EE658
Fall 2013
USC

Class Project

Group 8(in order of parts):

Jun Ren, Xinchun Wang,

Fangwei Situ, Cheng Ma

Content

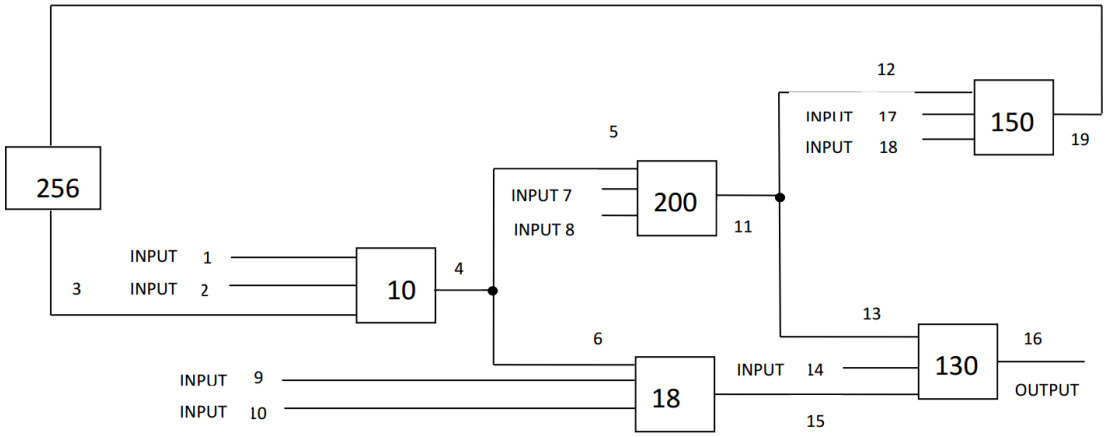
Preprocessor [Jun Ren]

Inputs: original circuit netlist (netlist.txt)
originalfault lists (original_fault_list.txt)
Outputs: levelizednetlist (logic_level.txt),
scan version of the netlist (scan_version_netlist.txt)
equivalence and dominance relationships
(fault_equivalence_dominance_relationship.txt)
filtered fault list(filtered_fault_list.txt)

Function: build and enhance data structure;
Fault collapsing
Levelize circuit
Strip out F/Fs and flag associated lines as PPOs or PPIs
I/O for controlling system flow if an truly integrated system

Algorithm:

1. scan versionnetlist: go through the provided original netlist, check each CLB originalnetlist standard (case 2):



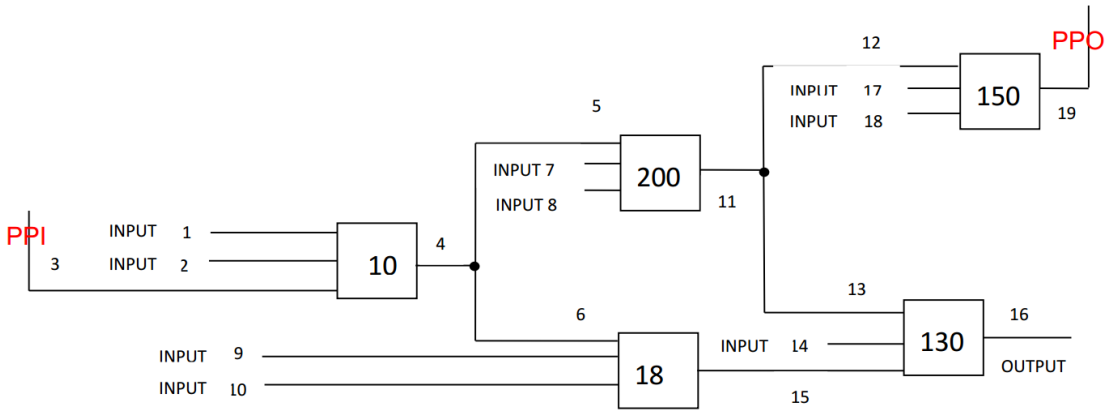
1 1 259 1 0
1 2 259 1 0
0 3 256 1 1 19
0 4 10 2 3 1 2 3
2 5 260 4
2 6 260 4
1 7 259 1 0
1 8 259 1 0
1 9 259 1 0
1 10 259 1 0
0 11 200 2 3 5 7 8
2 12 260 11

2 13 260 11
1 14 259 1 0
0 15 18 1 3 6 9 10
3 16 130 0 3 13 14 15
1 17 259 1 0
1 18 259 1 0
0 19 150 1 3 12 17 18

1st column: 0 – CLB output, 1 – PI, 2 –BR, 3 – PO
2nd column: line number
3rd column: gate type (Flip-flips are gate-type 256, Inputs are 259, Branches are 260)
4th column: # of fout
5th column: # of fin
6th column: fin lines number

if a CLB is FF(gate type is 256), strip out the FF, flag the input of FF asPPO and the output as PPI. Also, PPI has the same format as PI but starting with 4instead of 0, PPO has the same format as PO but starting with 5 instead of 0.

So we just need to copy the original netlist to the scan version netlist if the line is not input or output of FF; otherwise, change the 1st column value of the line to 4 or 5, and change the related fin and fout value. So the scan version netlist of case 2 should be:



1 1 259 1 0
1 2 259 1 0
4 3 259 1 0
0 4 10 2 3 1 2 3
2 5 260 4
2 6 260 4
1 7 259 1 0
1 8 259 1 0
1 9 259 1 0
1 10 259 1 0
0 11 200 2 3 5 7 8

2 12 260 11
2 13 260 11
1 14 259 1 0
0 15 18 1 3 6 9 10
3 16 130 0 3 13 14 15
1 17 259 1 0
1 18 259 1 0
5 19 150 0 3 12 17 18

2. Levelization

In order to simplify the levelization, we need to creat index, NumInpsReady,uppernode and downnode adding to the structure. And after stripping off FF, we can levelize based on the scan version netlist.

The fundamental thought is:

a) We need add an extra field called, NumInpsReady, to the data structure of each circuit element. Initially, all NumInpsReady for each element should be initialized to zero.
b) Also we need set a global variable array called Readyforcomputation, to help store the pointers indexed the nodes which have received all the information of levels from their upstream nodes. Of course we need allocate the memory space required by the new array Readyforcomputation.
c) Then we use levels of all the primary inputs to increment their downstream nodes'NumInpsReady. If the value of one downstream node's NumInpsReady matches its fanin, we can store the downstream node into the Readyforcomputation array.
d) After that, we can use the for loop of Readyforcomputation array to calculate each node's level value. Because new level-decided nodes can help increment its downstream nodes'Readyforcomputation, creating more nodes which are ready for calculating their level values.
e) Finally, we can levelize all the nodes.

After levelization, the output of level corresponding to each node should be as below:

1 0
2 0
3 0
4 1
5 2
6 2
7 0
8 0
9 0
10 0
11 3

12 4
13 4
14 0
15 3
16 5
17 0
18 0
19 5

3. Fault equivalence and dominance relationship

After build the information of PO and PPO of good circuit corresponding to exhaustive patterns of PI and PPI, firstly we need to find the detectable faults, whose PO or PPO are different with the ones of good circuit. And the pointers of these faults are stored in array.

Then comparing the each pairs of detectable faults, to find the equivalence and dominance relationship among them.

In this project, we can simulate all the faults in the original fault list, for each fault we need to go through all the possible inputs patterns, then we will get the outputs corresponding to each pattern, creating truth table for each fault. Here we need to use scan version netlist, striping off the FF and flag the PPI and PPO. Through comparing the BSF of each fault, we can get their relationship. And we can tell the detectable faults from the original faults. We use XOR to compare the faulty PO or PPO with the good ones, building the bitwise result detect[i] and detectPPO[i].

Then according to their detect[i] ordetectPPO[i], we can find the dominance relationship.

(a)

0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0

(b)

0	0	0	1	1	0	0	0
0	0	0	1	0	0	0	0

(c)

0	0	0	1	0	0	0	0
0	0	0	1	1	0	0	0

(d)

0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0

E.g. the detect[i] of case 1 maybe like the array above. Here we compare faultA and faultB. There are 4 possible kinds of conditions. In (a), A and B are equivalent; in (b), A dominates B; in (c), B dominates A; in (d), there's no equivalence or dominance relationship between them

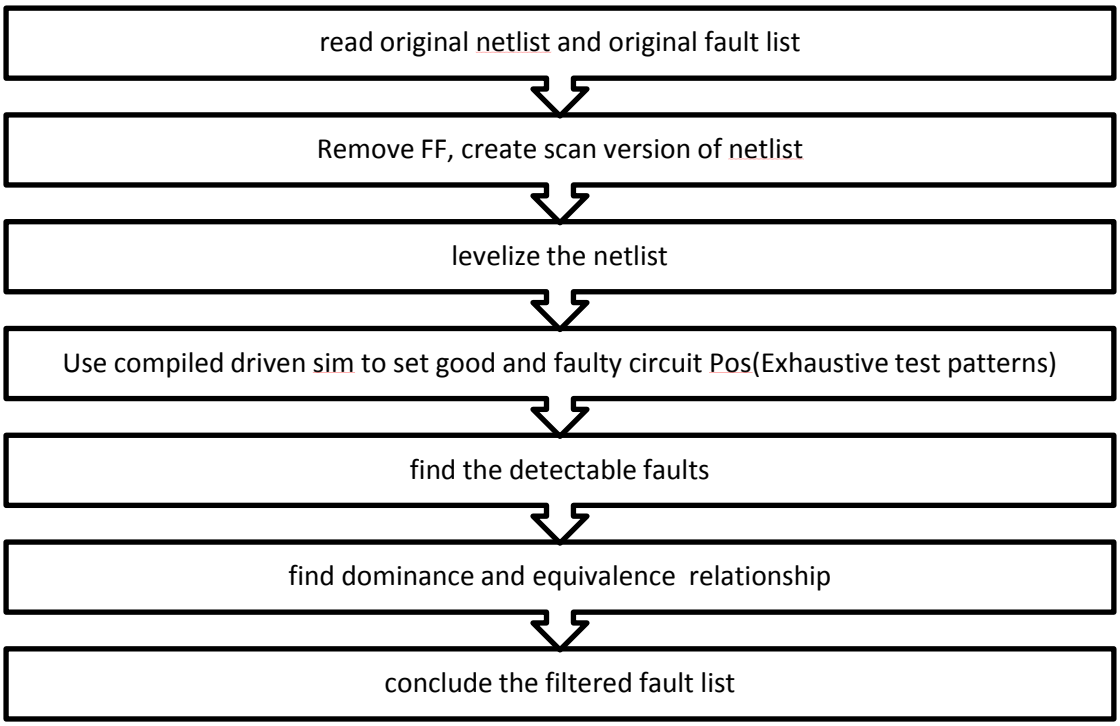
Data structure of Node[i]

```
typedef struct n_struc {
   unsigned indx;              /* node index(from 0 to NumOfLine - 1 */
   unsigned ntype;             //node type
   unsigned num;               /* line number(May be different from indx */
   unsigned gtype;          /* gate type */ //0~260
   unsigned fin;               /* number of fanins */
   unsigned fout;              /* number of fanouts */
   struct n_struc **unodes;    /* pointer to array of up nodes */
   struct n_struc **dnodes;    /* pointer to array of down nodes */
   int level;                  /* level of the gate output */
   int NumInpsReady;      /* Number of ready inputs*/
   int value;  //node value
} NSTRUC;
```

