An Automatic Interconnection Rectification Technique for SoC Design Integration

Chun-Yao Wang, Shing-Wu Tung and Jing-Yang Jou
Department of Electronics Engineering, National Chiao Tung University
Hsinchu 300, Taiwan, R.O.C.
Tel: 886-3-5726111, Fax: 886-3-5710580
e-mail:{wcyao,swtung,jyjou}@eda.ee.nctu.edu.tw

Abstract— This paper presents an automatic interconnection rectification (AIR) technique to correct the misplaced interconnection occurred in the integration of a SoC design automatically. The experimental results show that the AIR can correct the misplaced interconnection and therefore accelerates the integration verification of a SoC design.

I. Introduction

In the SoC era, system level integration and platform-based design [1] are evolving as a new paradigm in system designs, hence, design reuse and reusable building blocks (cores) trading are becoming popular. However, present design methodologies are not enough to deal with cores which come from different design groups and are mixed and matched to create a new system design. In particular, design verification is one of the most difficult task.

The focus of core-based design verification should be on how the cores communicate with each other [2]. However, prior to the interface verification, the interconnection between the cores in a SoC have to be verified first. This is because the SoC integrator has to connect a large number of ports in a SoC design. The likelihood of interconnection misplacements between the cores is high. Thus, the interconnection verification can be conducted as the first step to the interface verification between the cores in a SoC design.

By creating the testbenches at a high level, a connectivity-based design fault model, port order fault (POF), is proposed in [3]. This fault model is similar to the Type H design error ”incorrectly placed wire” in the logic level [4] [5]. The POF-based automatic verification pattern generation (AVPG) are also developed in [6] [7]. The AVPG are effective in generating the verification pattern set for detecting the misplacements of interconnection in a SoC design. However, the diagnosis and correction issues on the misplaced interconnection are even more important for SoC verification. Thus, to accelerate the SoC integration process, this paper presents an automatic interconnection rectification (AIR), which not only detects the erroneous interconnection among the cores, but also diagnoses and corrects them automatically.

Traditional diagnosis and correction algorithms in the logic level can be divided into two categories with respect to the underlying techniques: those based on symbolic techniques [8] [9] and those based on simulation techniques [5] [10] [11]. The approaches based on symbolic techniques can return valid correction and handle circuits with multiple errors well, however, they are not applicable to circuits that have no efficient Ordered Binary Decision Diagram (OBDD) [12] representation. Thus, to verify the interconnection among the IP cores with all description levels (soft, firm, and hard cores) embeded into a system, the AIR algorithm has to deal with IP cores that are described in different levels, for example, logic level, register transfer (RT) level, or even behavioral level. Consequently, the symbolic approach is inadequate to this application and the simulation based AIR algorithm is presented.

II. Preliminary

Definition 1: The type I POF is at least an output misplaced with an input. The type II POF is at least two inputs misplaced. The type III POF is at least two outputs misplaced [3].

It has been proven that the type II POFs **dominate** the other two types of POFs [6]. Thus, the AIR focuses on the **type II POFs**.

Definition 2:A **port sequence** is an input port numbers permutation. The **f**ault **f**ree **p**ort **s**equence (**FFPS**) is a port sequence that none of the input ports is misplaced. For an N-input core, the $N!$ permutations represent the $N!$ port sequences. Except the FFPS, the remaining (N!-1) port sequences are called **f**aulty **p**ort **s**equences (**FPSs**).

Example 1: The schematic representation of the FFPS 1234 and the FPS 1423 of BLK2 are shown in Fig. 1(a) and Fig. 1(b), respectively.

The undetected port sequences (UPSs) representation used in the AVPG [6] [7] is to implicitly represent the undetected port sequences remained in the fault set. Since it is also used in the AIR algorithm, we introduce it here briefly.

