

Vlsi Testing - Final Report

group 6

group members: r06943155 楊承濂 r06943086 張奕凡 r06921048 李友岐

1. Problem description:

In this final project, we need to develop an algorithm to detect transition delay fault (a.k.a. TDF), we should also implement N-detect function to enhance the credibility of our test. The performance of each group will be graded with fault coverage, test length, and run time, respectively.

There are some constraints which we need to follow. First, we should implement our TDF test with launch-on-shift method. Second, we cannot use fault dictionary to implement our algorithm because of the unpredictable usage of memory.

Transition delay fault is a kind of fault model, which can illustrate the phenomenon of output signal delay when the input pattern changes. In other words, we need to provide two test patterns (called V1 and V2 there) to detect TDF. V1 activates the fault and then V2 excites the fault and makes sure the fault can be propagated to output.

As noted, the algorithm is required to be implemented with launch-on-shift method, which means that there exists a shift relation between two input patterns.

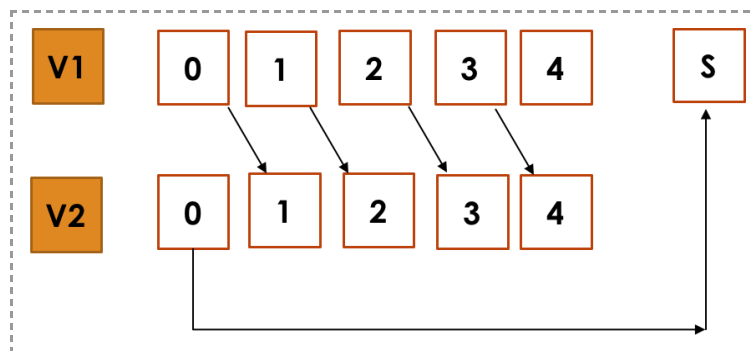


Figure. 1 Diagram of launch-on-shift method

Figure.1 shows how launch-on-shift method works. We can see that V2 is limited by the result of V1 shift right one bit, so we only have 1 free bit to decide once V1 is determined. Therefore, it is crucial to choosing a feasible pattern which meet both V1 and V2 requirements.

Furthermore, we need to use test patterns compression to reduce the numbers of test patterns due to the high correlation with the test cost. There are two techniques which we follow. One is Static Test Compression (a.k.a. STC) and the other is Dynamic Test Compression (a.k.a. DTC). Both will be mentioned in detail in Topic. 3 "Our proposed technique".

2. Past research:

In modern circuits, the working frequency is getting higher and higher. Since some chips can pass the single stuck-at fault test but cannot work normally in high speed, we need a new way to model the fault of delay.

There are two delay models introduced in VLSI Testing course — path delay fault model (PDF) and transition delay fault model (TDF). The PDF model was proposed by G.L.Smith in 1985. The delay defect may

cause the cumulative delay of a path to exceed the timing constraint. This model can detect small delay faults but the number of faults is exponential and the fault is hard to test by ATPG.

The other one is the transition delay fault model. The TDF model was proposed by Z.Barzilai in 1983. There are two types of transition faults — slow-to-rise (STR) and slow-to-fall (STF). We can locate the TDF in a point, so the TDF can correspond to single stuck-at fault. When we scale the time larger, the behavior of STR is like stuck-at 0 , and so is STF to stuck-at 1. Since this corresponding relation and the number of faults is linear to circuit size, it is easier than PDF to test by using ATPG method.

In general, TDF needs two distinct patterns to detect. The first pattern is activation pattern, which appropriate the initial value before transition. The second pattern is propagation pattern, this pattern is just like test pattern of single stuck-at fault that propagates the value after transition to POs. The TDF delay model can only detect large delay fault since the transition signals need to propagate from PIs to POs then we can observe them.

In this project, we apply launch on shift technique to generate test patterns, two patterns are similar that we can combine them together. This can help us to reduce the length of each pattern.

