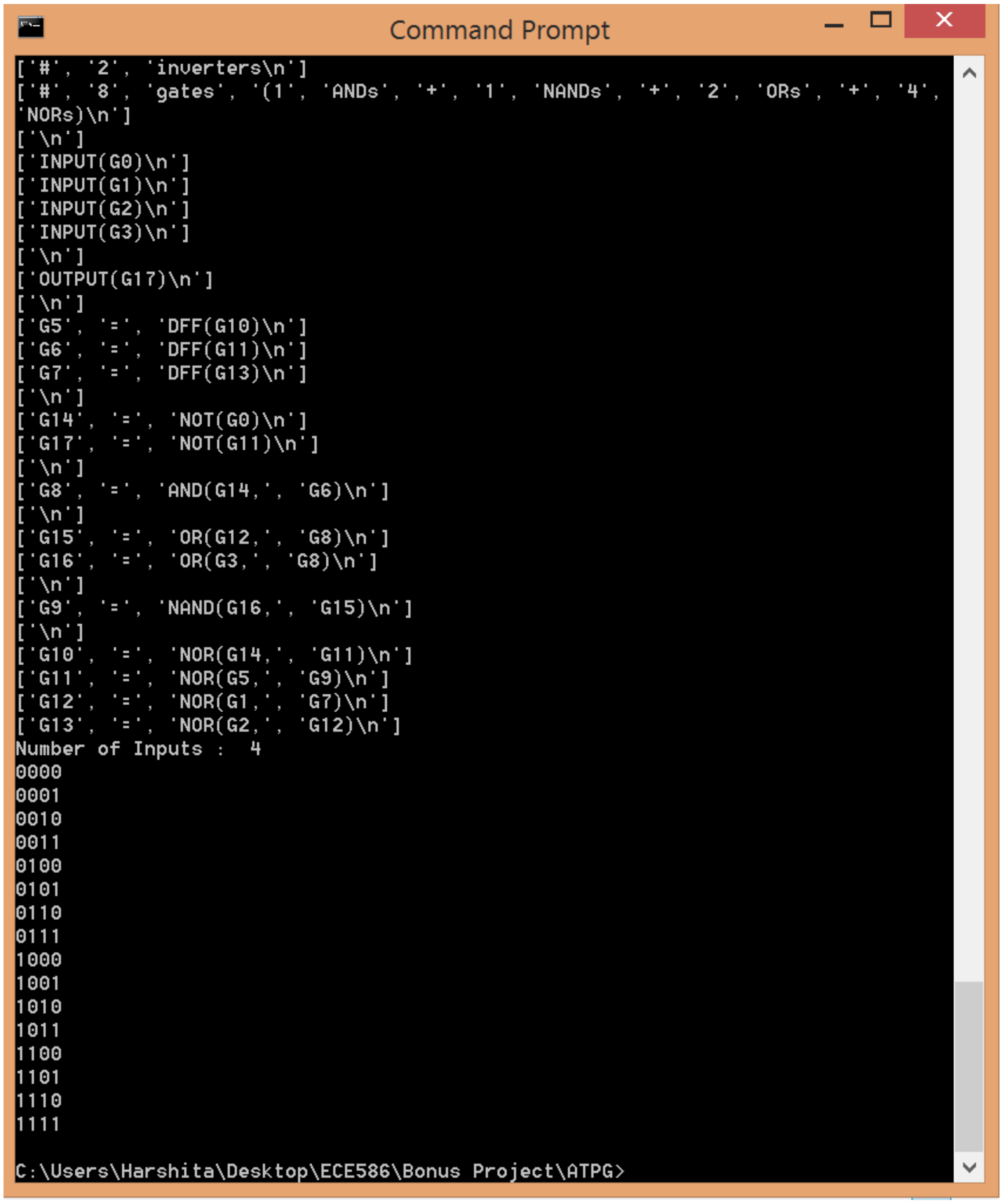
Checkpoint generation:



```
C:\Users\Harshita\Desktop\ECE586\Bonus Project\ATPG>py code.py
The contents of the file are:
['#', '4', 'inputs\n']
['#', '1', 'outputs\n']
['#', '3', 'D-type', 'flipflops\n']
['#', '2', 'inverters\n']
['#', '8', 'gates', '(1', 'ANDs', '+', '1', 'NANDs', '+', '2', 'ORs', '+', '4',
'NORs)\n']
['\n']
['INPUT(G0)\n']
['INPUT(G1)\n']
['INPUT(G2)\n']
['INPUT(G3)\n']
['\n']
['OUTPUT(G17)\n']
['\n']
['G5', '=', 'DFF(G10)\n']
['G6', '=', 'DFF(G11)\n']
['G7', '=', 'DFF(G13)\n']
['\n']
['G14', '=', 'NOT(G0)\n']
['G17', '=', 'NOT(G11)\n']
['\n']
['G8', '=', 'AND(G14,', 'G6)\n']
['\n']
['G15', '=', 'OR(G12,', 'G8)\n']
['G16', '=', 'OR(G3,', 'G8)\n']
['\n']
['G9', '=', 'NAND(G16,', 'G15)\n']
['\n']
['G10', '=', 'NOR(G14,', 'G11)\n']
['G11', '=', 'NOR(G5,', 'G9)\n']
['G12', '=', 'NOR(G1,', 'G7)\n']
['G13', '=', 'NOR(G2,', 'G12)\n']
