Programming Assignment #2
PODEM

Introduction:

We want to start up an EDA company that creates an NTU-ATPG *(automatic test pattern generation)* tool for single stuck-at faults. This NTU-ATPG system has two major functions: test pattern generation and fault simulation. For test pattern generation, it implements the PODEM algorithm [Goel 1981]. For fault simulation, it implements the parallel fault simulation algorithm. In this PA, we first show you how to run *ATPG* mode. Then, you are asked to complete the code in the TODO sections in the PODEM code.

Tutorial:

First, please enter the bin directory, and run the golden binary to see the results of *ATPG* mode. Note that you should run these commands on *Linux* platforms with GCC version 4.8 or newer. Otherwise, it may not work. Simply type the following commands.

```
cd bin
./golden_atpg ../sample_circuits/c17.ckt > ../pattern/c17.ptn
```

Then you will see results generated by PODEM in file "`../pattern/c17.ptn`".
Note that operator ">" saves the result to the file, so no output will be shown on the terminal.
Second, leave the bin directory, enter the *src* directory, and compile the source code by typing.

```
cd ..
cd src
make
```

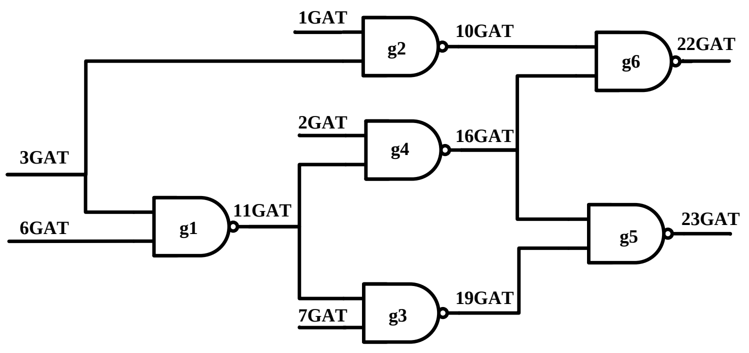
Make sure that the version of the compiler must support C^{++} 11. In case you see any compilation error, please fix it before you proceed. If you succeed, an executable file '`atpg`' should be correctly generated. Then, in the same directory, run this software in ATPG mode by typing the following command.

```
./atpg ../sample_circuits/c17.ckt > ../pattern/c17.ptn
```

The number of total faults, detected faults, and fault coverage should be shown in ./`patterns/c17.ptn`. Note that the fault coverage is not correct now because the TODO section is empty in the source codes we provided to you. Your job is to complete the code in the TODO sections so that your outputs are the same as those generated by the golden binary.

I/O Files:

We explain the results using the following circuit, *c17*, which is an ISCAS (international symposium of circuit and system) benchmark circuit.



Programming Assignment #2

Figure 1. *c17* circuit

Our example circuit file (c17.ckt) is shown below. The first line indicates the name of the circuit after the keyword "name". Then, the circuit primary inputs (PIs) and primary outputs (POs) are shown after the keyword *"i"* and *"o",* respectively.

The rest of the file stores logic gates. Logic gates are named in the format *"g#"*, where # is the gate index. The input wires and the output wire of a logic gate are separated by a semicolon.

For example, *"g1 nand 6GAT(3) 3GAT(2) ; 11GAT(5)"* means NAND gate *g1* has two inputs: wire 6GAT and wire 3GAT. The output of NAND gate *g1* is wire 11GAT. This circuit has 11 wires in total. Note that a wire can have multiple fanout branches. For example, wire 11GAT feeds both g3 and g4.

(The numbers in the parentheses are *total gates indices*. They are unique numbers for each PI, PO, or logic gate.)

```
name C17.iscas
i 1GAT(0)
i 2GAT(1)
i 3GAT(2)
i 6GAT(3)
i 7GAT(4)

o 22GAT(10)
o 23GAT(9)

g1 nand 6GAT(3) 3GAT(2) ; 11GAT(5)
g2 nand 3GAT(2) 1GAT(0) ; 10GAT(6)
g3 nand 7GAT(4) 11GAT(5) ; 19GAT(7)
g4 nand 11GAT(5) 2GAT(1) ; 16GAT(8)
g5 nand 19GAT(7) 16GAT(8) ; 23GAT(9)
g6 nand 16GAT(8) 10GAT(6) ; 22GAT(10)
```