Example 2: Given an 8-input core, the input ports are numbered from 1 to 8. The (12345678) represents the UPSs that caused by all possible misplacements among the port numbers

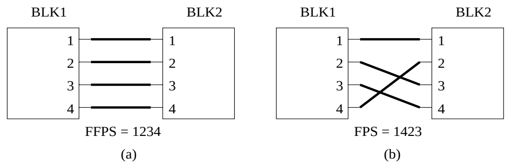


Fig. 1. The schematic representation of the FPS

pattern set	detected FPSs	remaining UPSs
——	——	Π_0= ini. UPSs
P $_1$	FPSs(P $_1$)	Π_1
P $_2$	FPSs(P $_2$)	Π_2
P $_3$	FPSs(P $_3$)	Π_3
⋮	⋮	⋮
P $_t$	FPSs(P $_t$)	Π_t = FFPS

Fig. 3. The relationship of P_i, FPSs(P_i) and remaining UPSs Π_i

in the same group, i.e., port 1 to port 8. The (125)(4)(3678) indicates the UPSs that caused by all possible misplacements among the port numbers 1, 2 and 5 and/or all possible misplacements among the port numbers 3, 6, 7 and 8. The (1)(2)(3)(4)(5)(6)(7)(8) represents 8!-1 POFs are all detected. If the UPSs representation is induced from (12345678) to (1)(2)(3)(4)(5)(6)(7)(8), all POFs are detected.

The environment and mechanism of POF verification, which exploits IEEE P1500 SECT, can be found in [6]. Thus, we omit describing them here due to the page limit.

III. The AIR Algorithm

A. AIR Overview

The input to the AIR is the simulation model of an IP core. The four stages of AIR are pattern generation, fault detection, fault diagnosis, and fault correction as shown in Fig. 2. In addition to these four stages, an instantaneous UPSs representation is associated with the AIR. This UPSs representation can indicate the remaining UPSs currently and guides the generation of further verification patterns. If the UPS is empty, the AIR is terminated and the interconnection in the integrated design are correct.

When a pattern set P_i is selected as the verification pattern set, some FPSs will be detected by P_i and the UPSs will be reduced from Π_{i-1} to Π_i where Π_i denotes the remaining UPSs after the verification by P_i. Then the pattern generation stage generates further verification pattern sets in the next iteration according to Π_i. Each P_i corresponds to a set of FPSs and is responsible for detecting them. The FPSs which are detected by P_i are denoted as FPSs(P_i). The relationship of P_i, FPSs(P_i), and UPSs Π_i is shown in Fig. 3. In Fig. 3, the initial UPSs are denoted as Π_0 and we assume the pattern set P_{i-1} is generated before P_i. When the last pattern set P_t is generated, the UPS is Π_t and it is the FFPS.

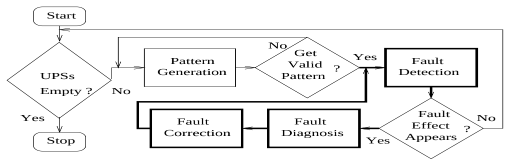


Fig. 2. The AIR flow

B. Pattern Generation

Definition 3: The set consists of all patterns with m 1s and (N-m) 0s is denoted as Θ^N_m, where $m \in [0, 1, 2, \cdots, N-1, N]$. The size of Θ^N_m is the number of patterns in Θ^N_m and is denoted as $|\Theta^N_m|$.
Example 3: For a 4-input core, Θ^4_0={0000}, $|\Theta^4_0|$ =1. Θ^4_1={1000, 0100, 0010, 0001}, $|\Theta^4_1|$ =4.
Theorem 1: Θ^N_m *can activate all (N!-1) POFs where m* $\in$ *[1, 2,* $\cdots$*, N-2, N-1].* [6]

According to Theorem 1, we arbitrarily apply one Θ^N_m to the inputs of core for $m \in$ [1, 2, $\cdots$, N-2, N-1]. Since $|\Theta^N_m|$ is smaller when m is closer to the end points of interval [1, 2, $\cdots$, N-1], **we select** m **from** 1 **up to** $\lfloor N/2 \rfloor$ **or from** N-1 **down to** $\lfloor N/2 \rfloor$.