Number of inputs = 4
Number of outputs =  1


The checkpoints of the circuit are :
G0
G1
G2
G3
G14
G8
G11
G12
```

Test Vector generation:

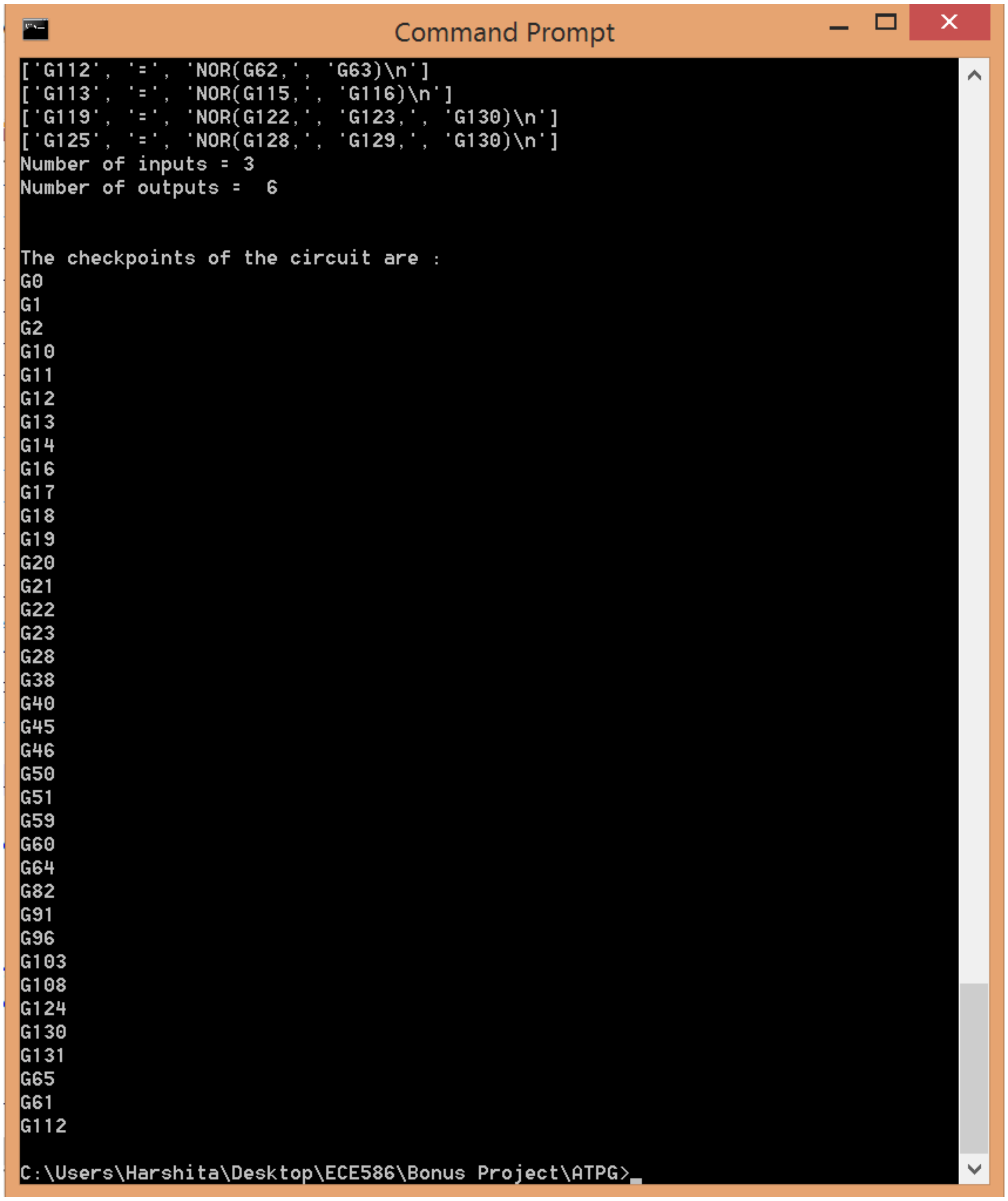


```
['#', '2', 'inverters\n']
['#', '8', 'gates', '(1', 'ANDs', '+', '1', 'NANDs', '+', '2', 'ORs', '+', '4',
'NORs)\n']
['\n']
['INPUT(G0)\n']
['INPUT(G1)\n']
['INPUT(G2)\n']
['INPUT(G3)\n']
['\n']
['OUTPUT(G17)\n']
['\n']
['G5', '=', 'DFF(G10)\n']
['G6', '=', 'DFF(G11)\n']
['G7', '=', 'DFF(G13)\n']
['\n']
['G14', '=', 'NOT(G0)\n']
['G17', '=', 'NOT(G11)\n']
['\n']
['G8', '=', 'AND(G14,', 'G6)\n']
['\n']
['G15', '=', 'OR(G12,', 'G8)\n']
['G16', '=', 'OR(G3,', 'G8)\n']
['\n']
['G9', '=', 'NAND(G16,', 'G15)\n']
['\n']
['G10', '=', 'NOR(G14,', 'G11)\n']
['G11', '=', 'NOR(G5,', 'G9)\n']
['G12', '=', 'NOR(G1,', 'G7)\n']
['G13', '=', 'NOR(G2,', 'G12)\n']
Number of Inputs :  4