3. Our proposed technique:

(1) Basic flow chart:

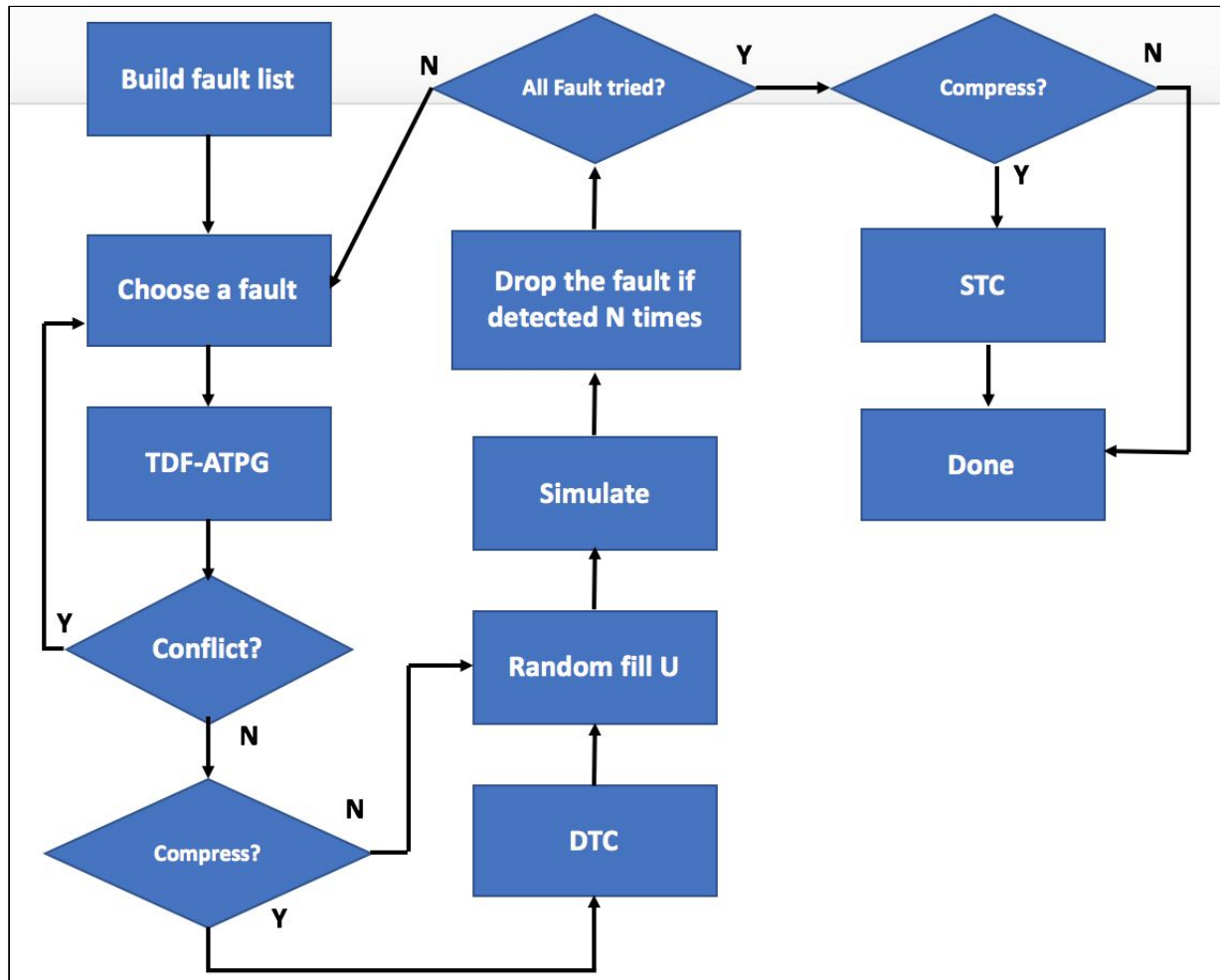


Figure. 2 Flow chart of our algorithm

(2) TDF-ATPG:

TDF-ATPG contains two vectors, v1 is to activate the fault, and v2 is to excite and propagate the fault. For v2 generation, it is just like the SSF-ATPG. Thus, we can modify the original PODEM code. Since we need to generate the test patterns by LOS technique, v1 is heavily related to v2. Therefore, we should consider both v1 and v2 when generating test pattern. There are totally three parts of the test generation, which are v1 activation, v2 excitation, and v2 propagation. Initially, we should do the first and second. To satisfy the condition, if any PI can be implied, we need to imply it first. If it need to make decision, then we do it later in our decision process.

```
/* v1 activation first*/
if (fault->io) {w = fault->node->owire[0]; } // gate output fault, Fig.8.3
else { // gate input fault. Fig. 8.4
    w = fault->node->iwire[fault->index];
}
switch(backward_imply(w,(fault->fault_type ),0)){
    case TRUE:
        find_v1=TRUE;
        sim0();
        break;
    case CONFLICT:
        no_test = TRUE;
        break;
    case FALSE:
        break;
}
```