Given an 8-input combinational core, the initial UPSs Π_0 are (12345678). The simulation results of Θ^8_1 are shown in Fig. 4 and are represented in symbolic output representation. The patterns with the same output are grouped into one set. Θ^8_1 patterns can be grouped into two sets, S1 and S2. When we select the smaller set S1 as the verification pattern set P_1, Π_1 have to be derived as well. The following paragraphs describe how to calculate the UPSs Π_i when P_i is generated.
Definition 4: Given a set of patterns S with the same length, we count the number of digits 1 in the same bit position to form a vector with the same length. This vector is called the characteristic vector (CV) of S and is denoted as CV_S.
Theorem 2: *A pattern set S', which consists of all same output patterns* $\in \Theta^N_m$*, turns to S after a FPS* λ*. If CV_S'* $\neq$ *CV_S, then the FPS* λ *will be detected by S'.*

In Fig. 4, we selected S1 as P_1, and according to Theorem 2, any port sequence which changes CV_S1 will be detected by S1. **Thus, the port misplacement that cannot change CV_S1 are regarded as the UPSs** Π_1.
Corollary 1: *If a pattern set S is selected as the verification pattern set* P_i*, the UPSs* Π_i *can be obtained by applying the same grouping result of CV_S over the UPSs* Π_{i-1}.

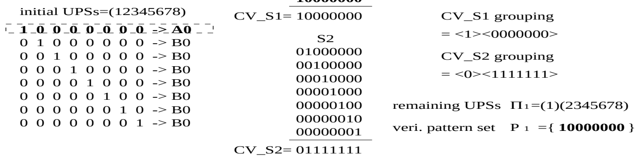

```
initial UPSs=(12345678)                  S1
1 0 0 0 0 0 0 0 -> A0                    10000000
0 1 0 0 0 0 0 0 -> B0         CV_S1= 10000000          CV_S1 grouping
0 0 1 0 0 0 0 0 -> B0                                   = <1><0000000>
0 0 0 1 0 0 0 0 -> B0                    S2
0 0 0 0 1 0 0 0 -> B0                    01000000       CV_S2 grouping
0 0 0 0 0 1 0 0 -> B0                    00100000       = <0><1111111>
0 0 0 0 0 0 1 0 -> B0                    00010000
0 0 0 0 0 0 0 1 -> B0                    00001000
                                         00000100    remaining UPSs  Π1=(1)(2345678)
                                         00000010
                                         00000001    veri. pattern set   P1 ={ 10000000 }
                              CV_S2= 01111111
```

Fig. 4. The simulation outputs of Θ^8_1

When S1 is selected as P_1. The grouping result of CV_S1 is <1><0000000>. Therefore, according to Corollary 1, Π_1 is obtained by the same grouping of CV_S1 over the UPSs Π_0=(12345678) directly and Π_1 become (1)(2345678). These results are also shown in Fig. 4. If the port sequence λ of the real interconnection in the integrated design $\in$ FPSs(P_1), when applying P_1 into the integrated design, λ will be detected.

To demonstrate the interconnection detection, diagnosis, and correction procedures, we assume the FPS λ of this example is given and is 83762451.

C. Fault Detection

We apply P_1 {10000000}into the design with the FPS λ 83762451, and find that the corresponding output of P_1 is B0 as shown in the first row of Fig. 5(a). Since the fault free output is A0, thus the fault effect appears and λ is detected by P_1.

D. Fault Diagnosis

We apply P_1 into the integrated design, and realize that the real applied input is not P_1 by observing the unexpected output B0. Since there are seven patterns that produce the B0 output, we do not know exactly what the actual applied pattern is. Nevertheless, **we know the actual applied pattern which produces the A0 output instead**. We simulate Θ_1^8 patterns with the FPS 83762451 and observe the outputs until the output is A0. These results are shown in Fig. 5(a). From Fig. 5(a), we find that when the last pattern {00000001} is the output becomes A0. This result implies that the FPS λ turns the pattern {00000001} to {10000000}. We put the pattern {00000001} into S1' and assume S1 is {10000000}. Then we calculate CV_S1'=00000001 and CV_S1=10000000. Afterward, the misplaced ports can be identified by comparing CV_S1' and CV_S1.