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

C:\Users\Harshita\Desktop\ECE586\Bonus Project\ATPG>
```

2. S298.bench

Checkpoints generation:

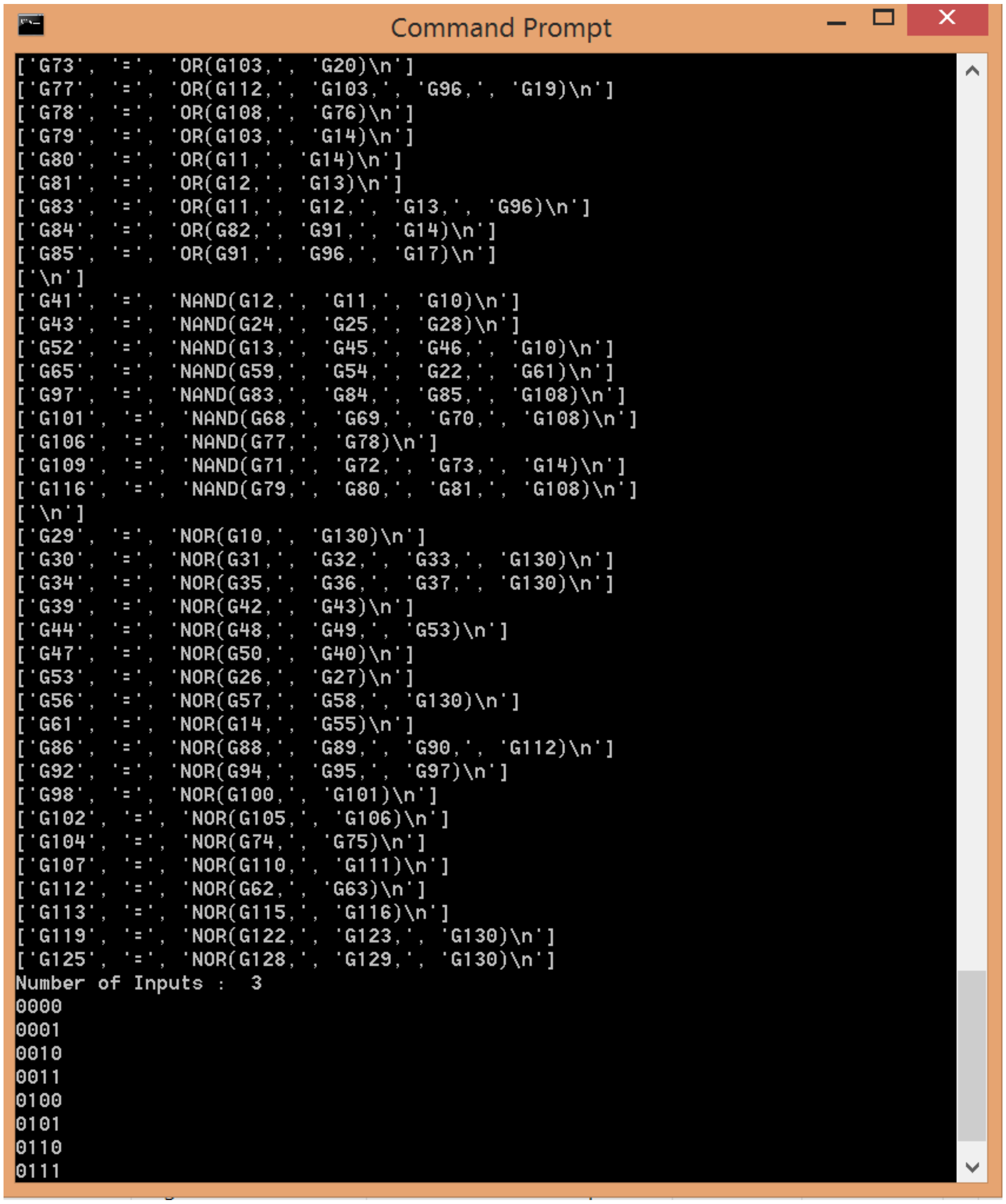

Command Prompt

```
['G112', '=', 'NOR(G62,', 'G63)\n']
['G113', '=', 'NOR(G115,', 'G116)\n']
['G119', '=', 'NOR(G122,', 'G123,', 'G130)\n']
['G125', '=', 'NOR(G128,', 'G129,', 'G130)\n']
Number of inputs = 3
Number of outputs =  6


The checkpoints of the circuit are :
G0
G1
G2
G10
G11
G12
G13
G14
G16
G17
G18
G19
G20
G21
G22
G23
G28
G38
G40
G45
G46
G50
G51
G59
G60
G64
G82
G91
G96
G103
G108
G124
G130
G131
G65
G61
G112

C:\Users\Harshita\Desktop\ECE586\Bonus Project\ATPG>_
```

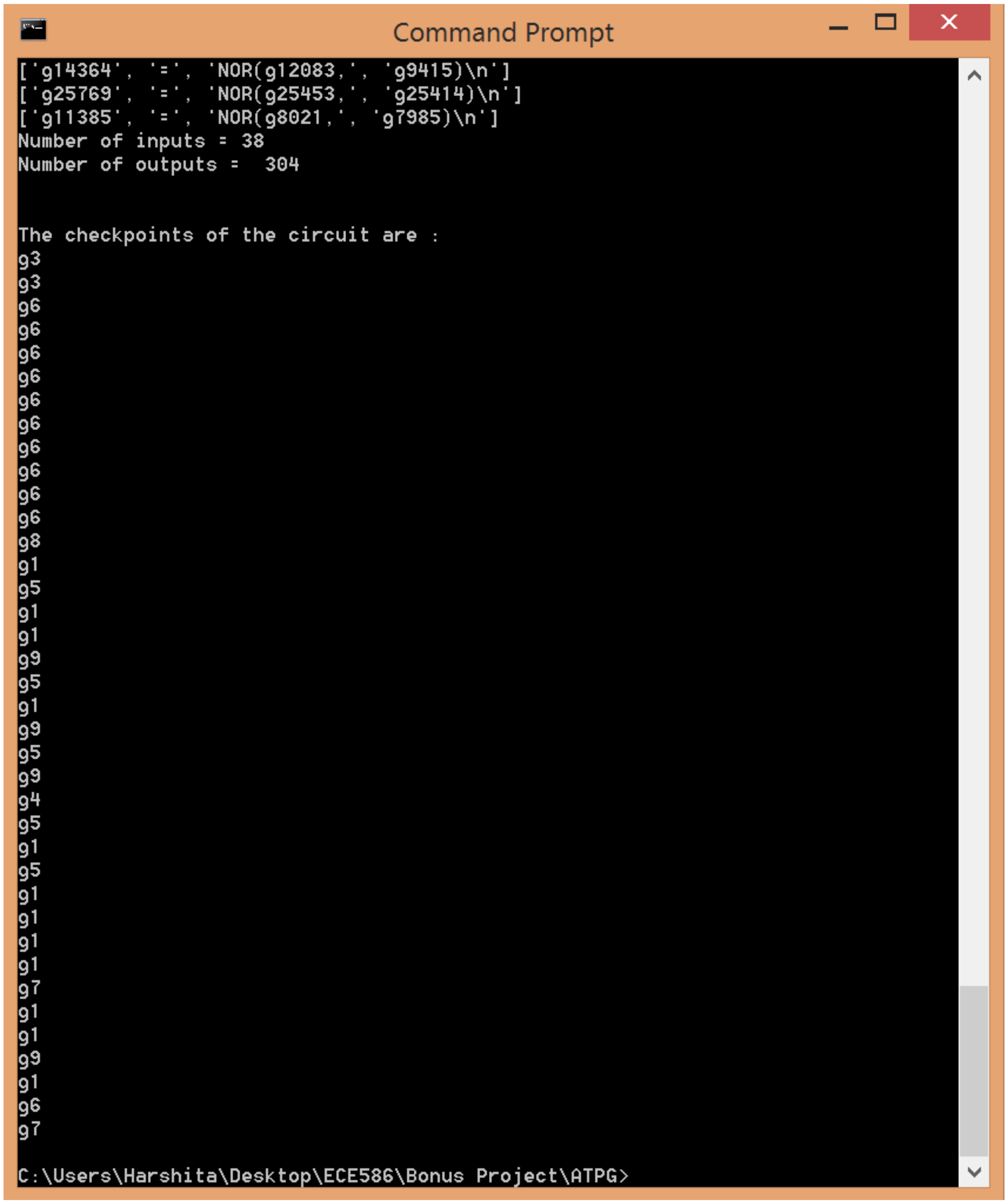
Test Vector generation:



```
['G73', '=', 'OR(G103,', 'G20)\n']
['G77', '=', 'OR(G112,', 'G103,', 'G96,', 'G19)\n']
['G78', '=', 'OR(G108,', 'G76)\n']
['G79', '=', 'OR(G103,', 'G14)\n']