Figure 2. *c17* circuit file

The following file is our test pattern (*c17.ptn*), which is generated by our ATPG tool. The circuit data is shown at the beginning. Then, all test vectors are shown after the keyword "T". The other ATPG results are shown after test vectors: fault coverage, number of backtracks, etc.

```
#Circuit Summary:
#---------------
#number of inputs = 5
#number of outputs = 2
#number of gates = 6
#number of wires = 11
#atpg: cputime for reading in circuit ../sample_circuits/c17.sim: 0.0s 0.0s
#atpg: cputime for levelling circuit ../sample_circuits/c17.sim: 0.0s 0.0s
#atpg: cputime for rearranging gate inputs ../sample_circuits/c17.sim: 0.0s 0.0s
#atpg: cputime for creating dummy nodes ../sample_circuits/c17.sim: 0.0s 0.0s
#number of equivalent faults = 22
#atpg: cputime for generating fault list ../sample_circuits/c17.sim: 0.0s 0.0s
T'00110'
T'10111'
T'10001'
T'01000'
T'11011'
```

Programming Assignment #2

```
T'01100'
T'10000'
T'01111'
#number of aborted faults = 0
#number of redundant faults = 0
#number of calling podem1 = 8
#total number of backtracks = 0
#FAULT COVERAGE RESULTS :
#number of test vectors = 8
#total number of gate faults = 34
#total number of detected faults = 34
#total gate fault coverage = 100.00%
#number of equivalent gate faults = 22
#number of equivalent detected faults = 22
#equivalent gate fault coverage = 100.00%
#atpg: cputime for test pattern generation ../sample_circuits/c17.sim: 0.0s 0.0s
```

Figure 3. *c17* pattern file (generated by ATPG)

Assignments:

The *readme* file contains essential information about the data structure of this PODEM code, so please read it carefully before you begin. You will need to write some code to fill in the TODO sections in the fault simulation code. Go to the *src* directory and open the file *podem.cpp*. You will see three TODO sections in this file. You can search for the keyword "TODO" to locate them. Your task is to complete the following four functions:

1) *podem*: this function implements the flow of PODEM algorithm.
2) *find_hardest_control:* this function finds the hardest to control input among all PIs
3) *find_easiest_control:* this function finds the easiest to control input among all PIs
4) *trace_unknown_path*: this function performs the x-path tracing that we learned in class. Please analyze the complexity of the function *trace_unknown_path()* in your report.

Once you finish coding the TODO sections, you should get the same fault coverage as our golden results.

1) Please write a report to explain what you have done in this PA. Also, fill in the following table in your report.

circuit number	number of gates	number of total faults	number of detected faults	number of undetected faults	fault coverage	number of test vector	run time
C432							
C499							
C880							
C1355							
C2670							
C3540							
C6288							
C7552							

Programming Assignment #2

2) (10% bonus) Please analyze the complexity of the function: trace_unknown_path. Can you implement this function with *O(n)* complexity where *n* is the number of nodes? What is the tradeoff of your implementation? Please explain what you did in your report.
Hint: trace_unknown_path can be implemented easily by recursively searching the unknown fanout of each node. However, the run time may be exponential growth if you trace the same path many times. You can reduce the search space by recording the path (or node) you have traced.

Grading:
80% correctness*
(If you get any two circuits correct you get 50%, and then get 5% each for the other six circuits)
20% report
10% bonus

*correctness will be evaluated by your **.ptn* file. If your fault coverage is lower than the golden binary, your score will be discounted.

Submission:
Make a directory <*student_id*>*_pa2*
Please copy 2 items /*src*, *report.pdf* into the directory. Then submit a single *.*tgz* file to NTU COOL. Include everything so that your code can be easily compiled using 'make'. You can use the following command to compress a whole directory:

```
tar -zcvf <student_id>_pa2.tgz <dir>
```

Here's a reference file structure:

```
r10943147_pa2/
        ├──src/         #Including *.cpp, *.h and makefile  only
        └──report.pdf   #Fill the table above and highlight your change of code
```

Reference:
[Goel 1981] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," IEEE Trans. on Computers, Vol. 30, No.3, pp215-222, 1981.

CAUTION!! We will check for plagiarism. Copying source code results in zero grades for both students!!