E. Fault Correction

Comparing CV_S1' and CV_S1, we observe that the 1st digit and the 8th digit of CV_S1' and CV_S1 are different. Thus, we switch the port 1 with the port 8 to let CV_S1 be the same as CV_S1'. The corrected FPS becomes 13762458 and is shown in Fig. 5(b).

Definition 5: The exchange of two ports is defined as a 2_switch. The exchange of port x and port y is denoted as 2_switch(x, y).

Definition 6: Given a set of n-bit patterns S', when a 2-switch is applied on S' and CV_S' is invariant, the 2-switch is called a CV invariant fault of S' (CVIF(S')). Otherwise, it is called a CV variant fault of S' (CVVF(S')).

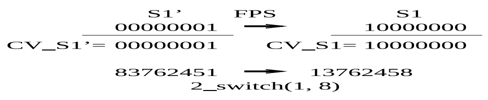
Theorem 3: *Given a set of n-bit patterns S' and a FPS λ. There exists a finite sequence of 2_switches, 2_$switch_1$ $\sim$ 2_$switch_l$, to convert the FFPS into λ, and this sequence of 2_switches must be in one of the following three categories:*

(I): 2_$switch_1$ $\sim$ 2_$switch_l$ are all CVIFs(S')

(II): 2_$switch_1$ $\sim$ 2_$switch_i$ are CVIFs(S') and 2_$switch_{i+1}$ $\sim$ 2_$switch_l$ are CVVFs(S') where $1 \leq i \leq$ l-1

Pattern	Outputs	
	Fault Free	FPS 83762451
10000000	A0	B0
01000000	B0	B0
00100000	B0	B0
00010000	B0	B0
00001000	B0	B0
00000100	B0	B0
00000010	B0	B0
00000001	B0	A0

(a)



(b)

P 1	Outputs	
	Fault Free	FPS 13762458
10000000	A0	A0

(c)

Fig. 5. Rectification processes of Θ_1^8

(III): 2_$switch_1$ $\sim$ 2_$switch_l$ are all CVVFs(S')

In this example, the 2_switch(1,8) is a CVVF(S1'). Therefore, according to Theorem 3, the corrected FPS λ' is a sequence of CVIFs(S1'). Theorem 3 guarantees that the CVIF(S1') are the only possible faults which we have to deal with in the succeeding iteration.

Since each P_i corresponds to a set of FPSs, FPSs(P_i), and is responsible for detecting them. If the corrected FPS λ' $\notin$ FPSs(P_i), P_i is not able to detect and correct any other faulty ports in λ'. At this time, further verification pattern set P_k, where $k > i$, are generated to detect and correct the other faulty ports in λ'. However, how can we know the corrected FPS λ' $\notin$ FPS(P_i) and cannot be corrected by P_i anymore ? The following corollary states the condition that has to be satisfied so that the corrected FPS λ' $\notin$ FPSs(P_i) (or $\in$ UPSs Π_i).

Corollary 2: *If the actual outputs are consistent with the expected ones when applying P_i into the integrated design with a corrected FPS λ', then the corrected FPS λ' $\in$ UPSs Π_i.*

Therefore, according to Corollary 2, we apply P_1 into the integrated design again with the corrected FPS λ' 13762458 to see whether the λ' can be further corrected by P_1. In Fig. 5(c), we find that the outputs of P_1 are both A0, thus, λ' $\in$ UPSs Π_1 (1)(2345678). At this time, we move to the next iteration to generate further pattern sets P_k ($k > 1$) to detect and correct the remaining faulty ports.

F. Summary

The FPS is corrected from 83762451 to 13762458, and the UPSs representation is reduced from (12345678) to (1)(2345678). These results illustrate that the UPSs representation presents the corrected FPS appropriately. The succeeding iterations in the AIR follow the same flow to rectify the misplaced interconnection. However, due to the page limit, we skip these repeated demonstration of the AIR algorithm here.