['G80', '=', 'OR(G11,', 'G14)\n']
['G81', '=', 'OR(G12,', 'G13)\n']
['G83', '=', 'OR(G11,', 'G12,', 'G13,', 'G96)\n']
['G84', '=', 'OR(G82,', 'G91,', 'G14)\n']
['G85', '=', 'OR(G91,', 'G96,', 'G17)\n']
['\n']
['G41', '=', 'NAND(G12,', 'G11,', 'G10)\n']
['G43', '=', 'NAND(G24,', 'G25,', 'G28)\n']
['G52', '=', 'NAND(G13,', 'G45,', 'G46,', 'G10)\n']
['G65', '=', 'NAND(G59,', 'G54,', 'G22,', 'G61)\n']
['G97', '=', 'NAND(G83,', 'G84,', 'G85,', 'G108)\n']
['G101', '=', 'NAND(G68,', 'G69,', 'G70,', 'G108)\n']
['G106', '=', 'NAND(G77,', 'G78)\n']
['G109', '=', 'NAND(G71,', 'G72,', 'G73,', 'G14)\n']
['G116', '=', 'NAND(G79,', 'G80,', 'G81,', 'G108)\n']
['\n']
['G29', '=', 'NOR(G10,', 'G130)\n']
['G30', '=', 'NOR(G31,', 'G32,', 'G33,', 'G130)\n']
['G34', '=', 'NOR(G35,', 'G36,', 'G37,', 'G130)\n']
['G39', '=', 'NOR(G42,', 'G43)\n']
['G44', '=', 'NOR(G48,', 'G49,', 'G53)\n']
['G47', '=', 'NOR(G50,', 'G40)\n']