Data structure of Fault[i]

```
typedef struct f_struc {
    int nodenum;
    NSTRUC *node;
    int ftype; //fault type
    int *PO; //the array store all PO values corresponding to each vector in order
    int *PPO;
    int *detect; //bitwise XOR with good circuit PO, to detect any difference
    int *detectPPO;
    struct f_struc **DomiFaults; //the array stores the dominated faults by this fault
    struct f_struc **EquiFaults; //the array stores the equivalent faults with this fault
    int Ndomi; //num of faults dominated
    int Nequi; //num of faults equaled
} FSTRUC;
```

Flowchart


read original netlist and original fault list
Remove FF, create scan version of netlist
levelize the netlist
Use compiled driven sim to set good and faulty circuit Pos(Exhaustive test patterns)
find the detectable faults
find dominance and equivalence relationship
conclude the filtered fault list

Logic simulator [Jun Ren]

Input: original circuit netlist (netlist.txt)
inertial, rise/fall delay value, andclock period (delay_values.txt)
initial state of the flip-flops, sequences applied to the primary inputs at each subsequent clock cycle (logic_simulator_input_sequences.txt)

Output: Timing diagrams at unit and clock-cycle level of selected gates and F/F outputs (waveform.txt)
State of all flip-flops after each clock edge and the state of all outputs before each clock edge (PO_FF_state.txt)
The value of the input to a flip-flop changes at the sampling time or one unit of time before the sampling time (report_on_glitches.txt)

Function: Run in unit-time mode or clock-cycle mode (0 delay)
Identify all glitches at inputs to F/Fs and on POs in unit time mode

Algorithm

1. Evaluate CLB

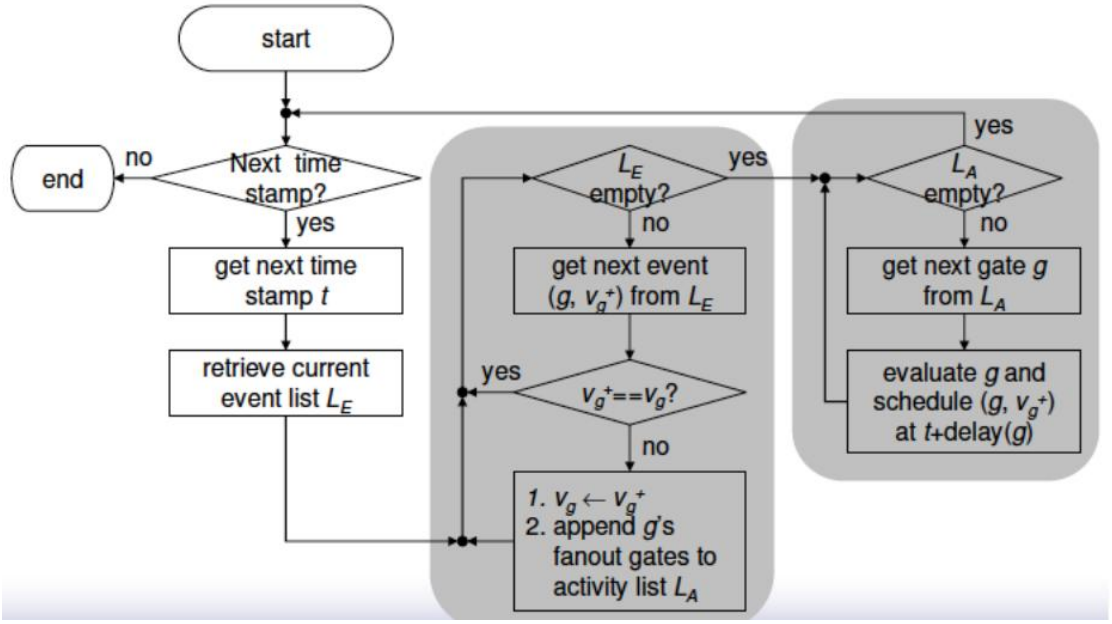
After the logic simulator read the netlist, it can build truth table for each CLB according to the CLB TYPE. When evaluate a CLB in the process, we can use the 3 uppernodes of the gate line to point to the output in the truth table.

2. Clock-cycle mode

Clock-cycle mode uses compiled driven simulation, so here we still need to levelize the netlist regardless of delay. The logic simulator read the input vectors, executing the model for every vector, displaying the result. Then in the next cycle, reading the 2^{nd} vectors. Each cycle the model reads the next vectors from file. For FF, the initial state value given is used as input, then in later cycles, we copy the input of FF in the last cycle to the output of FF. This kind of model assumes that during every cycle, each line value will be updated.

3. Unit-time mode

Unit-time mode uses 2-pass event-driven logic simulation. Although one-pass strategy isconcluded to be more efficient than two-pass strategy in the textbook, considering the complexity of possible cancelling last event due to the conflict of scheduling time and the violation of inertial delay, we decide to use 2-pass method, avoiding repeated evaluations of the CLB. The flow diagram is as below:



The main procedure is: in every time stamp, we retrieve the current event list. In the first pass, update all the nodes' value if the new value is different with the old value, and then put the node's fanout gates into an activity list. After all the events have been measured, in the second pass, we evaluate every gate in the activity list, the delay is decided according to rise or fall transition. Then we also need to check if the scheduling violates the inertial delay.

We should arrange every new vector input the model every one clock cycle, then the new vector and new value stored in FF can input into the model at the same time.

4. The timing wheel

Because the header linked list of time stamps is dense, we decide to use an array of headers, so the search can be avoided by using the time t to provide an index to the array. In each header, the associated lists of scheduled events are stored. The array of headers and their associated lists store events scheduled to occur between the current simulation time tc and tc+M-1. M is the size of the array, which is set as the largest amount of delay plus 1 in this project. Any event scheduled for time tc+d, where d<M, can be inserted in the event list without search. If we want to schedule an event at t, then we should put the event in the position (t mod M). In another word, the array of size M is used circularly. When the event in time stamp tc is evaluated, new event scheduled is inserted in the time array.

For example, the delay_values.txt is as below

15

4 1 2 1

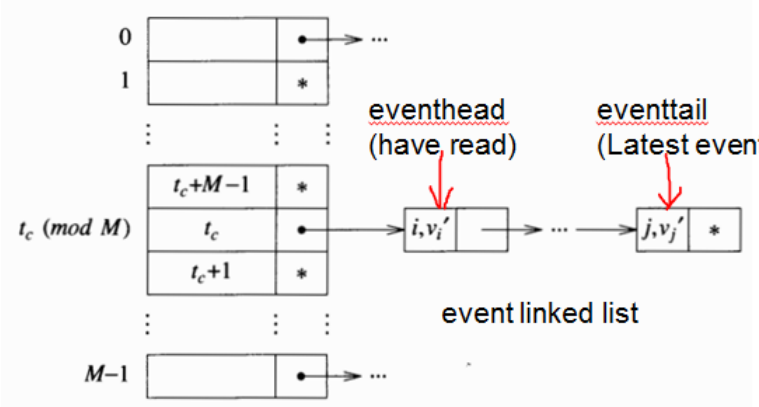
11 1 3 1

15 0 2 2

16 1 2 1

19 0 1 1

The largest amount of delay is 3, the unit time can be set 1. So M can be set as 3+1=4, the size of timearray is 4. If at array[1], tc=1, we want to schedule an event of 3 unit time delay, (1+3) mod 4 = 0, we need to insert the event into the array[0].



The array should include the time and the linked list of events, eventhead pointer and eventtail pointer.When all the events in a time stamp are updated andassociated events are scheduled, go to the next time stamp. In each events, there are Node[i] ,new value v' and the pointer pointing to the next event. Event links event, forming the linked list. The eventhead pointer is used to record last updated event; the eventtail pointer is used to record the latest event added to the tail of event list.

5. Waveform

In order to record the change and value of each node, we need a structure named DISPLAY to trace these transformations. As long as the value of nodes updated to a new value, the event will be recorded. Initially we set each node's value as 2, just indicating"x". The diagram below illustrate the event of node 2.We use GEVENT structure to record the event on the right.

num	Nevent	*event
1		
2	2	
3		
4		
5		
6		
…		

time	value
0	1
25	0

```
1 0(0-5), 1(5-10), 0(10-15), 1(15-20), 0(20-25), 1(25-30), 0(30-35), 1(35-40), 0(40-45), 1(45-50)
2 1(0-25), 0(25-50)
3 0(0-10), 1(10-35), 0(35-50)
4 0(0-5), 1(5-10), 0(10-15), 1(15-20), 0(20-25), 1(25-30), 0(30-35), 1(35-40), 0(40-45), 1(45-50)
5 0(0-5), 1(5-10), 0(10-15), 1(15-20), 0(20-25), 1(25-30), 0(30-35), 1(35-40), 0(40-45), 1(45-50)
6 1(0-25), 0(25-50)
7 1(0-25), 0(25-50)
8 0(0-10), 1(10-35), 0(35-50)
9 0(0-10), 1(10-35), 0(35-50)
10 x(0-4), 1(4-8), 0(8-19), 1(19-23), 0(23-34), 1(34-43), 0(43-49), 1(49-50)
11 x(0-4), 0(4-8), 1(8-34), 0(34-50)
```

6. Report the glitch of the input to FF

The glitch here means the input of FF changes at the sampling time or one unit time before the sampling time. So we define the event that input of FF changes one unit time before the rise edge of clock or at the rise edge as glitch.