The success of the AIR depends on the pattern generation stage strongly. Since the pattern generation stage will search all Θ_m^N, for m=1, 2, $\cdots$, N-1 if necessary, it is a complete algorithm [6] [7]. This complete pattern generation algorithm leads the AIR algorithm to be complete as well.

TABLE I
EXPERIMENTAL RESULTS ON COMBINATIONAL BENCHMARKS

bench	parameters		blind connection		guided connection	
	|PI|	gcs.	a/b	time	a/b	time
c17	5	6	5/5	0.1	3/3	0.1
c880	60	357	60/60	182	13/13	96
c1355	41	514	41/41	179	8/8	50
c1908	33	880	32/32	167	6/6	46.5
c432	36	160	36/36	43	6/6	7
c499	41	202	39/39	59	9/9	16.6
c3540	50	1667	48/48	882	12/12	268
c5315	178	2290	178/178	11505	34/34	2692
c2670	233	1161	232/232	10119	37/37	1937
c7552	207	3466	207/207	26689	39/39	6477
c6288	32	2416	32/32	773	8/8	466

G. The Sequential AIR

The development of the sequential AIR is based on the same assumption as the combinational AIR, i.e., the CUV is pre-verified and fault free. The fault occurs only at the interconnection between the cores. For the testability concern, most sequential cores are designed with scan chains. Thus, here we assume that the sequential cores in the experiments are scan-testable. These sequential cores can be set in arbitrary state and therefore they can be seen as combinational ones. Consequently, the AIR algorithm used in the combinational cores is applicable to the sequential ones. The only difference is that the sequential cores have to be set to a state by sequential AIR before evaluating outputs.

IV. EXPERIMENTAL RESULTS

The heuristic AIR, which adds the iteration counter to bound the processing time, has been integrated into the SIS [13] environment. Experiments are conducted over a set of benchmarks, which are in BLIF format. The simulation information of the BLIF benchmarks imitate the simulation model of IP cores.

Table I summaries the experimental results of the heuristic AIR. The $|\text{PI}|$ represents the number of inputs. The gate counts (gcs.) indicates the scale of a benchmark. The a/b presents ”number of corrected ports/number of faulty ports”.

The iteration bound in the experiment was set to 100. The AIR algorithm will be terminated automatically if the iteration counter is over the bound or the UPSs representation becomes empty. At the end of AIR, the number of corrected ports, and CPU time are returned. The number of corrected ports is obtained by comparing the final FPS with the FFPS. The CPU time is measured in second on an Ultra Sparc II workstation.

According to Table I and II, the faulty ports of each benchmark **are all corrected** except s5378 and s13207, and the processing time of each benchmark is acceptable. These results demonstrate that the heuristic AIR is able to correct the misplaced ports within reasonable efforts.

V. CONCLUSIONS

The AIR technique provides a solution to integrate the cores with correct interconnection automatically. Therefore this technique can reduce the time on design verification in core-based design methodology.