['G53', '=', 'NOR(G26,', 'G27)\n']
['G56', '=', 'NOR(G57,', 'G58,', 'G130)\n']
['G61', '=', 'NOR(G14,', 'G55)\n']
['G86', '=', 'NOR(G88,', 'G89,', 'G90,', 'G112)\n']
['G92', '=', 'NOR(G94,', 'G95,', 'G97)\n']
['G98', '=', 'NOR(G100,', 'G101)\n']
['G102', '=', 'NOR(G105,', 'G106)\n']
['G104', '=', 'NOR(G74,', 'G75)\n']
['G107', '=', 'NOR(G110,', 'G111)\n']
['G112', '=', 'NOR(G62,', 'G63)\n']
['G113', '=', 'NOR(G115,', 'G116)\n']
['G119', '=', 'NOR(G122,', 'G123,', 'G130)\n']
['G125', '=', 'NOR(G128,', 'G129,', 'G130)\n']
Number of Inputs :  3
0000
0001
0010
0011
0100
0101
0110
0111
```

3. S38584.bench

Checkpoints generation:

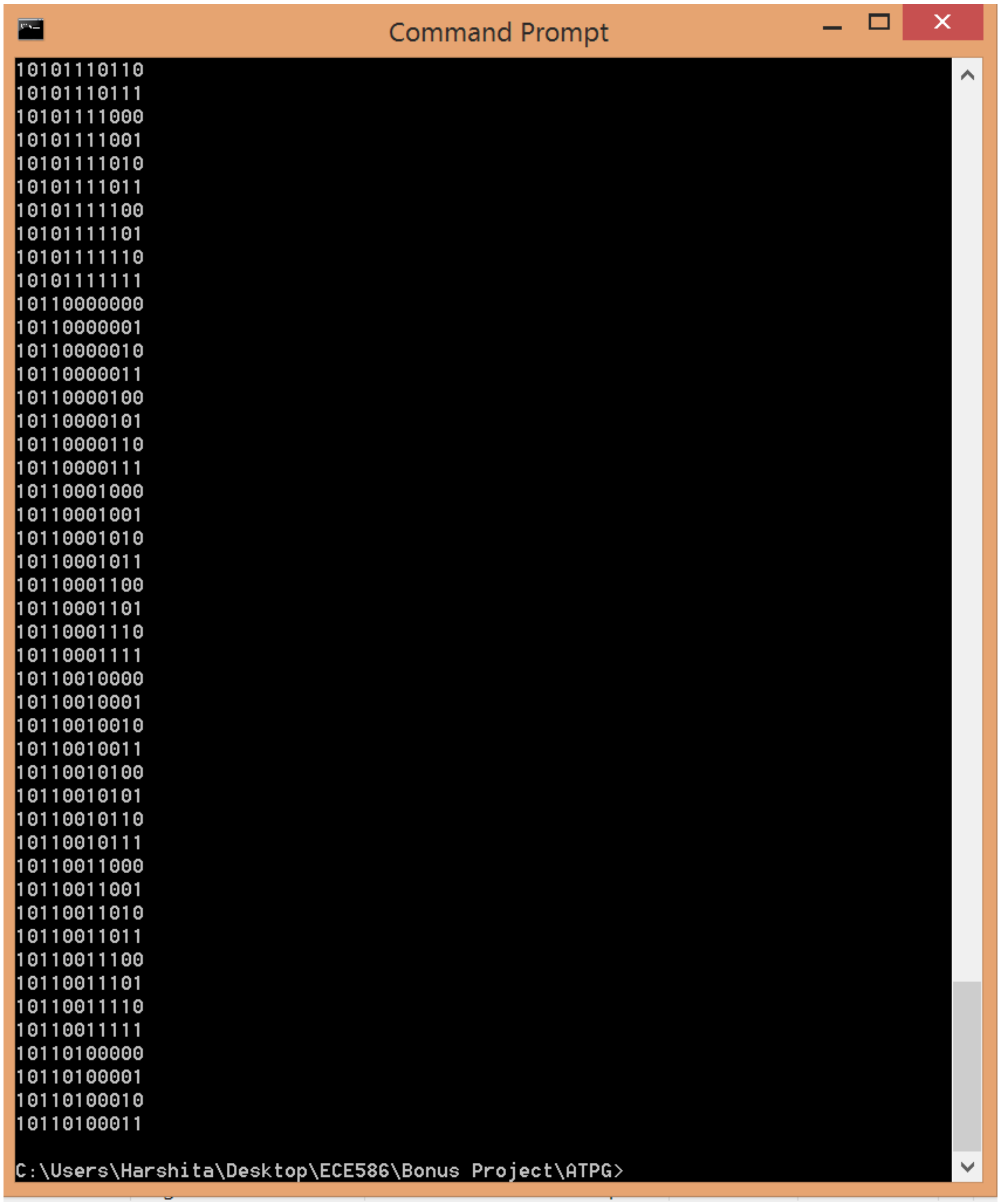


```
['g14364', '=', 'NOR(g12083,', 'g9415)\n']
['g25769', '=', 'NOR(g25453,', 'g25414)\n']
['g11385', '=', 'NOR(g8021,', 'g7985)\n']
Number of inputs = 38
Number of outputs =  304


The checkpoints of the circuit are :