Figure. 3

To activate the fault in v1, first we find the corresponding wire of fault, then we do backward imply. If the fault is STR, then we assign wire value to 0. If the fault is STF, then we assign wire value to 1. If any PI can be implied, we imply it. If it need to make decision, then we do nothing. After that, assign the PIs of v2 to the value of v1 based on LOS technique like Figure 1.

```
/* set the initial goal, assign the first PI. Fig 7.P1 */
switch (set_uniquely_implied_value(fault)) {
    case TRUE: // if a PI is assigned
        sim(); // Fig 7.3
        if (wfault = fault_evaluate(fault)) forward_imply(wfault); // propagate fault effect
        if (check_test()) find_test = TRUE; // if fault effect reaches P0, done. Fig 7.10
        break;
    case CONFLICT:
        no_test = TRUE; //printf("\nv2:%2d",fault->fault_no); // cannot achieve initial objective, no test
        break;
    case FALSE:
        break; //if no PI is reached, keep on backtracing. Fig 7.A
}
```

Figure. 4

To excite the fault in v2, the good value should be different with faulty value. If the fault is STR, then we assign wire value to 1. If the fault is STF, then we assign wire value to 0. If any PI can be implied, we imply it. If it need to make decision, then we do nothing.

```

wptr
test_possible(fault)
fptr fault;
{
    register nptr n;
    register wptr object_wire,w;
    register int object_level;
    nptr find_propagate_gate();
    wptr find_pi_assignment();
    int trace_unknown_path();

    if (fault->io) {w = fault->node->owire[0]; } // gate output fault, Fig.8.3
    else { // gate input fault. Fig. 8.4
        w = fault->node->iwire[fault->index];
    }

    if (w->value0!=fault->fault_type){
        if (w->value0!=U){return NULL;}
        else {return find_pi_assignment(w,(fault->fault_type ),0);}
    }
}

```

Figure. 5

The decision process is almost the same with origin PODEM code. The only difference is that besides v2, we need to make decision on v1 if it is not implied before. Therefore, we add this part into the function in podem.c: wptr test_possible(fault). We use value0 to represent the wire value of time frame 1. The corresponding wire value in time frame 1 should be equal to the faulty value in time frame 2. If it is not equivalent, we check whether it is an unknown. If so, we make decision to find a PI assignment to activate the fault. After that, assign the PIs of v2 to the value of v1 based on LOS technique like Figure 1. Otherwise, it means the previous assignment make the activation fail, we should backtrack to last decision.

```

store_pattern();
v1_activate(pattern_num,undet_fault );
undet_fault = fault_sim_a_vector(undet_fault,&current_dn);

```

Figure. 6

After generating a test pattern, we store it in the vector and simulate it. To speed up the time, we drop all the detected faults of that pattern.

(3) N-detection:

To achieve the N-detection, we need to modify two parts of original code. One is TDF-Sim and the other one is TDF-ATPG. For the former, we don't drop the fault as soon as it is detected. We drop the fault after it is detected by N times. For the latter, we generate multiple test patterns for one fault. Normally, there are many unknowns in the PI after doing PODEM, and we randomly assign these unknowns to zero or one. If we want to generate multiple patterns, we can randomly fill the origin pattern multiple times.

TDF-Sim:

```
/* the following two loops are both for fault dropping */
/* drop detected faults from the FRONT of the undetected fault list */
while(flist) {
    if (flist->detect == TRUE) {
        flist->detected_times++;
        detect_something=1;
        flist->detect = FALSE;
        if (doing_podem==1){flist->patterns=pattern_num;}
        if (flist->detected_times==ndet){
            flist->detect = TRUE;
            (*num_of_current_detect) += flist->eqv_fault_num;
            f = flist->pnext_undetected;
            flist->pnext_undetected = NULL;
            flist = f;
        }
    }
    else {break;}
}

/* drop detected faults from WITHIN the undetected fault list*/
if (flist) {
    for (f = flist; f->pnext_undetected; f = ftemp) {
        if (f->pnext_undetected->detect == TRUE) {
            f->pnext_undetected->detected_times++;
            detect_something=1;
            f->pnext_undetected->detect = FALSE;
            if (doing_podem==1){f->pnext_undetected->patterns=pattern_num;}
        }
        if (f->pnext_undetected->detected_times==ndet){
            f->pnext_undetected->detect = TRUE;
            (*num_of_current_detect) += f->pnext_undetected->eqv_fault_num;
            f->pnext_undetected = f->pnext_undetected->pnext_undetected;
            ftemp = f;
        }
        else {
            ftemp = f->pnext_undetected;
        }
    }
}
return(flist);
```