For example, clock period = 15, if input of FF changes at time 14 or 15, they both belong to glitch.

Data structure of Node[i]

```
typedef struct n_struc {
    unsigned indx;              /* node index(from 0 to NumOfLine - 1 */
    unsigned ntype;             //node type
    unsigned num;               /* line number(May be different from indx */
    unsigned gtype;          /* gate type */ //0~260
    unsigned fin;               /* number of fanins */
    unsigned fout;              /* number of fanouts */
    struct n_struc **unodes;    /* pointer to array of up nodes */
    struct n_struc **dnodes;    /* pointer to array of down nodes */
    int level;                  /* level of the gate output */
    int NumInpsReady;      /* Number of ready inputs*/
    int value;  //node value

    int lsv; //lsv=last scheduled value
    unsigned inertialDelay;
    unsigned riseDelay;
    unsigned fallDelay;
} NSTRUC;
```

Data structure of Event[i]

```
typedef struct ScheduledEvent{
    NSTRUC* node;
    int value;  //new scheduled value after evaluation
    struct ScheduledEvent* nextEvent;
    //pointer points to next event //event link list
} SEVENT;
```

Data structure of Time stamp

```
typedef struct TimeStamp{
    int time;
    struct ScheduledEvent* eventtail;
    //use tail pointer to record the last inserting event in a time stamp
    struct ScheduledEvent* eventhead;
    //use head pointer to record the event before which all the events we have processed
    struct ScheduledEvent* event;
} TSTAMP;
```

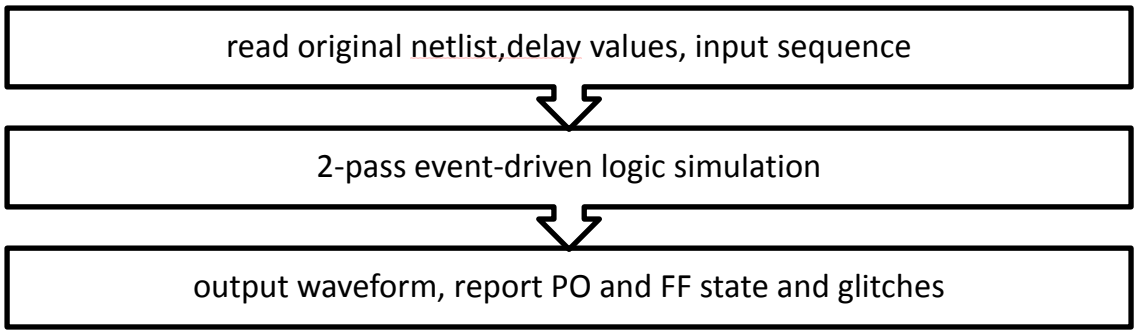
Data structure of display buffer

```
typedef struct display{  //display each node waveform
    int num;  //num of nodes
    int Nevents; //num of events of the node
    GEVENT* event;
} DISPLAY;
```

Data structure of graph event

```
typedef struct graphevent{
    int time;
    int value;
} GEVENT;
```

Flowchart



ATPG [Xinchun Wang]

Part I. Inputs and Outputs

The first input ATPG needs is a modified netlist, which is a scan version netlist. Originally there will be flip-flops in the circuit. Now it should be modified so that flip-flops will be removed and replaced by pseudo primary inputs and outputs. This is because ATPG for sequential circuit is very difficult to implement and we only do ATPG for combinational circuit in this project. The scan version netlist is provided by preprocessor module.

A scan version netlist contains at least 5 columns, corresponding to type, line number, function number, number of fan-out, number of fan-in. If it has fan-in, then we will list its fan-ins in column 6 and afterwards. Here is a example to illustrate the format of the scan version netlist.

0 11 200 2 3 5 7 8

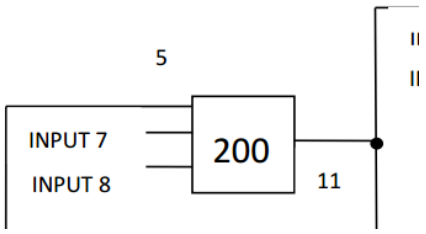


Figure 1 Scan Version Netlist Example

The line number is 11. The type 0 means it is a output of a CLB. The function of the CLB is 200. It has two fan-outs and three fan-ins. The fan-in number are 5,7,8.

The second input ATPG needs is the reduced fault list. The original fault list is processed by fault dominating and fault equivalence. This is also provided by preprocessor. The format of the reduced fault list is like this:

CLB 4 13

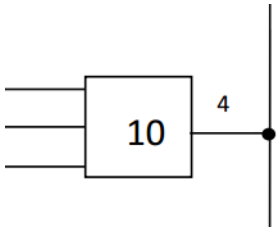


Figure 2 Reduced Fault List Example

It means that the CLB that has the output number 4 now carries out the function number 13 instead of original function 10.

The output of ATPG are two files. One is test_for_faults. Another one is input_vectors. The format of test_for_faults is like this:

CLB 10 25

4 0
0 1 0 1 0 0 0 X 1 1
The first line shows the faulty CLB with the faulty function number. The second line shows the initial state of FF(treated as PPI in ATPG). The third line shows the input vector that can detect this fault.

The format of input_vectors is like this:
4 0
0 1 0 1 0 0 0 1 1 1
It simply changes the X to 0 or 1.

The fault simulator will use this simple version of the test vector.

ATPG will use the fault list and scan version net list to create a output file for each fault. The output file will contain the fault detected and the specific test pattern for that fault. These outputs will be used by logic simulator and fault simulator.

Part II. Algorithm and/or Heuristic Description

1. Error Propagation (EP) and Line Justification (LJ) through Primitive Cubes and Primitive D-cubes

The common logic circuit is usually in gate level. In this project, we are emulating the FPGA system so the circuit is made up of connecting different CLBS together. A 3-input CLB is represented by a 3-variable BSF that has 256 entries. One of these entries represents the fault-free circuit. If there is a fault in the CLB, the truth table is changed to one of the remaining 255 possible BSFs.

For example, the following are 2 BSFs of a CLB with input 1,2,3 and output 4.

000	001	010	011	100	101	110	111	Represented in Decimal number
0	0	0	1	1	0	0	0	24 (fault free)
0	0	0	0	1	1	0	0	12

Figure 3 BSFs of a faulty CLB and a fault-free CLB

Without fault, the truth table of CLB is 24. If there is a certain fault in the CLB, the truth table is changed to 12. Here the integer number is obtained by adding 8 columns of a BSF by weight.

The most important thing we need to do in ATPG is error propagation and line justification. The ATPG algorithm described in the text book is based on primitive gates such as AND, NOR. For primitive gates , we can use controlling value and inversion value to do error propagation and line justification.

Take AND gate with controlling value C=0 and inversion I=0 as an example. VAL is the value at the output.

```
INVAL=VAL XOR 0
if (inval=1)
      then for every input j
      Justify (j, INVAL)
else
     select on input j of gate K
 Justify (j, INVAL)
```

Error propagation heuristic for a AND gate:

```
Propagate (L, error)
    for every input j of gate K other than L
            Justify (j, 1)
Propagate (K, error ⊕0)
```

But in CLB, we need to use primitive cube and primitive d cube to do the error propagation and line justification.

Basically there are three steps need to be done: fault activation, line justification and fault propagation.

(i)Fault Activation

Use the example of Fig 1. By comparing the output, we can see that the test pattern 011 and 101 can detect the fault. So that we can construct a primitive D-cube as following.

1	2	3	4
0	1	1	D
1	0	1	D'

Figure 4 Primitive D-Cube

In order to activate the fault 12, we can look up this table and choose any one of the two input patterns so that an error will appear at the output of that CLB.

The way to construct this cube is first by doing XOR of the outputs of the fault-free and faulty circuit under every same input patterns to find those input patterns that will make the output different. Then combine those input patterns that give the same correct and faulty output.

(ii) Line Justification

Based on the BSF 12, we can construct the corresponding primitive cubes as following.

1	2	3	4
0	X	X	0
1	0	X	1
1	1	X	0

Figure 5 Primitive Cube

For example, if we want to justify the output to be 1, we must select the test pattern 10X. The way to construct this table is similar to the way of constructing primitive D-cube. Namely by doing XOR of the outputs and combine the input patterns that have the same output together.

(iii) Error Propagation

For error propagation, we need a different type of primitive D-cube as in figure 2. The primitive D-cube is based on one fault-free circuit and one faulty circuit. Here we need a primitive D-cube that is only based on one faulty circuit. It can be constructed as following(using BSF12).

1	2	3	4
D	0	0	D
1	D'	1	D

Figure 6 Another Type of Primitive D-Cube

2. Backtracking

There are many decisions we can make in error propagation and line justification. We may choose one way to do them. It is possible that the decision we select will lead to an inconsistency detected by implication. If this happens, we need to restore the netlist to the status before the decision is made.

The execution of the backtracking can be visualized with the aid of a decision tree, as shown in the following

3. Imply and Check

For a node N with a pending value, the procedure of implication carries two main functions. The first one is to check for consistency with the old value. If the pending value is consistent with the old value, then assign the pending value.

The second one is to compute the values that once the value of node N is determined, they can also be uniquely determined. But those values are not assigned immediately. They are put in a assignment queue instead.

The general procedure of imply and check is that every time I pick an item from the assignment queue and do the consistency check. If the check is passed, implication is executed and new nodes with pending values are added into the assignment queue. The whole procedure ends when the assignment queue is empty.