g3
g3
g6
g6
g6
g6
g6
g6
g6
g6
g6
g6
g8
g1
g5
g1
g1
g9
g5
g1
g9
g5
g9
g4
g5
g1
g5
g1
g1
g1
g1
g7
g1
g1
g9
g1
g6
g7

C:\Users\Harshita\Desktop\ECE586\Bonus Project\ATPG>
```

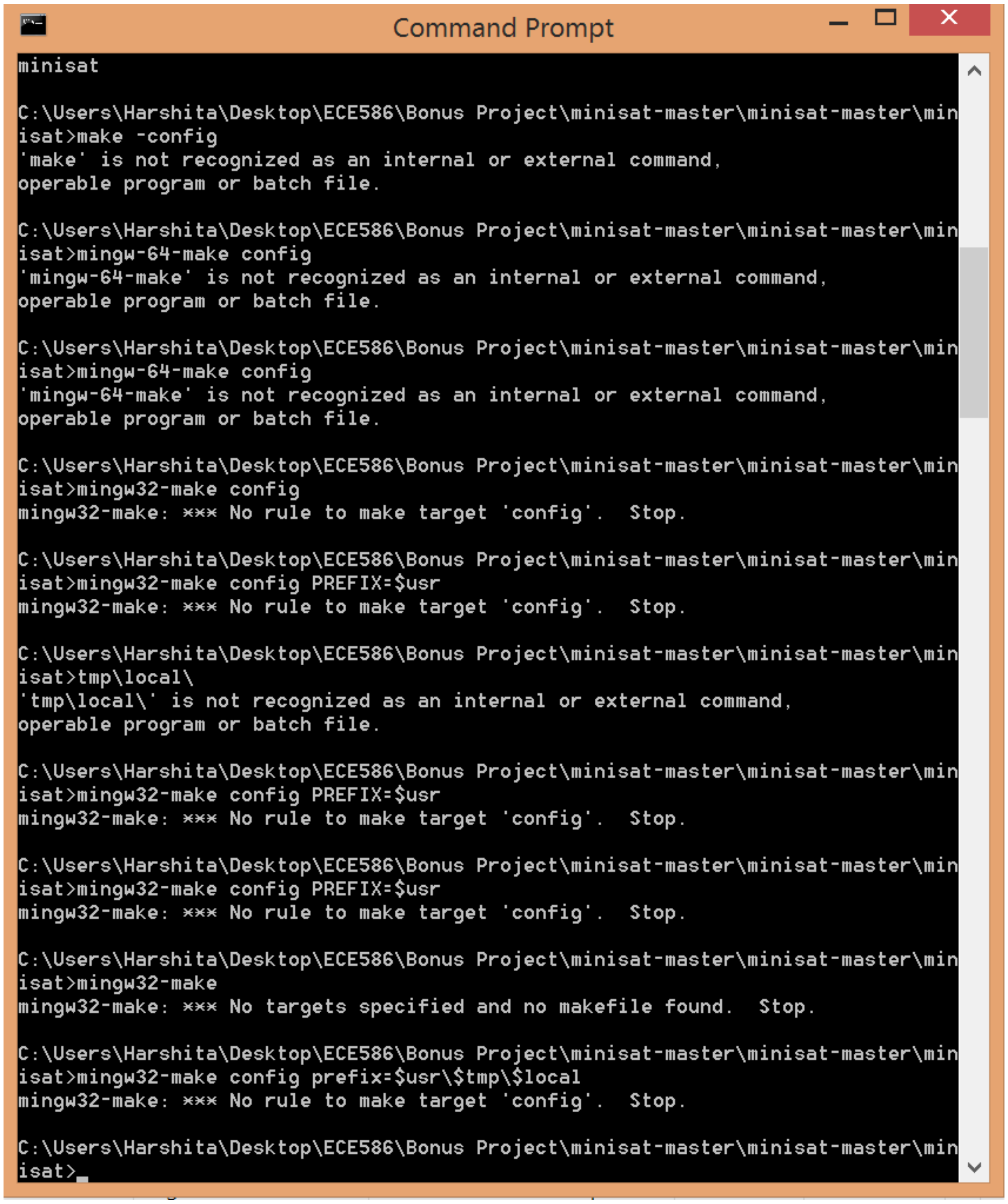
Test vector generation:



```
10101110110
10101110111
10101111000
10101111001
10101111010
10101111011
10101111100
10101111101
10101111110
10101111111
10110000000
10110000001
10110000010
10110000011
10110000100
10110000101
10110000110
10110000111
10110001000
10110001001
10110001010
10110001011
10110001100
10110001101
10110001110
10110001111
10110010000
10110010001
10110010010
10110010011
10110010100
10110010101
10110010110
10110010111
10110011000
10110011001
10110011010
10110011011
10110011100
10110011101
10110011110
10110011111
10110100000
10110100001
10110100010
10110100011

C:\Users\Harshita\Desktop\ECE586\Bonus Project\ATPG>
```

4. MiniSAT solver:



Command Prompt