Figure. 7: Fault Drop in TDF-Sim

We use a counter to record the detected times of that fault, if the number reaches the given N, then we drop that fault.


```

remember_unknown(){
    int i;
    for (i = 0; i < ncktin; i++) {
        switch (cktin[i]->value) {
            case 0: cktin[i]->valueU=0; break;
            case 1: cktin[i]->valueU=1; break;
            case D: cktin[i]->value = 1; cktin[i]->flag|=CHANGED; cktin[i]->valueU=1; break;
            case B: cktin[i]->value = 0; cktin[i]->flag|=CHANGED; cktin[i]->valueU=0; break;
            case U: cktin[i]->valueU=U; break; // denote it as U
        }
        if (i==ncktin-1){
            switch (sort_wlist[i]->value0) {
                case 0: sort_wlist[i+1]->valueU=0; break;
                case 1: sort_wlist[i+1]->valueU=1; break;
                case D: cktin[i]->value0 = 1; cktin[i]->flag0|=CHANGED; sort_wlist[i+1]->valueU=1; break;
                case B: cktin[i]->value0 = 0; cktin[i]->flag0|=CHANGED; sort_wlist[i+1]->valueU=0; break;
                case U: sort_wlist[i+1]->valueU=U; break; // denote it as U
            }
        }
    }
}

```

Figure 8: Record unknowns in TDF-ATPG

After doing PODEM for a given fault, we have a test cube, then we check which PIs are still unknowns. We use value-U to record it.

```

random_fill_unknown(){
    long rand();
    int i;
    for (i = 0; i < ncktin; i++) {
        switch (cktin[i]->valueU) {
            case 0: cktin[i]->value=0; break;
            case 1: cktin[i]->value=1; break;
            case U: cktin[i]->value = rand()%2; break; // random fill U
        }
        if (i==ncktin-1){
            switch (sort_wlist[i+1]->valueU) {
                case 0: cktin[i]->value0 =0; break;
                case 1: cktin[i]->value0 =1; break;
                case U: cktin[i]->value0 = rand()%2; break; // random fill U
            }
        }
    }
}

```

Figure 9: Random fill U in TDF-ATPG

Next, for each PI, if it is an unknown originally, we give it random value 0 or 1. After all unknowns are filled, we generate a test pattern. Keep doing this then we generate multiple patterns for one fault.

(4) Compression:

STC:

```
if (compress) {  
    int pn=pattern_num+1;  
    while (pn!=pattern_num){  
        pn=pattern_num;  
        undetect_fault = generate_detected_fault_list();  
        for (int i=pattern_num-1;i>=0;i--){  
            v1_activate(i, undetect_fault);  
            undetect_fault = fault_sim_a_vector(undetect_fault,&current_detect_num);  
            total_detect_num += current_detect_num;  
  
            if (detect_something==0){  
                erase_pattern(i);  
            }  
        }  
        random_switch_pattern();  
    }  
}
```

Figure. 10: STC

First, reorder the test patterns randomly. Then, simulate the patterns in new order. Some patterns might be dropped if it can not detect any undetected fault or increase detected times of any fault (ndet). Running this procedure until no pattern is dropped (every pattern is essential).