4. D Frontier

The D frontier consists of all CLBs whose output value is currently X but have one or more error signals on their inputs. Error propagate will select one CLB from the D frontier and propagate D or D_bar by looking up the primitive D cube. Every time is D-alg is called, the D frontier will be updated.

5. J Frontier

The J frontier consists of all CLBs whose output is known but is not implied by its input values. Line justification will select one CLB from J frontier and set the input by looking up the primitive cube. Every time is D-alg is called, the D frontier will be updated.

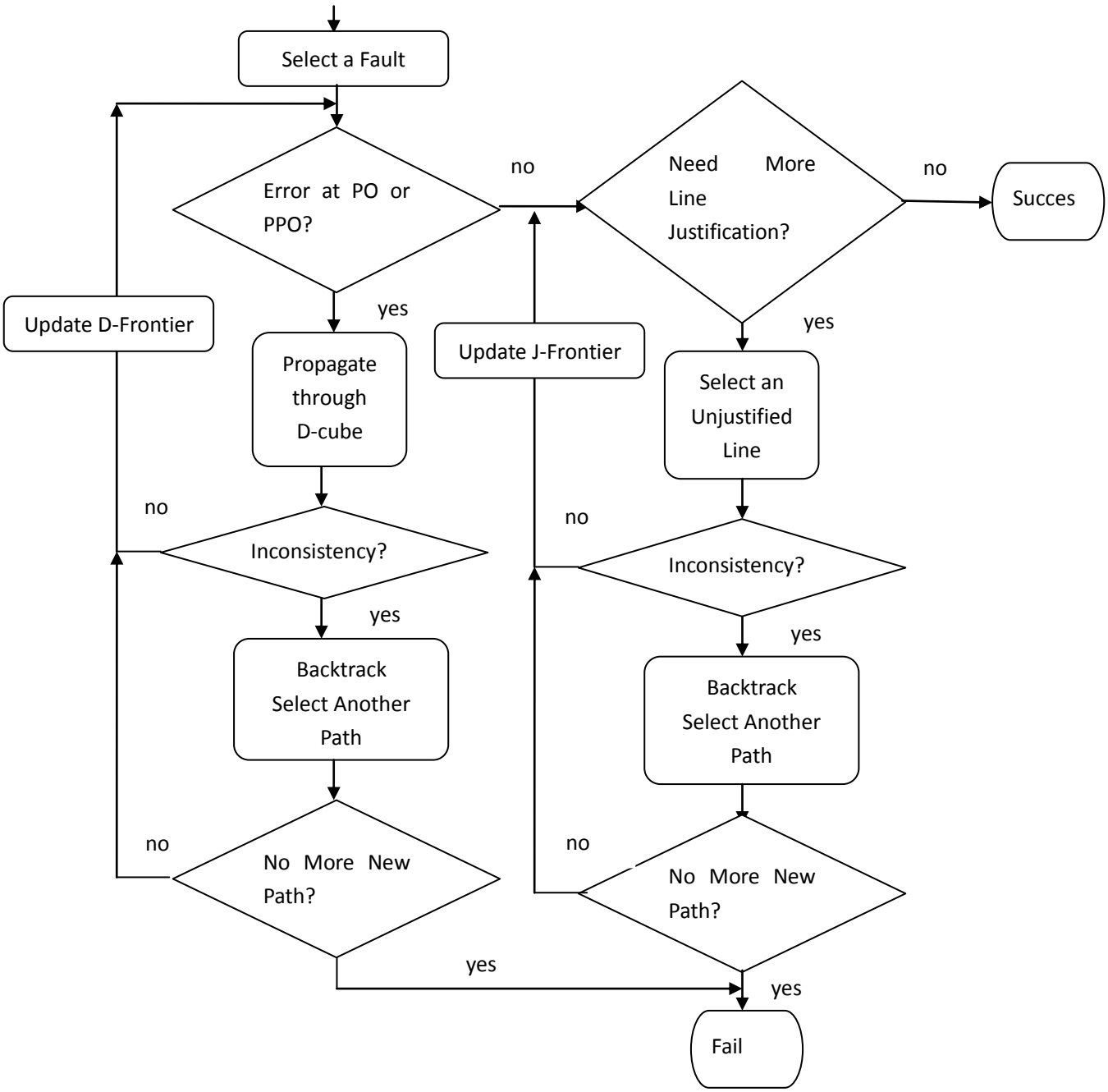


Figure 7 Flowchart of the D-alg

Part III. Pseudo Code

1.core code of D-algorithm

0=failure 1=success

```
D-algorithm()
    if (!imply_and_check()) = then return 0
    d_frontier= update_d_frontier()
    if (!error_at_PO_PPO()) then
```

```
        begin
           if(size of d_frontier==0) then return 0
           // if d_frontier is not empty
           select one untried CLB from D-frontier
           d-drive the error to the output of that CLB by looking up the D cube
           if D-algorithm() =1 then return 1
           if (number of untried CLB in the D-frontier==0)    then return 0
        end
//error propagates to one of POs
j_frontier= update_j_frontier()
if(size of j_frontier==0) then return 1
select an untried CLB from the set of J-frontier
begin
    if(output of CLB==0)
        select one untried way to set the input by looking up primitive cube that has 0 as output
    if(output of CLB==1)
        select one untried way to set the input by looking up primitive cube that has 1 as output
    if D-algorithm() =1 then return 1
end
return 0
```

---

2. Imply and Check

```
imply_and_check(node, value, direction){
          if(current_value !=X && current_value!=value_to_be_assigned) then output
"inconsistency"
          else set the node value
           if( type of the node is PI or PPI){
                 if(direction==forward){
                    if(number of fan-outs>1){
                       for (every downnode)          add_to_assignment_queue (node->downnode,
value, forward)
                    }
                    if(number of fan-outs<=1) {
                        temp=evaluate(node->downnode)
                        if (temp!=X)
                        add_to_assignment_queue (node->downnode, temp, forward)
                    }
                 }
           }
           if (type of the node is FB, PO, PPO, GATE){
                if (type of the node is FB){
                    if (direction==forward){
                        temp=evaluate(node->downnode)
```

```
                    if (temp!=X)
                    add_to_assignment_queue (node->downnode, temp, forward)
                }
                if (direction==backward){
                add_to_assignment_queue (node->upnode, value, backward)
                for (all node->upnode->downnode) add_to_assignment_queue
(node->upnode->downnode, value, forward)

                }
            }
            if (type of the node is GATE, PPO, PO){
                if (direction==forward){
                    if (number of fan-outs>1){
                    for(every downnode) add_to_assignment_queue (node->downnode,
value, forward)
                    }
                    if (number of fan-outs<=1){
                        if (number of fan-outs=1){
                            temp=evaluate(node->downnode)
                            if (temp!=X)
                            add_to_assignment_queue (node->downnode, temp, forward)
                        }
                    }
                }
                if (direction==backward){
                    if (value==0){
                        for (every fan-in of node) add_to_assignment_queue
(node->downnode, value that will make output 0, backward)
                    }
                     if (value==1){
                         for (every fan-in of node) add_to_assignment_queue
(node->downnode, value that will make output 1, backward)
                    }
                     if (value==X){
                         for (every fan-in of node) add_to_assignment_queue
(node->downnode, X, backward)
                    }
                  }
                 }
          }
}
```

---

3. The procedure to update the assignment queue

```
add_to_assignment_queue (node, value, direction)
  if (there is an element N add to the assignment queue) {
        create a new node pointer ap in the linked list
        give properties of N  to ap
        if (assignment_head==NULL && assignment_tail==null){
            assignment_head=ap
            assignment_tail=ap
        }
        else{
        assignment_tail->next = ap
        assignment_tail=ap
        }
    }
    if (doing pop operation){
        if(assignment_head==NULL && assignment_tail==null){
             return NULL;
        }
        if (assignHead==assignTail){
            temp=assign_head
            assign_head = NULL
            assign_tail = NULL
            return temp;
        }
        temp = assign_head
        assign_head = assign_head->next
        return temp

    }
}
```

Part IV. Data Structures and Functions

1. Data Structure

The first data structure I will use is a self defined structure that contains all the information of a node. The information includes line number, type, value, number of fan-in, number of fan-out, pointer to the up nodes, pointer to the down nodes and level of the node.

The second data structure I will use is the linked list. The linked list is used for assignment queue. Every node in the linked list is a self-defined structure that contains the node number, value, direction and a link to the next node. Every time it calls the D-alg and if the assignment queue is not empty, a node will pop out. Every time there is a new pending assignment, a node is pushed in. By moving the pointer we can modify the assignment queue.

The third specific data structure I will use is primitive cube and primitive D-cube. They are represented in 3-dimensional array. By addressing the three inputs, we can get the corresponding

output.

Those are the three main unique data structures for this project.

2. Functions

imply_and_check()

This function picks up the head node of the assignment queue. It first checks the consistency of the pending value with the old value. If the check is passed, then the pending value is assigned and nodes with values that can be determined by the implication will be added to the assignment queue.

error_at_PO_PPO()

This function is used to verify if the error has been propagated to the primary output (PO) or pseudo primary output(PPO). If it returns 1, it means that there is an error at PO or PPO so we can jump over the d-drive process.

find_D_D_bar(*np)

This function is used to find if the CLB that np points to has D or D_bar in its input. If it does, it returns 1. The returning value will be used in update_dfrontier() .

update_d_frontier()

Every time the D-algorithm is called, the d-frontier will be updated. The process is to traverse all the nodes and find CLBs that have X at the output and D or D_bar at the input and add them into the d-frontier array. This function will call the find_D_D_bar.

update_j_frontier()

Every time the D-algorithm is called, the j-frontier will be updated. The process is to traverse all the CLBs. For each CLB, we can find the value on its 3 inputs and use the input pattern to look up the primitive cube to see if the output value equals to the real output value. If it doesn't, this CLB will be added to the j-frontier array.

evaluate(node)

This function evaluates if the value of the output of a CLB can be determined by a input pattern. This function is used in the imply process when we need to know if the output of CLB is specified with the current input.

add_to_assignment_queue (node, value, direction)

A new node with a pending value will be added to the assignment queue by creating a new element and make it the tail of the assignment queue.

pop()

If there is at least one element in the assignment queue, then the head element will be poped out. The next element of head element will be made as the head element.