TABLE II
EXPERIMENTAL RESULTS ON SEQUENTIAL BENCHMARKS

bench	parameters			blind connection		guided connection	
	|PI|	gcs.	FFs	a/b	time	a/b	time
s1196	14	529	18	13/13	24	5/5	14.2
s1238	14	508	18	13/13	31	3/3	8.9
s1488	8	653	6	8/8	39	4/4	30
s1494	8	647	6	6/6	62	2/2	41
s15850	14	9786	597	13/13	1382	5/5	1252
s208	11	104	8	11/11	29.1	4/4	21
s27	4	10	3	4/4	0.1	2/2	0.1
s5378	35	2779	164	**29/33**	3149	9/9	3083
s641	35	379	19	35/35	136	9/9	81
s713	35	393	19	33/33	137	7/7	85
s820	18	289	5	17/17	1005	5/5	979
s832	18	287	5	18/18	1014	8/8	1004
s838	35	446	32	33/33	1111	8/8	908
s9234	36	5597	211	35/35	8133	9/9	7620
s444	3	181	21	3/3	62	2/2	56
s510	19	211	6	18/18	130	6/6	105
s344	9	160	15	9/9	4.5	4/4	2.6
s349	9	161	15	9/9	7.8	3/3	2.1
s382	3	158	21	3/3	44	3/3	44
s386	7	159	6	5/5	46	3/3	40
s400	3	162	21	3/3	49.6	3/3	46
s13207	31	8027	669	**23/28**	19114	4/4	9352
s1423	17	657	74	17/17	1586	5/5	375
s6669	83	3080	239	83/83	12954	16/16	10223
s4863	49	2342	104	49/49	7733	11/11	4551
s1269	18	569	37	16/16	56	4/4	24.8
s1512	29	780	57	29/29	2489	6/6	1932
s3271	26	1572	116	25/25	6254	7/7	4876
s3330	40	1789	132	40/40	805	6/6	156
s3384	43	1685	183	43/43	6377	9/9	4458
b10	11	153	17	10/10	170	3/3	157
b11	7	510	31	7/7	230	4/4	185
b12	5	880	121	5/5	482	3/3	315
b13	10	255	53	8/8	226	3/3	210
b14	32	5401	245	31/31	16332	7/7	13453
b15	37	7092	449	37/37	18742	10/10	16667

ACKNOWLEDGEMENTS

This work was supported in part by ROC National Science Council under Grant NSC90-2215-E-009-106

REFERENCES

[1] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd, ”Surviving the SoC revolution - a guide to platform-based design,” Kluwer Academic Publishers, 1999.

[2] J. A. Rowson, and A. Sangiovanni-Vincentelli, ”Interface-based design,” *in Proc. Design Automation Conference*, pp.178-183, Jun. 1997.

[3] S.-W. Tung, and J.-Y. Jou, ”A logic fault model for library coherence checking,” *Journal of Information Science and Engineering*, pp.567-586, Sep. 1998.

[4] M. S. Abadir, J. Ferguson, and T. E. Kirkland ”Logic design verification via test generation,” *IEEE Transactions on Computer-Aided Design*, vol.7, no.1, pp.138-148, Jan. 1988.

[5] A. Veneris, and I. N. Hajj, ”Design error diagnosis and correction via test vector simulation,” *IEEE Transactions on Computer-Aided Design*, vol.18, no.12, pp.1803-1816, Dec. 1999.

[6] C.-Y. Wang, S.-W. Tung, and J.-Y. Jou, ”On automatic verification pattern generation for SoC with port order fault model,” *IEEE Transactions on Computer-Aided Design*, vol.21, no.4, pp.466-479, Apr. 2002.

[7] C.-Y. Wang, S.-W. Tung, and J.-Y. Jou, ”An automorphic approach to verification pattern generation for SoC design verification using port order fault model,” *IEEE Transactions on Computer-Aided Design*, vol.21, no.10, Oct. 2002.

[8] S.-Y. Huang, K.-C. Chen, and K.-T. Cheng ”Incremental logic rectification,” *in Proc. VLSI Test Symposium*, pp.143-149, 1997.

[9] M. Fujita, Y. Tamiya, Y. kukimoto, and K.-C. Chen, ”Application of Boolean unification to combinational synthesis,” *in Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, pp.510-513. 1991.

[10] S.-Y. Huang, K.-C. Chen, and K.-T. Cheng ”ErrorTracer: design error diagnosis based on fault simulation techniques,” *IEEE Transactions on Computer-Aided Design*, vol.18, no.9, pp.1341-1352, Sep. 1999.

[11] I. Pomeranz and S. M. Reddy ”On error correction in macro-based circuits,” *IEEE Transactions on Computer-Aided Design*, vol.16, no.10, pp.1088-1100, Oct. 1997.

[12] R. E. Bryant ”Graph-based algorithms for Boolean function manipulation,” *IEEE Transactions on Computer*, vol.C-35, no.8, pp.677-691, 1986.

[13] E. M. Sentovich, K. T. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli, ”Sequential circuit design using synthesis and optimization,” *in Proc. IEEE International Conference on Computer Design*, pp.328-333, Oct. 1992.