DTC:

```
if (compress && podemx==1){  
    secondary_fault=fault->pnext_undetect;  
    while (secondary_fault && secondary_fault->detected_times>0){  
        secondary_fault= secondary_fault->pnext_undetect;  
    }  
    if ( secondary_fault){ podem(secondary_fault,current_backtracks, 1);}  
    return TRUE;  
}  
  
for (i = 0; i < ncktwire; i++) {  
    if (sort_wlist[i]->value ==0){sort_wlist[i]->value =1;}  
    if (sort_wlist[i]->value ==B){sort_wlist[i]->value =0;}  
}
```

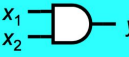
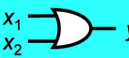

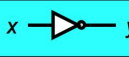
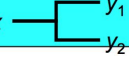
Figure. 11: DTC

Using PODEM-X for dynamic compression. After generating a test pattern for a primary fault, not every PI is assigned, so there are some unused PIs, wires and POs. Our Objective is trying to detect one or more secondary faults through these unused circuit. We call PODEM again after we found a test pattern of a primary fault, when calling PODEM for a secondary fault, we rewrite wire values with D and B to 1 or 0 and fix the values in the circuit instead of initialize whole circuit in original PODEM. Keep doing PODEM until every undetected fault is tried. Besides, we choose a fault that hasn't been detected (detected times=0) as secondary fault.

Sort the fault list order by CO:

Combinational observability is a way to measure whether the logical value of the wire is easy to be propagated or not, so we calculate all CO of wires in the circuit. We use greedy method to sort the fault list from large CO to small. By this way, we can generate test pattern for harder faults first. We can calculate CO by the following tables. We calculate the controllability of wires (topological order) and then calculate the observability value(reverse topological order) according to the following table.

CC⁰(N) & CC¹(N)

	CC ⁰ (y)	CC ¹ (y)
Primary inputs	1	1
	$\min[CC^0(x_1), CC^0(x_2)] + 1$	$CC^1(x_1) + CC^1(x_2) + 1$
	$CC^0(x_1) + CC^0(x_2) + 1$	$\min[CC^1(x_1), CC^1(x_2)] + 1$
	$\min[CC^0(x_1) + CC^0(x_2), CC^1(x_1) + CC^1(x_2)] + 1$	$\min[CC^0(x_1) + CC^1(x_2), CC^1(x_1) + CC^0(x_2)] + 1$
	$CC^1(x) + 1$	$CC^0(x) + 1$
	$CC^0(y_1) = CC^0(y_2) = CC^0(x)$	$CC^1(y_1) = CC^1(y_2) = CC^1(x)$

© National Taiwan University

Figure. 12: Combinational Controllability table

CO(N)

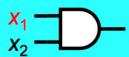



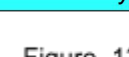
	CO(x ₁)
Primary outputs	0
	$CO(y) + CC^1(x_2) + 1$
	$CO(y) + CC^0(x_2) + 1$
	$CO(y) + \min[CC^0(x_2), CC^1(x_2)] + 1$
	$CO(y) + 1$
	$\min[CO(y_1), CO(y_2)]$

Figure. 13: Combinational Observability table

```
void reorder_fault_list(){
    calculate_observability();
    for (fptr f = first_fault; f; f = f->pnext_undetected) {
        fault_array.push_back(f);
    }
    sort(fault_array.begin(), fault_array.end(), myrule);

    fptr f;
    first_fault=NULL;
    for (int i=0; i<fault_array.size(); i++) {
        f = fault_array[i];
        f->pnext = first_fault;
        f->pnext_undetected = first_fault;
        first_fault = f;
    }
}
```

Figure. 14: Sort the fault list by CO

If we want to re-order the fault list by CO, we calculate the CO for each wire first. Next, we put all faults into a std::vector and use standard library algorithm to sort them by the number of CO. Finally, we rebuild the fault list.

4. Experimental results:

The proposed algorithms were implemented in the C and C++ language. All experiments were conducted on a Linux machine with two 6-core Xeon 2.3 GHz CPU and 32 GB RAM. ISCAS and advanced benchmark circuits which TA provided were selected for experiments.

(1) Basic Case:

-ndet 1

circuit number	fault coverage	test length	run time	fault coverage	test length	run time
C432	11.62 %	27	0.07 s	11.62 %	22	0.09 s
C499	94.77 %	166	0.28 s	94.56 %	82	0.50 s
C880	50.38 %	118	0.20 s	50.28 %	62	0.61 s
C1355	38.26 %	86	0.52 s	38.26 %	60	10.12 s
C2670	94.10 %	371	1.06 s	94.03 %	153	7.61 s
C3540	23.15 %	161	8.63 s	23.25 %	75	15.82 s
C6288	97.62 %	202	2.31 s	97.63 %	76	48.40 s
C7552	98.29 %	695	5.33 s	98.28 %	286	27.26 s