Fault Simulation [Fangwei Situ]

Part I. Inputs and Outputs

input list:

1. The original circuit: netlist.i

2. A list of faults, already reduced after fault equivalence and fault dominance(from preprocessor): fault_list.txt

3. A sequence of test patterns in binary format(from ATPG): input_vectors.txt

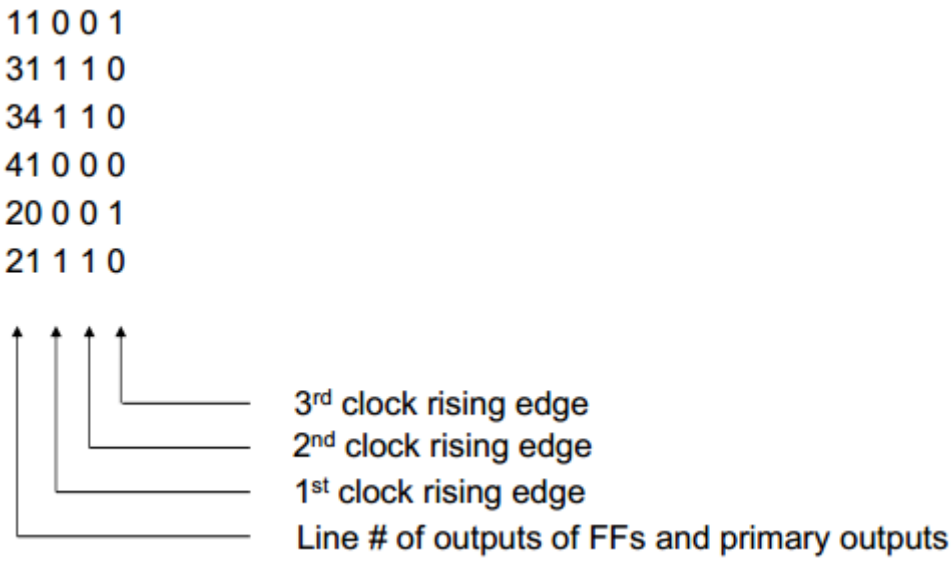
4. (additional) A list of all signal lines with their outline number and level：logic_level.txt. This file is produced by the preprocessor. Below is an example which shows the format of the file:

```
1 0
2 0
3 0
11 2
4 1
5 1
6 1
7 1
8 1
9 1
10 2
```

The first column(1, 2 ,3, 11...) is the outline number of a signal line; the second is the level of a signal line.

Why levels are needed for fault simulator and how they are utilized will be further discussed in the algorithm part.

5. the state of all flip-flops after each clock edge and the state of all outputs before each clock edge: PO_FF_state.txt. This file is produced by logic simulator and used for comparison of fault free simulation with logic simulation. The format is fixed and given in class. Below shows its format.



output list:

1. First 5 output patterns for each fault -no fault dropping: five_output_patterns.txt
2. Faults that are detected for each clock cycle and at which PO's they are detected, and at which PO's and/or PPO's, no fault dropping: detection_report.txt
3. Compare the PO values for fault free circuit with those from the logic simulator, and report differences: differences_with_logic_simulator.txt
4. results showing runtime as a function of degree of fault: runtime.txt. Below is an example which shows the format of the file:

```
2 0.000201s
3 0.000234s
4 0.000311s
5 0.000397s
6 0.000512s
7 0.000691s
```

The first column(2, 3, 4...) is the degree of fault dropping, or after how many times detected a fault is dropped; the second is the runtime of the program. How runtime of this program is measured will be further discussed in the algorithm part.

function of this program:

1. Create truth tables to store the mappings of all CLBs.
2. Read in all input files, namely netlist.txt, fault_list.txt ,and test_vector.txt.
3. Create all the 32 circuit for simulation. Do fault injection by combining information in fault list and netlist.
4. Simulate 1 good circuits and 31 faulty circuits in parallel. Drop faults when appropriate. Save the value of each FFs' input for the next time simulation.
5. Save information for the five_output.txt. Also compare the result of faulty circuits from the result of fault-free circuit. If the result is different, that means this fault is detected. If not, means

this fault is undetected. Save the information in differences_with_logic_simulator.txt and detection_report.txt.

Part II. Algorithm and/or Heuristic Description

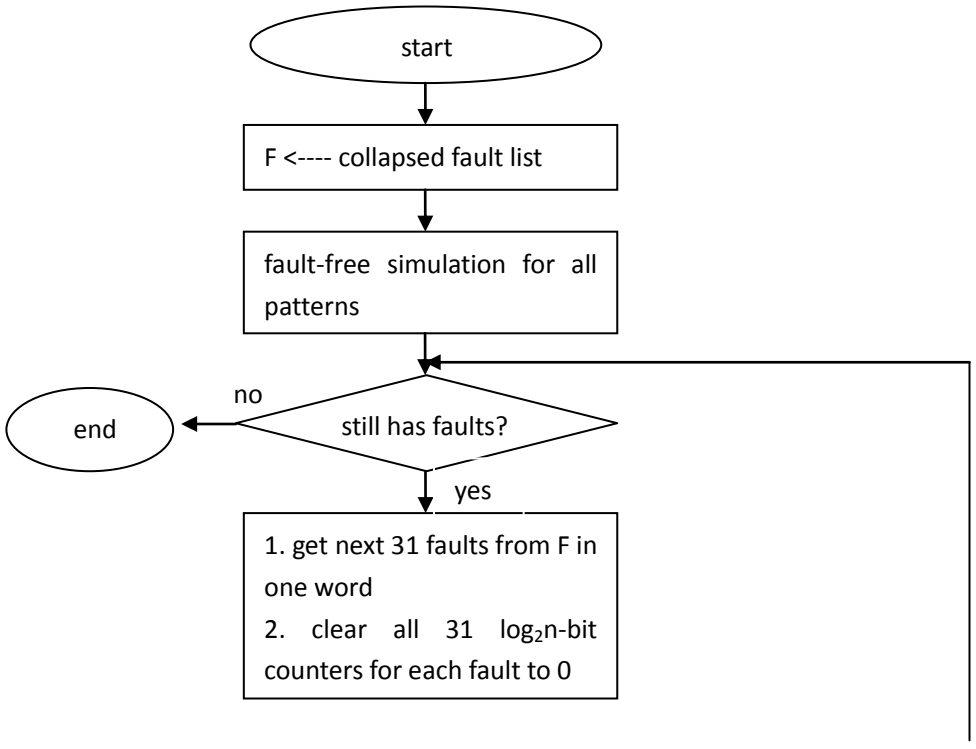
1. Parallel Fault simulation

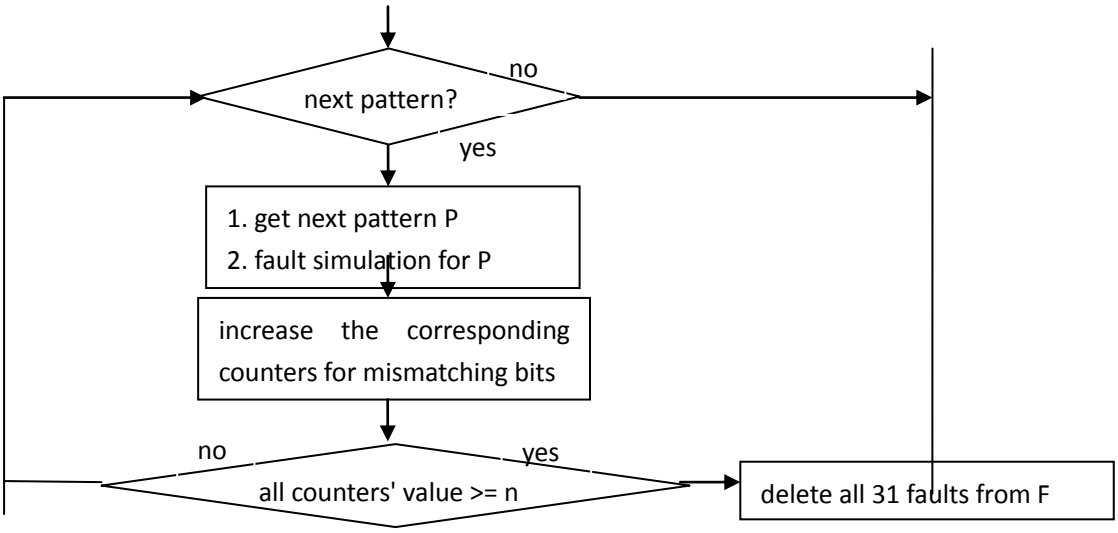
1.1 Comparison with Serial Fault Simulation and Parallel Pattern Fault Simulation

Compared to Serial Fault Simulation, which simulates one faulty circuit with one test pattern at one time and repeats for each fault of interest, Parallel Fault Simulation(PFS) can do it more efficiently by simulating a number of word_size(denoted as W) inputs in one simulation run. Parallel Simulation can be either parallel in faults or parallel in pattern s. Parallel Pattern Fault Simulation means there are W test patterns simulated in parallel for one circuit; while Parallel Fault Simulation simulates (W - 1) circuits and one good circuit in parallel for one test pattern. Parallel Fault Simulation(PFS) will be used here.