```
minisat

C:\Users\Harshita\Desktop\ECE586\Bonus Project\minisat-master\minisat-master\min
isat>make -config
'make' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Harshita\Desktop\ECE586\Bonus Project\minisat-master\minisat-master\min
isat>mingw-64-make config
'mingw-64-make' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Harshita\Desktop\ECE586\Bonus Project\minisat-master\minisat-master\min
isat>mingw-64-make config
'mingw-64-make' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Harshita\Desktop\ECE586\Bonus Project\minisat-master\minisat-master\min
isat>mingw32-make config
mingw32-make: *** No rule to make target 'config'.  Stop.

C:\Users\Harshita\Desktop\ECE586\Bonus Project\minisat-master\minisat-master\min
isat>mingw32-make config PREFIX=$usr
mingw32-make: *** No rule to make target 'config'.  Stop.

C:\Users\Harshita\Desktop\ECE586\Bonus Project\minisat-master\minisat-master\min
isat>tmp\local\
'tmp\local\' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Harshita\Desktop\ECE586\Bonus Project\minisat-master\minisat-master\min
isat>mingw32-make config PREFIX=$usr
mingw32-make: *** No rule to make target 'config'.  Stop.

C:\Users\Harshita\Desktop\ECE586\Bonus Project\minisat-master\minisat-master\min
isat>mingw32-make config PREFIX=$usr
mingw32-make: *** No rule to make target 'config'.  Stop.

C:\Users\Harshita\Desktop\ECE586\Bonus Project\minisat-master\minisat-master\min
isat>mingw32-make
mingw32-make: *** No targets specified and no makefile found.  Stop.

C:\Users\Harshita\Desktop\ECE586\Bonus Project\minisat-master\minisat-master\min
isat>mingw32-make config prefix=$usr\$tmp\$local
mingw32-make: *** No rule to make target 'config'.  Stop.

C:\Users\Harshita\Desktop\ECE586\Bonus Project\minisat-master\minisat-master\min
isat>_
```