-ndet 1 -compression

-ndet 8

circuit number	fault coverage	test length	run time	fault coverage	test length	run time
C432	11.62 %	189	0.06 s	11.62 %	170	0.15 s
C499	95.02 %	936	0.45 s	95.02 %	567	2.92 s
C880	50.38 %	620	0.27 s	50.38 %	401	2.70 s
C1355	38.26 %	596	0.60 s	38.26 %	455	11.03s
C2670	94.10 %	1904	2.09 s	94.13 %	1158	25.99 s
C3540	23.25 %	965	9.57 s	23.25 %	550	45.27 s
C6288	97.66 %	991	5.33 s	97.66 %	382	88.16 s
C7552	98.31 %	3700	12.60 s	98.34 %	1962	197.28 s

-ndet 8 -compression

From our result, we can find that most cases remain same fault coverage after compression, and the test length is heavily reduced. However, we need to spend more CPU time to compress the test patterns. It's a tradeoff, but normally the low test cost is important than low CPU time.

(2) Advance Case:

-ndet 1

circuit name	fault coverage	test length	run time	fault coverage	test length	run time
adder	87.34 %	66	2.755 s	87.77 %	50	4.268 s
bar	98.63 %	241	1.393 s	98.63 %	110	14.11 s
dec	86.59 %	394	0.174 s	85.90 %	304	0.983 s
max	88.99 %	2477	65.95 s	88.99 %	1746	3930 s
multi	99.68 %	720	140.4 s	99.67 %	182	3588 s
sin	95.37 %	593	67.29 s	95.21 %	374	909.3 s

-ndet 1 -compression

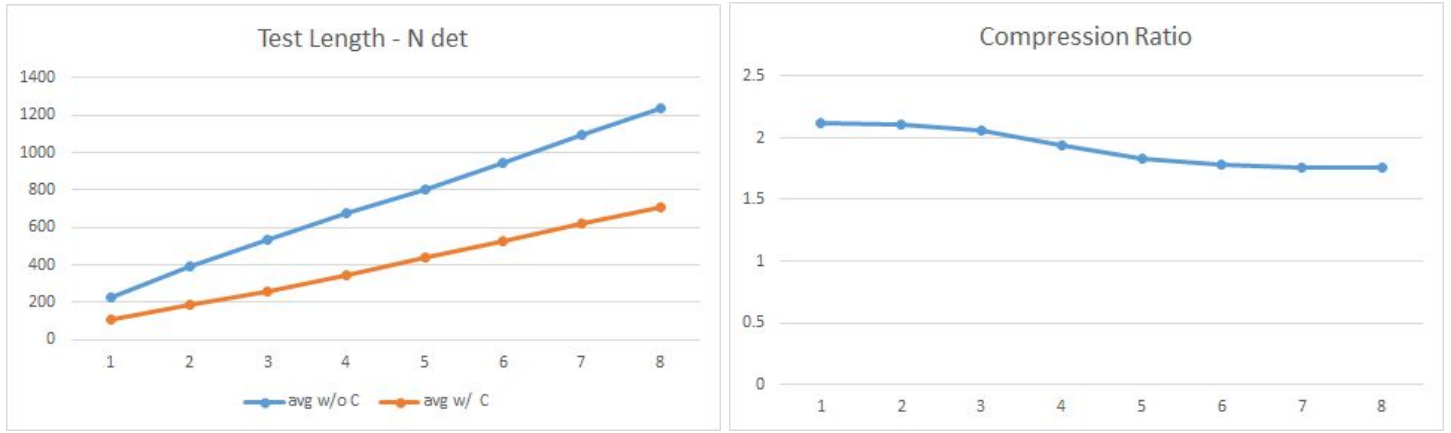
-ndet 8

circuit name	fault coverage	test length	run time	fault coverage	test length	run time
adder	87.53 %	253	2.979 s	87.67 %	187	8.928 s
bar	98.63 %	853	3.181 s	98.63 %	409	71.70 s
dec	86.59 %	2951	0.231 s	85.68 %	2378	2.705 s
max	89.00 %	17645	142.8 s	89.00 %	13891	6498 s
multi	99.68 %	3430	487.8 s	99.68 %	1331	6077 s
sin	95.92 %	4490	134.0 s	95.85 %	2822	1721 s

-ndet 8 -compression

For the given advance cases, first three cases are smaller, the last three cases are considerably larger. We can find that we spend much CPU time compressing the last three cases. Although it is time-consuming, we save lots of memory after compression.