1.2 The flow of Parallel Fault simulation in textbook

The word_size in this project is 32. The lowest bit out of all 32, namely bit0, is used for fault free circuit, and the rest 31 bits are for 31 different faulty circuits. A faulty circuit is obtained by injecting a fault into an original fault-free circuit. It is done through masking, which is a Boolean operation of having a word of a signal line do 'and' and 'or' bitwisely with mask vectors. After circuit construction, fault simulation starts and after all the bitwise logical operation, the final value of words of outputs become the results of simulation. If any of the higher 31 bits are different from the lowest bit, then the faults of the corresponding bits are detected because they don't have the same outputs as the good circuit. And for those bits having the same results, those faults are not detected.





flowchart of parallel fault simulation

part 3 flow of the program

cread function(read the netlist)

```
182             if(gt == 256) NFF++;//if it's a FF
183           }
184       }
185       tbl = (int *) malloc(++ntbl * sizeof(int));
186
187       fseek(fd, 0L, 0);
188       i = 0;
189       while(fgets(buf, MAXLINE, fd) != NULL) {
190           if(sscanf(buf,"%d %d", &tp, &nd) == 2) tbl[nd] = i++;
191       }
192       allocate();
193
194       fseek(fd, 0L, 0);
195       while(fscanf(fd, "%d %d", &tp, &nd) != EOF) {
196           np = &Node[tbl[nd]];
197           np->num = nd;
198           if(tp == PI) Pinput[ni++] = np;
199           else if(tp == PO) Poutput[no++] = np;
200           switch(tp) {
201               case PI:
202               case PO:
203               case GATE:
204                   fscanf(fd, "%d %d %d", &np->type[0], &np->fout, &np->fin);
205                   for(z=1;z<32;z++){
206                       np->type[z] = np->type[0];
207                   }
208
209                   if(np->type[0] == 256){
210                       FF[ffcnt++] = np;
211                   }
212                   break;
213
                      --
```

readLV function(read the levels)

```
readLV(char *cp)//read logic level.
{   char buf[MAXLINE];
    int ntbl, *tbl, i, j, k, nd, tp, fo, fi, ni = 0, no = 0;
    FILE *fd;
    NSTRUC *np;

//   if((fd = fopen("Logic_Level.txt","r")) == NULL) {
//    printf("File Logic_Level.txt does not exist!\n");
//    return;
//   }
  sscanf(cp, "%s", buf);
  if((fd = fopen(buf,"r")) == NULL) {
      printf("File %s does not exist!\n", buf);
      return;
  }


   lvlMax = 0;
   while(fgets(buf, MAXLINE, fd) != NULL) {
       if(sscanf(buf,"%d %d", &tp, &nd) == 2) {//tp is outline num; nd is level of node!!
           int x;
           for(x=0; x < Nnodes; x++){
               np = &Node[x];
               if (np->num == tp){
                   np->level = nd;
               //printf("point reached; tp = %d, nd = %d, lvlMax = %d \n", tp, nd, lvlMax); all correct!!!
                   if(nd > lvlMax){
                       lvlMax = nd;
                   }
               }
           }
       }
```

readFL function 1(read the fault list)

```
readFL(char *cp)//read fault lists
{   char buf[MAXLINE];
    int ntbl, *tbl, i, j, k, nd, tp, fo, fi, ni = 0, no = 0;
    FILE *fd;
    NSTRUC *np;

//   if((fd = fopen("fault_list.txt","r")) == NULL) {
//    printf("File fault_list.txt does not exist!\n");
//    return;
//   }
  sscanf(cp, "%s", buf);
  if((fd = fopen(buf,"r")) == NULL) {
      printf("File %s does not exist!\n", buf);
      return;
  }


   int cnt = 0;//fault counter
   while(fgets(buf, MAXLINE, fd) != NULL) {//get one line each time and one fault for each line
       cnt++;
   }
   faultNum = cnt;
   faultList = (int *) malloc(3 * cnt * sizeof(int));
   //A mapping of key(outline numbers of faulty CLBs) to value(BSFs of faulty CLBs after fault injection) is created implemented by a 1-dimensional dynamic ar
   //For example, for 1st fault, namely CLB 4 13: faultList[0] = 4; faultList[1] = 13;
   //For 2nd fault, namely CLB 11 201: faultList[3] = 11; faultList[4] = 201;...
   //and faultList[2] faultList[5] are counter to count how many times a fault has been detected!!!
   cnt = 0;
   fseek(fd, 0L, 0);
   while(fgets(buf, MAXLINE, fd) != NULL) {
       cnt++;
       if(sscanf(buf,"%*s %d %d", &tp, &nd) == 2) {//skips the string of "CLB"; tp is outline num; nd is the faulty BSF!!
```

readFL function 2

```
540     //A mapping of key(outline numbers of faulty CLBs) t
541     //For example, for 1st fault, namely CLB 4 13: faultL
542     //For 2nd fault, namely CLB 11 201: faultList[3] = 1
543     //and faultList[2] faultList[5] are counter to count
544     cnt = 0;
545     fseek(fd, 0L, 0);
546     while(fgets(buf, MAXLINE, fd) != NULL) {
547         cnt++;
548         if(sscanf(buf,"%*s %d %d", &tp, &nd) == 2) {//sk
549             // int x;
550             // for(x=0; x < Nnodes; x++){
551                 // np = &Node[x];
552                 // if (np->num == tp){
553                     // np->type[cnt] = nd;
554                 // }
555             // }
556             faultList[3*(cnt - 1)] = tp;//see comments al
557             faultList[3*(cnt - 1) + 1] = nd;
558         }
559
560     }
561     for(cnt=0;cnt<faultNum;cnt++){//set the times that a
562      faultList[3*cnt + 2] = 0;
563     }
564 //    int kk;
565 //for(kk=0;kk<2*cnt;kk++){//debug use
566 //     printf("%d ", faultList[kk]);
567 //}
568     fclose(fd);
569     printf("==> FL OK\n");
570 }
571
```

readT function 1 (read the input vectors)

```
440  readT(char *cp)//read test patterns
441  {   char buf[MAXLINE];
442     int ntbl, *tbl, i, j, k, nd, tp, fo, fi, ni = 0, no = 0;
443     FILE *fd;
444     NSTRUC *np;
445
446  //   if((fd = fopen("input_vectors.txt","r")) == NULL) {
447  //    printf("File input_vectors.txt does not exist!\n");
448  //    return;
449  //   }
450    sscanf(cp, "%s", buf);
451    if((fd = fopen(buf,"r")) == NULL) {
452        printf("File %s does not exist!\n", buf);
453        return;
454    }
455
456     int cnt = 0;//row counter;
457     int ffcnt;//count how many FFs it has
458     int picnt = 0;//count how many PIs it has
459
460     while(fgets(buf, MAXLINE, fd) != NULL) {//read one line each time
461     //the value of cnt indicates the row that is being read!!
462         cnt++;//cnt should be 11 at last
463         //printf("cnt is %d\n", cnt);
464     }
465     clkCycle = cnt - 1;//should be 11-1=10 in second test case//the sar
466     testPatterns = (int *) malloc(clkCycle * Npi * sizeof(int));//creat
467     FFValue = (int *) malloc((32* (clkCycle+1) * NFF) *sizeof(int));//I
468     //note that FFValue[0] are used for the 1st sim...;the last set, na
469
470     cnt = 0;
471     ffcnt = 0;
```

readT function 2

```
473
474  for(cnt=0;cnt!=clkCycle+1;){//read one line each time
475  //the value of cnt indicates the row that is being read!!
476      cnt++;
477
478      if(cnt == 1){//read initial state of FFs
479          ///read two integers each time;
480          for(ffcnt=0;ffcnt<NFF;){//tp is the outline# of FF; nd is the FF value!! //assume the FF in file are aligned, hence tp is not neede
481              ffcnt++;
482              fscanf(fd, "%*d %d", &nd);
483              //FFValue[32][clkCycle+1][NFF]
484              //FFValue[0][0][ffcnt - 1]
485              *(FFValue + ffcnt - 1 + 0 * NFF + 0 * (clkCycle+1) * NFF) = nd;//the 1st FF is the 0th element int the FFValue array!!
486              int bit;
487              for(bit=1;bit<32;bit++){
488                  //FFValue[32][clkCycle+1][NFF]
489                  //FFValue[bit][0][ffcnt - 1]; = FFValue[0][0][ffcnt - 1];
490                  *(FFValue + ffcnt - 1 + 0 * NFF + bit * (clkCycle+1) * NFF) = *(FFValue + ffcnt - 1 + 0 * NFF + 0 * (clkCycle+1) * NFF);
491              }
492          }
493      }
494      else{//cnt != 1
495          for(picnt=0;picnt<Npi;){//tp is outline num; nd is the faulty BSF!! //this just read one row of T!! The rest needs to be complete!!
496              picnt++;
497              fscanf(fd, "%d", &nd);
498              //np = *(Pinput + picnt);//the index should be aligned??
499              //np -> nodeValue[0] = nd;
500
501
502              //testPatterns[clkCycle][Npi]
503              //testPatterns[cnt - 1 - 1][picnt - 1] = nd;
504              *(testPatterns + picnt - 1 + (cnt - 1 - 1) *Npi) = nd;
```

fault injection

```
635 simulate(){//do parallel fault simulation
636     int currLvl;//current level
637     int currClkCycle;//current clock cycle, or current test pattern being simulated; starting fr
638     int i, j, x, y, k;
639     unsigned long ain, bin, cin, dout;
640     int bit = 1;//if fault number is greater than 31, bit needs to be reset in the middle of the
641     NSTRUC *np;
642 
643     for(currClkCycle=0; currClkCycle < clkCycle; ){//should be clkCycle instead of clkCycle+1!!!
644     //fault injection:needs to be complete in case numOfFauly is greater than 31!!!
645         if(currClkCycle == 0){
646 
647             //Fault injection!!!
648             for(x=0;(faultCnt<faultNum && (x<31));x++){//replace the old BSF with the faulty one,
649                 for(y=0;y<Nnodes;y++){
650                     np = & Node[y];
651                     if(faultList[3*faultCnt] == np->num){//do mask creation for matching node
652                         //if type is btwn 0~255, convert type(int) to an 8-element array
653                         //compare the corresponding bits
654                         np->type[x+1] = faultList[3*faultCnt+1];
655                     }
656                 }
657                 printf("x is %d; faultCnt is %d\n",x, faultCnt);
658                 faultCnt++;
659             }
660             if(faultCnt == faultNum){
661                 simDone = 1;
662                 //printf("point11 reached; simDone is %d\n", simDone);
663                 //printf("point reached");
664             }
665             //after injection, all node's BSF are up-to-date.
```

create f0-f7

```
664                 }
665                 //after injection, all node's BSF are up-to-date.
666                 //now creates f[0][32], f[1][32], f[2][32], ... , f7[7][32], an
667                 int fArr[8][32];
668                 for(y=0;y<Nnodes;y++){
669                     np = & Node[y];
670                     int currBSF[8];//means the BSF in binary form(8bit:f0-f7) f
671                     int currBit;
672                     int currFi;
673
674                     for(currBit=0;currBit<32;currBit++){
675                         longToArr(np->type[currBit], 8, currBSF);//how come cur
676                         for(currFi=0;currFi<8;currFi++){
677                             fArr[currFi][currBit] = currBSF[currFi];//
678                         }
679                     }
680
681                     // convert f[currFi][currBit] into f[currFi]);
682                     for(currFi=0;currFi<8;currFi++){
683                         for(i=0;i<32;i++){
684                             np->f[currFi] += fArr[currFi][i] * pow(2, i);//i.e.
685                         }
686                     }
687                 }
688
689
690             }
691             currClkCycle++;
692             //fault injection only gets execute once
693
694             initialize(currClkCycle);
695             for(currLvl=0;currLvl < lvlMax+1;currLvl++){
```

Initialize all node-values to -1 and set values of FFs and PIs

```
368 //initialize the whole circuit at the beginning of each cycle, namely set all node values to -1 and set FFs' and PIs' output values.
369 initialize(int currClkCycle){//currClkCycle starting from 1!!!
370     int i, j;
371     NSTRUC *np;
372     for(i=0;i<Nnodes;i++){
373         np = &Node[i];
374         for(j=0;j<32;j++){
375             np->nodeValue[j] = -1;
376         }
377     }
378
379     for(i=0;i<NFF;i++){//update outputs of FFs with values saved in last similation
380         np = FF[i];
381         for(j=0;j<32;j++){
382                 //FFValue[32][clkCycle+1][NFF]
383                 //FFValue[j][currClkCycle-1][i];//
384
385             np->nodeValue[j] = *(FFValue + i + (currClkCycle-1)*NFF + j * (clkCycle+1) * NFF);//element 0 is used for simulation 1.
386         }
387     }
388
389     for(i=0;i<Npi;i++){//update PIs with values stored in testPatterns!
390         np = Pinput[i];
391         for(j=0;j<32;j++){
392             //testPatterns[clkCycle][Npi]
393             //testPatterns[currClkCycle-1][i]
394             np->nodeValue[j] = *(testPatterns + i + (currClkCycle-1) *Npi);//element 0 is used for simulation 1.
395             //printf("Npi %d, bit %d is value: %d\n", i, j, np->nodeValue[j]);//its correct here!!!
396         }
397         //printf("currClkCycle is %d; PI(%d) is %d ", currClkCycle, i, *(testPatterns + i + (currClkCycle-1) *Npi));
398     }
399     //printf("\n");
```

Evaluation 1

```
694         initialize(currClkCycle);
695         for(currLvl=0;currLvl < lvlMax+1;currLvl++){
696 
697             //printf("point reached; currLvl = %d, lvlMax = %d \n", currLvl, lvlMax);//deubg use
698 
699             for(i=0; i<Nnodes; i++){
700                 np = &Node[i];
701                 if(np->level == currLvl){
702                     //if type is Branch, copy the up_node's value
703                     if (np->type[0] == 260) {
704                         for (j=0; j<32; j++) {
705                             np->nodeValue[j] = np->unodes[0]->nodeValue[j];
706                         }
707                             //Check if this is a CLB in our truth table for 0~255
708                     }
709                     else if (np->type[0]>=0 && np->type[0]<=255) {
710 
711                         //nodeValue Arr to long
712                         ain = arrToLong(np->unodes[0]->nodeValue, 32);
713                         bin = arrToLong(np->unodes[1]->nodeValue, 32);
714                         cin = arrToLong(np->unodes[2]->nodeValue, 32);
```

Evaluation 2

```
716                                 dout =
717                                     np->f[0] & ~ain & ~bin & ~cin |
718                                     //bit1
719                                     np->f[1] & ~ain & ~bin & cin |
720                                     //bit2
721                                     np->f[2] & ~ain & bin & ~cin |
722                                     //bit3
723                                     np->f[3] & ~ain & bin & cin |
724                                     //bit4
725                                     np->f[4] & ain & ~bin & ~cin |
726                                     //bit5
727                                     np->f[5] & ain & ~bin & cin |
728                                     //bit6
729                                     np->f[6] & ain & bin & ~cin |
730                                     //bit7
731                                     np->f[7] & ain & bin & cin;
732                                 //nodeValue unsigned long to Arr
733 //                                printf("point reached; currClkCycle is %d\n", (
734 //                                printf("point reached; currLvl = %d, lvlMax = %
735 //                                printf("%x %x %x %x\n", ain, bin, cin, dout);
736                                 longToArr(dout, 32, np->nodeValue);
737 //                                int kkkk;
738 //                                for(kkkk=0;kkkk<32;kkkk++){
739 //                                      printf("%d ", np->nodeValue[kkkk]);
740 //                                }
741 //                                printf("\n\n\n");
742                             }
743                         }
744                     }
745                 }
746                 saveData(currClkCycle, faultCnt);//save data of each clock cyc
```

Save data 1

```
590 saveData(int currClkCycle, int faultCnt){//save data for the current clock cycle
591     int i, j;
592     NSTRUC *np;
593     int times = (faultCnt-1) / 31; //if it's 0, then it's the 1st run; if it's 1, then it's the 2nd run...
594 
595     if(times == 0){
596         //output[faultNum+1][Npo][clkCycle]
597         //output[i][j][currClkCycle - 1]
598         for(i=0;i<faultNum+1;i++){
599             for(j=0;j<Npo;j++){
600                 np = Poutput[j];
601                 *(output + currClkCycle-1 + j*clkCycle + i*Npo*clkCycle) = np->nodeValue[i];//note that eleme
602             }
603         }
604 
605 
606     }
607     else{//(times >= 1)
608         //output[faultNum+1][Npo][clkCycle]
609         //output[i+times*31][j][currClkCycle - 1]
610         for(i=1;i<faultNum-31+1;i++){// i starts from 1, because we don;t save the result of fault free sim a
611             //say there are 40 faults. The 1st sim save results for 31 faults.
612             //This time we save bit1 to bit9 for the rest 9 faults, namely from i=1 to i=9(i<40-31+1=10)
613             for(j=0;j<Npo;j++){
614                 np = Poutput[j];
615                 *(output + currClkCycle-1 + j*clkCycle + (i+times*31)*Npo*clkCycle) = np->nodeValue[i];//note
616             }
617         }
618     }
```

Save data 2

```
    for(i=0;i<32;i++){//save results for FF
        int ffcnt;
        for(ffcnt=0;ffcnt<NFF;){
            ffcnt++;
            np = FF[ffcnt-1];
            //FFValue[32][clkCycle+1][NFF];
            //FFValue[i][currClkCycle][ffcnt-1];
            *(FFValue + ffcnt - 1 + currClkCycle * NFF + i * (clkCycle+1) * NFF) = np->unodes[0]->nodeValue[i];/
        }
    }
}
```

Produce five output patterns file

```
generateFOP(){//five_output_patterns.txt
    int i, j, k;
    NSTRUC *np;
    FILE *fd;
    fd = fopen("MY_five_output_patterns.txt", "w");
    for(i=0;(i<faultNum+1) && (i<31);i++){//i coulde be larger than 31 here!!
        if(i == 0){
            fprintf(fd, "FF\n");
        }
        else{
            fprintf(fd, "CLB %d %d\n", faultList[3*(i-1)], faultList[3*(i-1)+1]);
        }
        for(j=0;j<Npo;j++){
            np = Poutput[j];
            int currTest;
            //output[32][Npo][clkCycle]
            //output[i][j][currTest]
            //dont need to care about faultCnt here.
            fprintf(fd, "%d ", np->num);//five_output_patterns
            for(currTest = 1; currTest < 5+1; currTest++){
                fprintf(fd, "%d ", *(output + currTest + j*clkCycle + i*Npo*clkCycle));//five_output_patterns
                if(currTest == 5){
                    fprintf(fd, "\n");
                }
            }
        }
    }
    fclose(fd);
}
```

Produce detection report

```
799 generateDR(){//detection_report.txt
800     int i, j, k, t;
801     FILE *fd;
802     NSTRUC *np;
803     fd = fopen("MY_detection_report.txt", "w");
804         for(k=1;k<clkCycle+1;k++){//for test pattern k
805             fprintf(fd, "pattern %d\n", k);
806 
807             //printf(fd, "pattern %d\n", k);
808 
809             for(i=1;i<faultNum+1;i++){//starting from the 1st fault
810                 t = 0;//how many times the current fault has been detected;
811                 for(j=0;j<Npo;j++){
812                     np = Poutput[j];
813                     //output[faultNum+1][Npo][clkCycle]
814                     //output[i][j][k - 1]) != output[0][j][k - 1]
815 
816                     //i coulde be larger than 31
817                     if(*(output + (k-1) + j*clkCycle + i*Npo*clkCycle) != *(output + (k-1) + j*clkCycle + 0*Npo*clkCycle)){
818                         faultList[3*(i-1)+2]++;
819                         if(faultList[3*(i-1)+2] < 10){//if a fault has been detected 10 times, the we drop the fault!!
820                             if(t == 0){
821                                 fprintf(fd, "CLB %d %d %d ", faultList[3*(i-1)], faultList[3*(i-1)+1], np->num);
822                             }
823                             else{
824                                 fprintf(fd, "%d", np->num);
825                             }
826                             t++;//if t is not 0, it means that this fault has been detected at other POs.
827                         }
828                     }
829                 }
830                 if(t != 0){//if t == 0, then nothing has been printed, hence no need to print newline!
```

Diagnosis [Cheng Ma]

1. Introduction

The goals of fault diagnosis are to ascertain whether faults are present (fault detection) in the CUT and to identify them (fault location). Fault detection can be done by comparing logic simulation results with fault simulation results, and fault location is usually performed with the aid of a fault dictionary which is usually exhaustive. But here I'm using a non-exhaustive method to locate the fault of CLB as described below.

2. The non-exhaustive method

For the dictionary-based method, the dictionary corresponding to a input vector is a full dictionary (where we do simulation for fault 0 to fault 255 for every CLB) but actually it is not necessary and this is where we can improve. A CLB is fully represented by its Boolean Switching Function. What we all have to do is to determine the eight rows of a CLB (or as many as we can determine).