(3) The average test length vs detection number N:



As the figure, we can find that the average test length is heavily reduced after compression. The more the detection number, the more the test length decrease. Besides, Our compression ratio is approximately 2 for each detection number N.

(4) The comparison of re-ordering (All cases in -ndet 8 -compression):

From large CO to small CO				From small CO to large CO		
circuit number	fault coverage	test length	run time	fault coverage	test length	run time
C432	11.62 %	170	0.15 s	11.62 %	170	0.62 s
C499	95.02 %	567	2.92 s	94.94 %	516	2.56 s
C880	50.38 %	401	2.70 s	50.38 %	414	4.14 s
C1355	38.26 %	455	11.03s	37.97 %	436	9.05 s
C2670	94.13 %	1158	25.99 s	94.10 %	1160	23.12 s
C3540	23.25 %	550	45.27 s	23.20 %	533	62.56 s
C6288	97.66 %	382	88.16 s	97.66 %	327	59.93 s
C7552	98.34 %	1962	197.28 s	98.34 %	1878	226.75 s

From large CO to small CO				Use origin fault list order		
circuit number	fault coverage	test length	run time	fault coverage	test length	run time
C432	11.62 %	170	0.15 s	11.62 %	172	0.51 s
C499	95.02 %	567	2.92 s	95.02 %	598	2.48 s
C880	50.38 %	401	2.70 s	50.38 %	426	3.73 s
C1355	38.26 %	455	11.03s	38.26 %	471	10.53 s
C2670	94.13 %	1158	25.99 s	94.10 %	1207	23.72 s
C3540	23.25 %	550	45.27 s	23.22 %	546	58.01 s
C6288	97.66 %	382	88.16 s	97.65 %	352	55.19 s
C7552	98.34 %	1962	197.28 s	98.33 %	1909	176.68 s

After computing CO, we sort the fault by high CO to low CO, because if we generate a test pattern for a fault that is hard to detect, then other faults that are easy to detect may be detected by this pattern too. In order to prove this speculation, we also show the result of sorting fault list form small CO to large CO and using origin fault list order. The answer is interesting, there are no optimal method for all cases. Different case requires different fault list order.

(5) The comparison of making decision on v1 or v2 first(All cases in -ndet 8 -compression):

Making decision on v1 first				Making decision on v2 first		
circuit number	fault coverage	test length	run time	fault coverage	test length	run time
C432	11.62 %	170	0.15 s	11.62 %	170	0.17 s
C499	95.02 %	567	2.92 s	95.60 %	529	2.45 s
C880	50.38 %	401	2.70 s	50.38 %	408	2.47 s
C1355	38.26 %	455	11.03s	38.04 %	464	13.41 s
C2670	94.13 %	1158	25.99 s	94.10 %	1136	26.62 s
C3540	23.25 %	550	45.27 s	23.21 %	551	60.38 s
C6288	97.66 %	382	88.16 s	97.65 %	370	82.77 s
C7552	98.34 %	1962	197.28 s	97.88 %	1985	251.34 s

Since v2 is more complicated, most people think we should generate v2 first, and then make it be a valid v1. However, we tried a different way. We make decision to generate a valid v1 before generating v2. The result show that there is no optimal way for all cases. Some cases have better performance when generating v1 first, while other cases don't.

5. References:

- (1) I.Hamzaoglu, J.Patel, "Test set compaction algorithms for combinational circuits", ICCAD 1998.
- (2) H. Ichihara, "Dynamic test compression using statistical coding", ATS 2001.
- (3) N.Devtaprasanna, "Methods for improving transition delay fault coverage using broadside tests", ITC 2005.
- (4) N.A. Touba, "Survey of Test Vector Compression Techniques", DTC 2006.
- (5) Xiang, Dong, et al. "Compact test generation with an Influence input measure for launch-on-capture transition fault testing, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 22.9 (2014).
- (6) Chien-Mo Li, "VLSI Testing lecture note chapter 9 and 15 " NTUGIEE.

6. Members and their contribution:

r06943155 楊承濂: Compression
r06943086 張奕凡: Compression
r06921048 李友岐: TDF-ATPG, N-Detection