Imagine a CLB with three PIs (say 001) as inputs, for a certain test vector T1, we are using only one row of its BSF (because PI has no fault and inputs will be fixed 001 for this T1). So we don't have to insert and simulate fault 0 to 255 for this CLB.

For a more general case, more than one input is from outputs of other CLBs, say inputs are 111. Since our project assumption is that we only have one faulty CLB, so this inputs might be 011, 101 or 110, if inputs are from three different outputs (if two or three of inputs come from the same branch, this input vector could be any value from 000 to 111). Actually we can focus on the output of a CLB and this way we don't even have to worry about branches. A fault is the output of a CLB going from 0 to 1 or from 1 to 0. With the one faulty CLB assumption, if we assume the output is wrong then inputs of this CLB (outputs of its upstream CLBs) must be right and fixed for a given test input vector. This brings much convenience. I implement the method with this conclusion and two lemmas below.

Lemma 1: Two same circuits with same inputs, will have same outputs.

Lemma2: Two circuits with same inputs get same outputs, they might and might not be the same circuits.

So for a test vector from ATE, I do logic simulation and compare with ATE response,

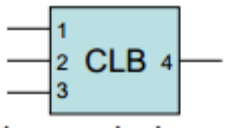
CASE1:

if they match, this CUT might be correct and might not (lemma2). This will help us find faults, if exist, that cannot propagate to the POs (because the POs match logic simulation results). To find them, we flip one CLB output at a time and do fault simulation, if ATE response and fault simulation results (1) match, then this insertion is a possible fault because it does not cause any conflicts; (2)mismatch, then this insertion is a impossible fault on this CUT(contrapositive of lemma1). This actually determines only one row (say Row1)of a CLB BSF(at the beginning

the 8 rows of a CLB in this CUT are all unknown for us). So any fault that has the same Row1 is possible fault. It's clear that there will be 128 (half of 256) possible faults for this CLB after this vector testing. We actually can just record the BSF rows or bits.

CASE2:

if they mismatch, then we can narrow down the faulty area: cone insertion. And CLBs in the intersection will be targets to flip the outputs. The rest is the same as CASE1.



Gate-type 0

1	2	3	4
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Gate-type 1

1	2	3	4
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Gate-type 2

1	2	3	4
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Gate-type 3

1	2	3	4
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

....

Gate-type 255

1	2	3	4
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

3. Steps of the code.

1. Read netlist.txt
2. Levelize (logic simulation is level by level)
3. Read input_output.txt
4. Logic simulation and compare with ATE
5. Cone insertion (if PO mismatches)
6. Fault injection
7. Fault simulation (only simulate the down cone of this fault, time reduced)
8. Compare faulty outputs with ATE, store and print results.
9. Flip the fault back and simulate down cone (to reset to logic simulation results)
10. Do step 4 to step 9 for every test vector in the input_output.txt

4. Some results

For test case 2:

input_output.txt which I'm using(deleted *** here):

3 1

010111001--00

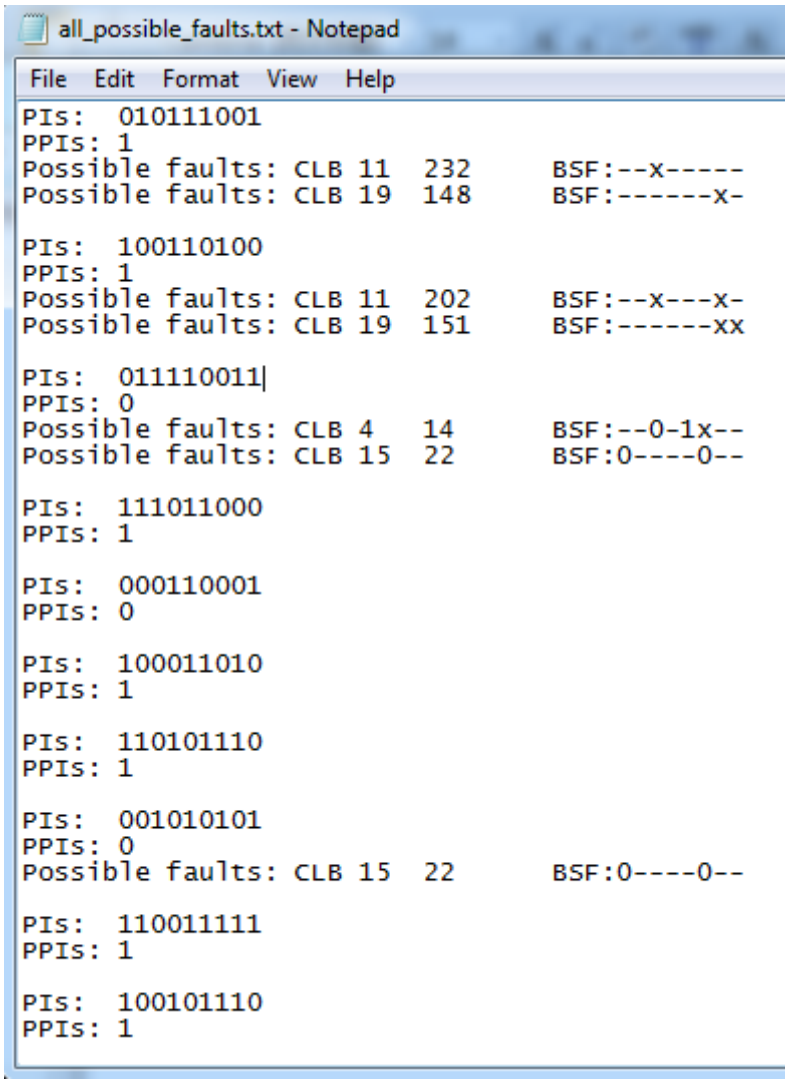
***

3 1
100110100--01
***
3 0
011110011--01
***
3 1
111011000--10
***
3 0
000110001--10
***
3 1
100011010--11
***
3 1
110101110--11
***
3 0
001010101--01
***
3 1
110011111--11
***
3 1
100101110--11
***

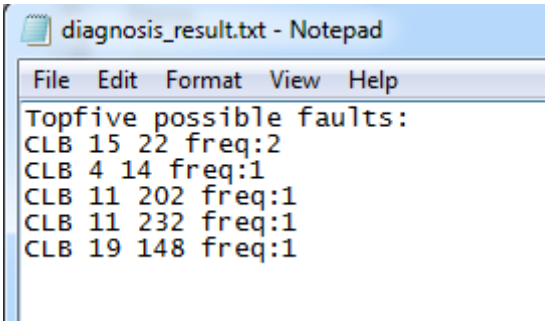
‘x’:likely fault;

‘-’: not tested;

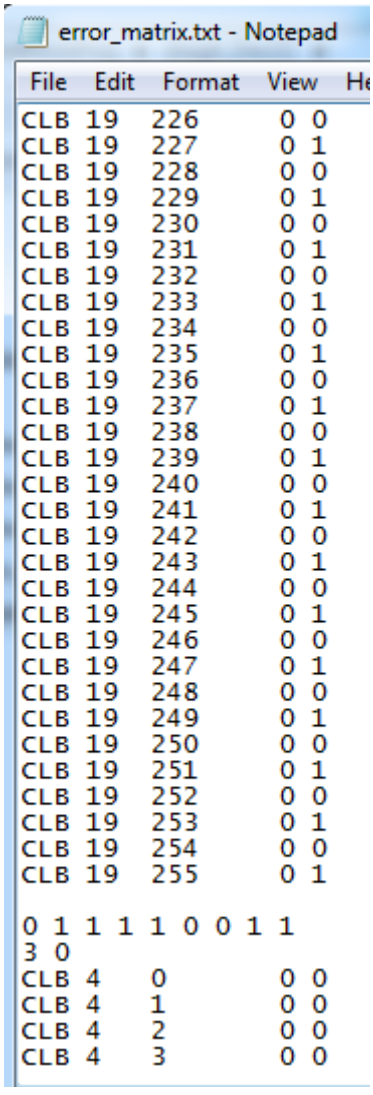
'0' or '1': as it is in the fault free CLB



Top five possible faults(ranked by the frequency that it gets tested):



I still produced the full dictionary for reference.



Integration

```
set folder1=preprocessor
set folder2=logic_sim
set folder3=PFS
set folder4=diagnose
cd .\%folder1%
rmdir /s /q .\%folder1%_result
mkdir .\%folder1%_result
preprocessor.exe < preprocessor_command.txt
move /Y scan_version_netlist.txt .\%folder1%_result
move /Y equivalence_and_dominance_relationship.txt .\%folder1%_result
move /Y filtered_fault_list.txt .\%folder1%_result
copy .\logic_level.txt ..\%folder3%\logic_level.txt
move /Y logic_level.txt .\%folder1%_result
echo Successfully running preprocessor!!!
cd ..\%folder2%
rmdir /s /q .\%folder2%_result
mkdir .\%folder2%_result
logic_sim < logic_sim_command.txt
move /Y PO_FF_state.txt .\%folder2%_result
move /Y report_on_glitches.txt .\%folder2%_result
move /Y waveform.txt .\%folder2%_result
echo Successfully running logic_sim!!!
cd ..\%folder3%
rmdir /s /q .\%folder3%_result
mkdir .\%folder3%_result
HW3 < command.txt
move /Y MY_five_output_patterns.txt .\%folder3%_result
move /Y MY_detection_report.txt .\%folder3%_result
echo Successfully running PFS!!!
cd ..\%folder4%
rmdir /s /q .\%folder4%_result
mkdir .\%folder4%_result
Diagnosis_ChengMa < cmd.txt
move /Y error_matrix.txt .\%folder4%_result
move /Y diagnosis_result.txt .\%folder4%_result
move /Y all_possible_faults.txt .\%folder4%_result
echo Successfully running diagnose!!!
```

The integration is implemented by running all programs (preprocessor, logic simulator, ..., diagnose) one by one using a batch file which is written in windows shell. For each program, the batch file moves all output files to a folder called results and copy files that are needed by other programs to their folders.